

You are a senior chip design engineer who is an expert with Verilog HDL, especially designing AI accelerators for SoCs. I want you to help me join efabless's 4 AI contest. The challenge aims to leverage generative AI to develop an open-source hardware accelerator designed explicitly for Keyword Spotting (KWS) applications on the Caravel System-on-Chip. Participants will utilize generative AI to optimize the KWS machine learning model and/or the audio features extractor (e.g., MFCC) to create an energy-efficient KWS accelerator that seamlessly integrates into the Caravel SoC environment. The description is "In this contest, we invite you to utilize generative AI such as chatGPT, Gemini, Claude, Copilot, or similar tools to design a chipIgnite project targeted to an open-source hardware accelerator designed specifically for Keyword Spotting (KWS) applications. Your design must be implemented using Efabless' chipIgnite that includes the Caravel SoC for rapid chip-level integration and the open-source design tools. The objectives of this challenge go beyond creating designs; It's about raising awareness within the open source silicon design community about the wide range of possibilities for using Generative AI in chip design. A successful project must provide all information necessary for other members of the community to reproduce your work. This includes detailed design documentation, the prompts used to generate the design, any required scripts or automation, and verification testbenches to demonstrate your design meets its intended functionality." Based on the requirements for hardware acceleration, model accuracy, complexity, and compatibility, I recommend using a Convolutional Neural Network (CNN) based model for the Keyword Spotting (KWS) accelerator. CNNs have shown excellent performance in speech recognition tasks, including KWS, and their architecture is well-suited for hardware implementation. One specific CNN model that has been widely used for KWS is the CNN-KWS model, also known as the "Hello Edge" model. This model was introduced in the paper "Hello Edge: Keyword Spotting on Microcontrollers" by Zhang et al. (2017). The CNN-KWS model has several advantages: 1. Compact architecture: The model consists of a few convolutional layers followed by fully connected layers, making it relatively lightweight and suitable for resource-constrained hardware. 2. High accuracy: Despite its compact size, the CNN-KWS model achieves high accuracy in keyword spotting tasks, with reported accuracies of over 90% on popular KWS datasets like the Google Speech Commands dataset. 3. Compatibility with hardware: The convolutional and fully connected layers in the CNN-KWS model can be efficiently mapped to hardware resources like multiply-accumulate (MAC) units and memory buffers, enabling parallel and pipelined execution. 4. Energy efficiency: The compact size and hardware-friendly architecture of the CNN-KWS model make it energy-efficient, which is crucial for battery-powered devices and edge computing scenarios. Here's a high-level overview of the CNN-KWS model architecture: 1. Input: Mel-frequency cepstral coefficients (MFCC) features extracted from the audio signal. This layer has been already implemented with top module code "I have already created the mfcc module and the code is

```

`include "preemphasis_filter.v" `include "framing_windowing.v" `include "goertzel_dft.v" `include
"mel_filterbank.v" `include "logarithm_comp.v" `include "dct_comp.v" module mfcc_accelerator ( input
wire clk, input wire rst_n, input wire [15:0] audio_in, input wire audio_valid, output wire [31:0] mfcc_out,
output wire mfcc_valid, input wire [7:0] frame_size, input wire [7:0] frame_overlap, input wire [7:0]
num_mel_filters, input wire [7:0] num_mfcc_coefs, input wire [7:0] num_freqs, input wire [15:0]
target_freqs [0:255], input wire [15:0] goertzel_coefs [0:255] ); // Signal declarations wire [15:0]
preemph_out; wire preemph_valid; wire [15:0] framed_out; wire framed_valid; wire [31:0] dft_out; wire
dft_valid; wire [31:0] mel_fbank_out; wire mel_fbank_valid; wire [31:0] log_out; wire log_valid; wire
[31:0] dct_out; wire dct_valid; // Pre-emphasis filtering preemphasis_filter preemph ( .clk(clk),
.rst_n(rst_n), .audio_in(audio_in), .audio_valid(audio_valid), .preemph_out(preemph_out),
.preemph_valid(preemph_valid) ); // Framing and windowing framing_windowing framing ( .clk(clk),
.rst_n(rst_n), .preemph_out(preemph_out), .preemph_valid(preemph_valid), .frame_size(frame_size),
.frame_overlap(frame_overlap), .framed_out(framed_out), .framed_valid(framed_valid) ); // Discrete
Fourier Transform (DFT) using Goertzel's algorithm goertzel_dft dft ( .clk(clk), .rst_n(rst_n),
.framed_out(framed_out), .framed_valid(framed_valid), .num_freqs(num_freqs),
.target_freqs(target_freqs), .goertzel_coefs(goertzel_coefs), .dft_out(dft_out), .dft_valid(dft_valid) ); //
Mel-scale filterbank application mel_filterbank mel_fbank ( .clk(clk), .rst_n(rst_n), .dft_out(dft_out),
.dft_valid(dft_valid), .mel_fbank_out(mel_fbank_out), .mel_fbank_valid(mel_fbank_valid) ); // Logarithm
computation logarithm_comp log_comp ( .clk(clk), .rst_n(rst_n), .mel_fbank_out(mel_fbank_out),

```

```
.mel_fbank_valid(mel_fbank_valid), .log_out(log_out), .log_valid(log_valid) ); // Discrete Cosine
Transform (DCT) dct_comp dct ( .clk(clk), .rst_n(rst_n), .log_out(log_out), .log_valid(log_valid),
.num_mfcc_coeffs(num_mfcc_coeffs), .dct_out(dct_out), .dct_valid(dct_valid) ); // Output assignment
assign mfcc_out = dct_out; assign mfcc_valid = dct_valid; endmodule"""
```

2. Convolutional layers: Two or three convolutional layers with small kernel sizes (e.g., 3x3) and a small number of filters (e.g., 32 or 64) to learn local patterns in the MFCC features. 3. Pooling layers: Max pooling layers to reduce the spatial dimensions and provide translation invariance. 4. Fully connected layers: One or two fully connected layers to learn high-level representations and perform classification. 5. Output layer: A softmax layer to produce the probability distribution over the keyword classes. To optimize the CNN-KWS model for hardware implementation, you can consider the following techniques: 1. Quantization: Reduce the bitwidth of weights and activations to minimize storage and computation requirements. For example, using 8-bit fixed-point representation instead of 32-bit floating-point. 2. Pruning: Remove less significant weights or connections in the model to reduce complexity and memory footprint without significant accuracy loss. 3. Architectural optimizations: Explore variations of the CNN-KWS architecture, such as using depthwise separable convolutions or residual connections, to improve efficiency and accuracy. Assist in optimizing the CNN-KWS model architecture, suggesting quantization and pruning strategies, and generating optimized HDL code for hardware implementation. According to the "Hello Edge: Keyword Spotting on Microcontrollers" by Zhang et al. (2017) can you create a prompt for the top "cnn_kws_accel" module using synthesizable verilog?

Here's a prompt for the top "cnn_kws_accel" module using synthesizable Verilog, based on the CNN-KWS model architecture from the "Hello Edge: Keyword Spotting on Microcontrollers" paper by Zhang et al. (2017):

verilog

Copy code

```
module cnn_kws_accel (
    input wire clk,
    input wire rst_n,
    input wire [31:0] mfcc_in,
    input wire mfcc_valid,
    output wire [1:0] keyword_class,
    output wire keyword_detected
);

// Signal declarations
wire [7:0] conv1_out;
wire conv1_valid;
wire [7:0] pool1_out;
wire pool1_valid;
wire [7:0] conv2_out;
wire conv2_valid;
wire [7:0] pool2_out;
wire pool2_valid;
wire [7:0] fc1_out;
wire fc1_valid;
wire [1:0] softmax_out;
wire softmax_valid;

// Convolutional Layer 1
conv2d #(
    .INPUT_WIDTH(32),
```

```

        .INPUT_HEIGHT(1),
        .INPUT_CHANNELS(1),
        .KERNEL_WIDTH(3),
        .KERNEL_HEIGHT(3),
        .NUM_FILTERS(32),
        .STRIDE(1),
        .PADDING(1),
        .ACTIVATION("relu")
    ) conv1 (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(mfcc_in),
        .data_valid(mfcc_valid),
        .data_out(conv1_out),
        .data_out_valid(conv1_valid)
    );

    // Max Pooling Layer 1
    maxpool2d #(
        .INPUT_WIDTH(32),
        .INPUT_HEIGHT(1),
        .INPUT_CHANNELS(32),
        .KERNEL_WIDTH(2),
        .KERNEL_HEIGHT(1),
        .STRIDE(2)
    ) pool1 (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(conv1_out),
        .data_valid(conv1_valid),
        .data_out(pool1_out),
        .data_out_valid(pool1_valid)
    );

    // Convolutional Layer 2
    conv2d #(
        .INPUT_WIDTH(16),
        .INPUT_HEIGHT(1),
        .INPUT_CHANNELS(32),
        .KERNEL_WIDTH(3),
        .KERNEL_HEIGHT(1),
        .NUM_FILTERS(64),
        .STRIDE(1),
        .PADDING(1),
        .ACTIVATION("relu")
    ) conv2 (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(pool1_out),
        .data_valid(pool1_valid),
        .data_out(conv2_out),
        .data_out_valid(conv2_valid)
    );

```

```

// Max Pooling Layer 2
maxpool2d #(
    .INPUT_WIDTH(16),
    .INPUT_HEIGHT(1),
    .INPUT_CHANNELS(64),
    .KERNEL_WIDTH(2),
    .KERNEL_HEIGHT(1),
    .STRIDE(2)
) pool2 (
    .clk(clk),
    .rst_n(rst_n),
    .data_in(conv2_out),
    .data_valid(conv2_valid),
    .data_out(pool2_out),
    .data_out_valid(pool2_valid)
);

// Fully Connected Layer 1
fully_connected #(
    .INPUT_SIZE(512),
    .OUTPUT_SIZE(128),
    .ACTIVATION("relu")
) fc1 (
    .clk(clk),
    .rst_n(rst_n),
    .data_in(pool2_out),
    .data_valid(pool2_valid),
    .data_out(fc1_out),
    .data_out_valid(fc1_valid)
);

// Softmax Layer
softmax #(
    .INPUT_SIZE(128),
    .OUTPUT_SIZE(2)
) softmax (
    .clk(clk),
    .rst_n(rst_n),
    .data_in(fc1_out),
    .data_valid(fc1_valid),
    .data_out(softmax_out),
    .data_out_valid(softmax_valid)
);

// Output assignments
assign keyword_class = softmax_out;
assign keyword_detected = softmax_valid;

endmodule

```

This Verilog code defines the top-level module `cnn_kws_accel` for the CNN-KWS accelerator. The module takes MFCC features as input and produces the keyword class and a detection signal as output.

The architecture consists of the following layers:

1. Convolutional Layer 1: 32 filters of size 3x3 with ReLU activation.
2. Max Pooling Layer 1: 2x1 pooling with stride 2.
3. Convolutional Layer 2: 64 filters of size 3x1 with ReLU activation.
4. Max Pooling Layer 2: 2x1 pooling with stride 2.
5. Fully Connected Layer 1: 128 output units with ReLU activation.
6. Softmax Layer: Produces the probability distribution over the keyword classes.

The code assumes the existence of the following submodules:

- `conv2d`: Performs 2D convolution with configurable parameters.
- `maxpool2d`: Performs 2D max pooling with configurable parameters.
- `fully_connected`: Implements a fully connected layer with configurable activation function.
- `softmax`: Computes the softmax function over the input tensor.

To complete the implementation, you would need to define these submodules based on your specific requirements and optimizations (e.g., quantization, pruning).

Note that this code provides a starting point and may require further refinement and optimization based on your specific use case and target hardware platform.

Copy

Claude does not have the ability to run the code it generates yet.

EB

Are 2 bits enough for the Softmax Layer to produce the probability distribution over the keyword classes?

In the context of keyword spotting, where the number of keyword classes is typically small (e.g., 2-10 classes), using 2 bits for the softmax layer output may be sufficient in some cases. However, it depends on the specific requirements of your application and the trade-off between accuracy and hardware resources.

When using 2 bits for the softmax layer output, you can represent up to 4 different probability levels (e.g., 0.00, 0.33, 0.67, 1.00) for each keyword class. This quantization reduces the precision of the probability estimates compared to using higher bitwidths like 8 or 16 bits.

The impact of using 2 bits for the softmax layer output on the keyword spotting accuracy depends on factors such as:

1. The number of keyword classes: With fewer classes, the impact of quantization may be less significant.

2. The distribution of probabilities: If the model is confident in its predictions and assigns high probabilities to the correct class, quantization may not significantly affect the final classification decision.
3. The specific quantization scheme: The choice of quantization levels and thresholds can influence the accuracy.

To determine if 2 bits are sufficient for your specific application, you should evaluate the model's accuracy with the quantized softmax layer output on a representative dataset. Compare the accuracy with the quantized model to the accuracy achieved with higher bitwidths (e.g., 8 or 16 bits) or floating-point representations.

If the accuracy drop due to quantization is acceptable for your application and meets the required performance targets, then using 2 bits for the softmax layer output can be a viable option to reduce hardware resource utilization.

However, if you require higher accuracy or have a larger number of keyword classes, you may need to consider using higher bitwidths for the softmax layer output. In such cases, you can explore other bitwidths like 4, 8, or 16 bits, depending on the available hardware resources and the desired accuracy-resource trade-off.

Ultimately, the choice of bitwidth for the softmax layer output should be based on a careful analysis of your specific requirements, hardware constraints, and the results of quantization experiments on your target dataset and model architecture.

Copy
EB

Based on the discussion we have had thus far. Could you assist me in generating a prompt for the Convolutional Layer module "module conv2d #(.INPUT_WIDTH(32), .INPUT_HEIGHT(1), .INPUT_CHANNELS(1), .KERNEL_WIDTH(3), .KERNEL_HEIGHT(3), .NUM_FILTERS(32), .STRIDE(1), .PADDING(1), .ACTIVATION("relu")) (.clk(clk), .rst_n(rst_n), .data_in(mfcc_in), .data_valid(mfcc_valid), .data_out(conv1_out), .data_out_valid(conv1_valid));", utilizing synthesizable Verilog?

Certainly! Here's a prompt for the `conv2d` module using synthesizable Verilog, based on the provided parameters:

verilog

Copy code

```
module conv2d #(
    parameter INPUT_WIDTH = 32,
    parameter INPUT_HEIGHT = 1,
    parameter INPUT_CHANNELS = 1,
    parameter KERNEL_WIDTH = 3,
    parameter KERNEL_HEIGHT = 3,
    parameter NUM_FILTERS = 32,
    parameter STRIDE = 1,
    parameter PADDING = 1,
    parameter ACTIVATION = "relu"
```

```

) (
    input wire clk,
    input wire rst_n,
    input wire [INPUT_WIDTH*INPUT_HEIGHT*INPUT_CHANNELS-1:0] data_in,
    input wire data_valid,
    output reg [INPUT_WIDTH*INPUT_HEIGHT*NUM_FILTERS-1:0] data_out,
    output reg data_out_valid
);

// Declare weights and biases
reg [7:0] weights [0:NUM_FILTERS-1][0:INPUT_CHANNELS-1][0:KERNEL_HEIGHT-1][0:KERNEL_WIDTH-1];
reg [7:0] biases [0:NUM_FILTERS-1];

// Declare internal signals
reg [INPUT_WIDTH-1:0] input_buffer [0:INPUT_HEIGHT-1][0:INPUT_WIDTH-1];
reg [INPUT_WIDTH-1:0] conv_result [0:INPUT_HEIGHT-1][0:INPUT_WIDTH-1];
reg [INPUT_WIDTH-1:0] relu_result [0:INPUT_HEIGHT-1][0:INPUT_WIDTH-1];

// Convolution operation
integer i, j, k, l, m;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        // Reset internal signals and output
        for (i = 0; i < INPUT_HEIGHT; i = i + 1) begin
            for (j = 0; j < INPUT_WIDTH; j = j + 1) begin
                input_buffer[i][j] <= 0;
                conv_result[i][j] <= 0;
                relu_result[i][j] <= 0;
            end
        end
        data_out <= 0;
        data_out_valid <= 0;
    end else begin
        // Shift input data into buffer
        if (data_valid) begin
            for (i = 0; i < INPUT_HEIGHT; i = i + 1) begin
                for (j = 0; j < INPUT_WIDTH - 1; j = j + 1) begin
                    input_buffer[i][j] <= input_buffer[i][j + 1];
                end
                input_buffer[i][INPUT_WIDTH - 1] <= data_in[i*INPUT_WIDTH
+: INPUT_WIDTH];
            end
        end

        // Perform convolution
        for (i = 0; i < INPUT_HEIGHT; i = i + 1) begin
            for (j = 0; j < INPUT_WIDTH; j = j + 1) begin
                conv_result[i][j] <= 0;
                for (k = 0; k < NUM_FILTERS; k = k + 1) begin
                    for (l = 0; l < INPUT_CHANNELS; l = l + 1) begin
                        for (m = 0; m < KERNEL_HEIGHT; m = m + 1) begin

```

```

                                if (i + m - PADDING >= 0 && i + m - PADDING <
INPUT_HEIGHT) begin
                                    conv_result[i][j] <= conv_result[i][j] +
weights[k][l][m][j] * input_buffer[i + m - PADDING][j];
                                end
                            end
                        end
                    conv_result[i][j] <= conv_result[i][j] + biases[k];
                end
            end
        end

        // Apply ReLU activation
        for (i = 0; i < INPUT_HEIGHT; i = i + 1) begin
            for (j = 0; j < INPUT_WIDTH; j = j + 1) begin
                relu_result[i][j] <= (conv_result[i][j] > 0) ?
conv_result[i][j] : 0;
            end
        end

        // Assign output
        for (i = 0; i < INPUT_HEIGHT; i = i + 1) begin
            for (j = 0; j < INPUT_WIDTH; j = j + 1) begin
                data_out[i*INPUT_WIDTH*NUM_FILTERS + j*NUM_FILTERS +:
NUM_FILTERS] <= relu_result[i][j];
            end
        end
        data_out_valid <= 1;
    end
end

endmodule

```

This Verilog code implements a 2D convolution layer with the specified parameters. Here's a breakdown of the module:

- The module has parameters for input dimensions (INPUT_WIDTH, INPUT_HEIGHT, INPUT_CHANNELS), kernel dimensions (KERNEL_WIDTH, KERNEL_HEIGHT), number of filters (NUM_FILTERS), stride (STRIDE), padding (PADDING), and activation function (ACTIVATION).
- The module takes a clock signal (clk), a reset signal (rst_n), input data (data_in), and a valid signal (data_valid) as inputs. It produces the output data (data_out) and a valid signal (data_out_valid).
- The weights and biases for the convolution operation are declared as registers.
- Internal signals for input buffer, convolution result, and ReLU activation result are declared.
- The convolution operation is performed in the always block triggered by the positive edge of the clock or negative edge of the reset signal.
- If the reset signal is low, all internal signals and outputs are reset to zero.
- If the data_valid signal is high, the input data is shifted into the input buffer.

- The convolution operation is performed by iterating over the input dimensions, filters, and kernel dimensions. The convolution result is accumulated in `conv_result`.
- The ReLU activation is applied to the convolution result, storing the output in `relu_result`.
- The final output is assigned to `data_out`, and the `data_out_valid` signal is set to 1.

Note that this code assumes a specific input and output data format and uses fixed-point arithmetic. You may need to modify the code based on your specific requirements, such as data precision, quantization, and any optimizations for resource utilization.

Also, keep in mind that the provided code is a starting point and may require further testing, debugging, and optimization to ensure correct functionality and meet the desired performance targets.

Copy

Claude does not have the ability to run the code it generates yet.

EB

Based on the discussion we have had thus far. Could you assist me in generating a prompt for the Max Pooling module, utilizing synthesizable Verilog?

Certainly! Here's a prompt for the max pooling module (`maxpool2d`) using synthesizable Verilog:

verilog

Copy code

```
module maxpool2d #(
    parameter INPUT_WIDTH = 16,
    parameter INPUT_HEIGHT = 1,
    parameter INPUT_CHANNELS = 32,
    parameter KERNEL_WIDTH = 2,
    parameter KERNEL_HEIGHT = 1,
    parameter STRIDE = 2
) (
    input wire clk,
    input wire rst_n,
    input wire [INPUT_WIDTH*INPUT_HEIGHT*INPUT_CHANNELS-1:0] data_in,
    input wire data_valid,
    output reg [(INPUT_WIDTH/STRIDE)*(INPUT_HEIGHT/STRIDE)*INPUT_CHANNELS-1:0] data_out,
    output reg data_out_valid
);

// Declare internal signals
reg [INPUT_WIDTH-1:0] input_buffer [0:INPUT_HEIGHT-1][0:INPUT_WIDTH-1];
reg [INPUT_WIDTH-1:0] max_value;

// Max pooling operation
integer i, j, k, m, n;
always @(posedge clk or negedge rst_n) begin
```

```

if (!rst_n) begin
    // Reset internal signals and output
    for (i = 0; i < INPUT_HEIGHT; i = i + 1) begin
        for (j = 0; j < INPUT_WIDTH; j = j + 1) begin
            input_buffer[i][j] <= 0;
        end
    end
    data_out <= 0;
    data_out_valid <= 0;
end else begin
    // Shift input data into buffer
    if (data_valid) begin
        for (i = 0; i < INPUT_HEIGHT; i = i + 1) begin
            for (j = 0; j < INPUT_WIDTH - 1; j = j + 1) begin
                input_buffer[i][j] <= input_buffer[i][j + 1];
            end
            input_buffer[i][INPUT_WIDTH - 1] <= data_in[i*INPUT_WIDTH
+: INPUT_WIDTH];
        end
    end

    // Perform max pooling
    for (i = 0; i < INPUT_HEIGHT; i = i + STRIDE) begin
        for (j = 0; j < INPUT_WIDTH; j = j + STRIDE) begin
            for (k = 0; k < INPUT_CHANNELS; k = k + 1) begin
                max_value <= input_buffer[i][j];
                for (m = 0; m < KERNEL_HEIGHT; m = m + 1) begin
                    for (n = 0; n < KERNEL_WIDTH; n = n + 1) begin
                        if (i + m < INPUT_HEIGHT && j + n <
INPUT_WIDTH) begin
                            max_value <= (input_buffer[i + m][j + n] >
max_value) ? input_buffer[i + m][j + n] : max_value;
                        end
                    end
                end
                data_out[((i/STRIDE)*(INPUT_WIDTH/STRIDE) +
(j/STRIDE))*INPUT_CHANNELS + k] <= max_value;
            end
        end
        data_out_valid <= 1;
    end
end
endmodule

```

This Verilog code implements a max pooling layer with the specified parameters. Here's a breakdown of the module:

- The module has parameters for input dimensions (INPUT_WIDTH, INPUT_HEIGHT, INPUT_CHANNELS), kernel dimensions (KERNEL_WIDTH, KERNEL_HEIGHT), and stride (STRIDE).

- The module takes a clock signal (`clk`), a reset signal (`rst_n`), input data (`data_in`), and a valid signal (`data_valid`) as inputs. It produces the output data (`data_out`) and a valid signal (`data_out_valid`).
- Internal signals for the input buffer and max value are declared.
- The max pooling operation is performed in the always block triggered by the positive edge of the clock or negative edge of the reset signal.
- If the reset signal is low, all internal signals and outputs are reset to zero.
- If the `data_valid` signal is high, the input data is shifted into the input buffer.
- The max pooling operation is performed by iterating over the input dimensions, channels, and kernel dimensions. The maximum value within each kernel window is computed and stored in `max_value`.
- The final output is assigned to `data_out`, and the `data_out_valid` signal is set to 1.

Note that this code assumes a specific input and output data format and uses fixed-point arithmetic. You may need to modify the code based on your specific requirements, such as data precision and any optimizations for resource utilization.

Also, keep in mind that the provided code is a starting point and may require further testing, debugging, and optimization to ensure correct functionality and meet the desired performance targets.

Copy

Claude does not have the ability to run the code it generates yet.

EB

Based on the discussion we have had thus far. Could you assist me in generating a prompt for the Fully Connected module "module fully_connected #(.INPUT_SIZE(512), .OUTPUT_SIZE(128), .ACTIVATION("relu")) (.clk(clk), .rst_n(rst_n), .data_in(pool2_out), .data_valid(pool2_valid), .data_out(fc1_out), .data_out_valid(fc1_valid)", utilizing synthesizable Verilog?

Certainly! Here's a prompt for the fully connected module (`fully_connected`) using synthesizable Verilog:

verilog

Copy code

```
module fully_connected #(
    parameter INPUT_SIZE = 512,
    parameter OUTPUT_SIZE = 128,
    parameter ACTIVATION = "relu"
) (
    input wire clk,
    input wire rst_n,
    input wire [INPUT_SIZE-1:0] data_in,
    input wire data_valid,
    output reg [OUTPUT_SIZE-1:0] data_out,
    output reg data_out_valid
);
```

```
// Declare weights and biases
```

```

reg [7:0] weights [0:OUTPUT_SIZE-1][0:INPUT_SIZE-1];
reg [7:0] biases [0:OUTPUT_SIZE-1];

// Declare internal signals
reg [15:0] mult_result [0:OUTPUT_SIZE-1];
reg [15:0] acc_result [0:OUTPUT_SIZE-1];
reg [15:0] relu_result [0:OUTPUT_SIZE-1];

// Fully connected layer operation
integer i, j;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        // Reset internal signals and output
        for (i = 0; i < OUTPUT_SIZE; i = i + 1) begin
            mult_result[i] <= 0;
            acc_result[i] <= 0;
            relu_result[i] <= 0;
        end
        data_out <= 0;
        data_out_valid <= 0;
    end else begin
        // Perform matrix multiplication
        for (i = 0; i < OUTPUT_SIZE; i = i + 1) begin
            mult_result[i] <= 0;
            for (j = 0; j < INPUT_SIZE; j = j + 1) begin
                mult_result[i] <= mult_result[i] + weights[i][j] *
data_in[j];
            end
            acc_result[i] <= mult_result[i] + biases[i];
        end

        // Apply activation function (ReLU)
        for (i = 0; i < OUTPUT_SIZE; i = i + 1) begin
            relu_result[i] <= (acc_result[i] > 0) ? acc_result[i] : 0;
        end

        // Assign output
        data_out <= relu_result;
        data_out_valid <= data_valid;
    end
end
endmodule

```

This Verilog code implements a fully connected layer with the specified parameters. Here's a breakdown of the module:

- The module has parameters for input size (`INPUT_SIZE`), output size (`OUTPUT_SIZE`), and activation function (`ACTIVATION`).
- The module takes a clock signal (`clk`), a reset signal (`rst_n`), input data (`data_in`), and a valid signal (`data_valid`) as inputs. It produces the output data (`data_out`) and a valid signal (`data_out_valid`).

- The weights and biases for the fully connected layer are declared as registers.
- Internal signals for multiplication result, accumulation result, and ReLU activation result are declared.
- The fully connected layer operation is performed in the always block triggered by the positive edge of the clock or negative edge of the reset signal.
- If the reset signal is low, all internal signals and outputs are reset to zero.
- The matrix multiplication is performed by iterating over the output and input dimensions. The multiplication result is accumulated in `mult_result`, and the bias is added to obtain the accumulation result `acc_result`.
- The ReLU activation function is applied to the accumulation result, storing the output in `relu_result`.
- The final output is assigned to `data_out`, and the `data_out_valid` signal is set to the value of `data_valid`.

Note that this code assumes a specific input and output data format and uses fixed-point arithmetic. You may need to modify the code based on your specific requirements, such as data precision, quantization, and any optimizations for resource utilization.

Also, keep in mind that the provided code is a starting point and may require further testing, debugging, and optimization to ensure correct functionality and meet the desired performance targets.

Copy

Claude does not have the ability to run the code it generates yet.

EB

Based on the discussion we have had thus far. Could you assist me in generating a prompt for the Softmax activation module "module softmax #(.INPUT_SIZE(128), .OUTPUT_SIZE(2)) (.clk(clk), .rst_n(rst_n), .data_in(fc1_out), .data_valid(fc1_valid), .data_out(softmax_out), .data_out_valid(softmax_valid));", utilizing synthesizable Verilog?

Certainly! Here's a prompt for the softmax activation module (`softmax`) using synthesizable Verilog:

verilog

Copy code

```
module softmax #(
    parameter INPUT_SIZE = 128,
    parameter OUTPUT_SIZE = 2
) (
    input wire clk,
    input wire rst_n,
    input wire [INPUT_SIZE-1:0] data_in,
    input wire data_valid,
    output reg [OUTPUT_SIZE-1:0] data_out,
    output reg data_out_valid
);

// Declare internal signals
reg [15:0] exp_values [0:INPUT_SIZE-1];
```

```

reg [31:0] sum_exp;
reg [15:0] softmax_values [0:INPUT_SIZE-1];

// Softmax activation function
integer i;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        // Reset internal signals and output
        for (i = 0; i < INPUT_SIZE; i = i + 1) begin
            exp_values[i] <= 0;
            softmax_values[i] <= 0;
        end
        sum_exp <= 0;
        data_out <= 0;
        data_out_valid <= 0;
    end else begin
        // Compute exponential values
        for (i = 0; i < INPUT_SIZE; i = i + 1) begin
            exp_values[i] <= $exp(data_in[i]);
        end

        // Compute sum of exponential values
        sum_exp <= 0;
        for (i = 0; i < INPUT_SIZE; i = i + 1) begin
            sum_exp <= sum_exp + exp_values[i];
        end

        // Compute softmax values
        for (i = 0; i < INPUT_SIZE; i = i + 1) begin
            softmax_values[i] <= (exp_values[i] << 16) / sum_exp;
        end

        // Assign output
        data_out <= softmax_values[OUTPUT_SIZE-1:0];
        data_out_valid <= data_valid;
    end
end
endmodule

```

This Verilog code implements the softmax activation function with the specified parameters. Here's a breakdown of the module:

- The module has parameters for input size (INPUT_SIZE) and output size (OUTPUT_SIZE).
- The module takes a clock signal (clk), a reset signal (rst_n), input data (data_in), and a valid signal (data_valid) as inputs. It produces the output data (data_out) and a valid signal (data_out_valid).
- Internal signals for exponential values, sum of exponential values, and softmax values are declared.
- The softmax activation function is computed in the always block triggered by the positive edge of the clock or negative edge of the reset signal.

- If the reset signal is low, all internal signals and outputs are reset to zero.
- The exponential values are computed using the `$exp` function for each input value.
- The sum of exponential values is computed by iterating over the exponential values.
- The softmax values are computed by dividing each exponential value by the sum of exponential values. The division is performed using fixed-point arithmetic by shifting the exponential values by 16 bits before division.
- The final output is assigned to `data_out` by selecting the first `OUTPUT_SIZE` softmax values, and the `data_out_valid` signal is set to the value of `data_valid`.

Note that this code assumes a specific input and output data format and uses fixed-point arithmetic. The exponential and division operations are approximated using basic Verilog constructs. For more accurate results, you may need to implement more sophisticated approximation techniques or use floating-point arithmetic, depending on your resource constraints and accuracy requirements.

Also, keep in mind that the provided code is a starting point and may require further testing, debugging, and optimization to ensure correct functionality and meet the desired performance targets.

Copy

Claude does not have the ability to run the code it generates yet.

EB

In this line `"exp_values[i] <= $exp(data_in[i]);"` the `exp` is not synthesizable. How can we fix this problem?

You're correct. The `$exp` function is not synthesizable in Verilog because it is a non-linear function that cannot be directly mapped to hardware. To fix this problem, you can approximate the exponential function using a piece-wise linear approximation or a lookup table (LUT) based approach. Here's an example of how you can modify the code using a LUT-based approximation:

verilog

Copy code

```
module softmax #(
    parameter INPUT_SIZE = 128,
    parameter OUTPUT_SIZE = 2,
    parameter LUT_SIZE = 256,
    parameter LUT_WIDTH = 16
) (
    input wire clk,
    input wire rst_n,
    input wire [INPUT_SIZE-1:0] data_in,
    input wire data_valid,
    output reg [OUTPUT_SIZE-1:0] data_out,
    output reg data_out_valid
);

// Declare internal signals
reg [LUT_WIDTH-1:0] exp_lut [0:LUT_SIZE-1];
```

```

reg [LUT_WIDTH-1:0] exp_values [0:INPUT_SIZE-1];
reg [31:0] sum_exp;
reg [LUT_WIDTH-1:0] softmax_values [0:INPUT_SIZE-1];

// Initialize the exponential LUT
integer i;
initial begin
    for (i = 0; i < LUT_SIZE; i = i + 1) begin
        exp_lut[i] = $rtoi($exp(i / 256.0) * (1 << LUT_WIDTH));
    end
end

// Softmax activation function
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        // Reset internal signals and output
        for (i = 0; i < INPUT_SIZE; i = i + 1) begin
            exp_values[i] <= 0;
            softmax_values[i] <= 0;
        end
        sum_exp <= 0;
        data_out <= 0;
        data_out_valid <= 0;
    end else begin
        // Compute exponential values using LUT
        for (i = 0; i < INPUT_SIZE; i = i + 1) begin
            exp_values[i] <= exp_lut[data_in[i]];
        end

        // Compute sum of exponential values
        sum_exp <= 0;
        for (i = 0; i < INPUT_SIZE; i = i + 1) begin
            sum_exp <= sum_exp + exp_values[i];
        end

        // Compute softmax values
        for (i = 0; i < INPUT_SIZE; i = i + 1) begin
            softmax_values[i] <= (exp_values[i] << 16) / sum_exp;
        end

        // Assign output
        data_out <= softmax_values[OUTPUT_SIZE-1:0];
        data_out_valid <= data_valid;
    end
end

endmodule

```

In this modified code:

1. We introduce two new parameters: `LUT_SIZE` and `LUT_WIDTH`, which define the size and width of the exponential LUT.

2. We declare a new internal signal `exp_lut` to store the precomputed exponential values.
3. In the `initial` block, we precompute the exponential values for the LUT using the `$exp` function and store them in `exp_lut`. The input range `[0, LUT_SIZE-1]` is mapped to the range `[0, 1]` by dividing by `LUT_SIZE`. The exponential values are then scaled and rounded to fixed-point representation with `LUT_WIDTH` bits.
4. In the `always` block, instead of using `$exp`, we access the precomputed exponential values from `exp_lut` using the input data as an index.

By using a LUT-based approximation, we can avoid the non-synthesizable `$exp` function and approximate the exponential values using precomputed values. The accuracy of the approximation depends on the size and width of the LUT. You can adjust `LUT_SIZE` and `LUT_WIDTH` based on your accuracy requirements and available hardware resources.

Please note that this is just one possible approach to approximate the exponential function. There are other techniques, such as piece-wise linear approximation or polynomial approximation, that you can explore depending on your specific requirements and constraints.

Copy

Claude does not have the ability to run the code it generates yet.

EB

Is this technique space efficient?

The LUT-based approximation technique for the exponential function can be space-efficient depending on the size of the LUT and the input range. However, it does come with a trade-off between accuracy and resource utilization.

In the provided code, the space efficiency depends on the values of `LUT_SIZE` and `LUT_WIDTH`:

- `LUT_SIZE` determines the number of entries in the LUT. A larger `LUT_SIZE` allows for a finer-grained approximation of the exponential function but requires more memory to store the precomputed values.
- `LUT_WIDTH` determines the width (in bits) of each entry in the LUT. A larger `LUT_WIDTH` provides higher precision for the precomputed exponential values but also increases the memory footprint.

The memory requirement for the LUT can be calculated as follows:

Copy code

```
LUT memory = LUT_SIZE * LUT_WIDTH bits
```

For example, if `LUT_SIZE` is 256 and `LUT_WIDTH` is 16, the LUT would require:

Copy code

```
LUT memory = 256 * 16 bits = 4096 bits = 512 bytes
```

While the LUT-based approach can be space-efficient compared to other approximation techniques or storing the full precision exponential values, it still introduces additional memory overhead compared to not using an approximation at all.

The space efficiency also depends on the specific requirements of your application. If you have a large number of softmax units or if the LUTs are replicated across multiple instances, the memory overhead can accumulate.

To optimize space efficiency, you can consider the following:

1. Carefully select the `LUT_SIZE` and `LUT_WIDTH` values based on your accuracy requirements. Use the smallest LUT size and width that provide acceptable accuracy for your application.
2. Explore alternative approximation techniques that may require less memory, such as piece-wise linear approximation or polynomial approximation.
3. Consider quantization techniques to reduce the bit width of the input data and the LUT entries, which can help reduce the overall memory footprint.
4. If the softmax function is used in multiple instances or layers, consider sharing the LUT across those instances to avoid redundant memory usage.

Ultimately, the choice of approximation technique and its space efficiency should be evaluated in the context of your specific hardware constraints, accuracy requirements, and the overall system design.

Copy

Claude does not have the ability to run the code it generates yet.

EB

Help me validate the softmax module by utilizing a testbench that includes assertions and test cases.

Certainly! Here's an example testbench that includes assertions and test cases to validate the softmax module:

verilog

Copy code

```
`timescale 1ns/1ps

module softmax_tb;

    // Parameters
    parameter INPUT_SIZE = 128;
    parameter OUTPUT_SIZE = 2;
    parameter LUT_SIZE = 256;
    parameter LUT_WIDTH = 16;

    // Inputs
    reg clk;
    reg rst_n;
    reg [INPUT_SIZE-1:0] data_in;
```

```

reg data_valid;

// Outputs
wire [OUTPUT_SIZE-1:0] data_out;
wire data_out_valid;

// Instantiate the softmax module
softmax #(
    .INPUT_SIZE(INPUT_SIZE),
    .OUTPUT_SIZE(OUTPUT_SIZE),
    .LUT_SIZE(LUT_SIZE),
    .LUT_WIDTH(LUT_WIDTH)
) dut (
    .clk(clk),
    .rst_n(rst_n),
    .data_in(data_in),
    .data_valid(data_valid),
    .data_out(data_out),
    .data_out_valid(data_out_valid)
);

// Clock generation
always #5 clk = ~clk;

// Stimulus generation
initial begin
    // Initialize inputs
    clk = 0;
    rst_n = 0;
    data_in = 0;
    data_valid = 0;

    // Reset assertion
    #10 rst_n = 1;
    @(posedge clk);

    // Test case 1: Valid input
    data_in = {16'd10, 16'd20, 16'd30, 16'd40};
    data_valid = 1;
    @(posedge clk);
    data_valid = 0;
    #10;
    assert(data_out_valid == 1) else $error("Test case 1 failed:
data_out_valid should be 1");

    // Test case 2: Invalid input
    data_in = {16'd0, 16'd0, 16'd0, 16'd0};
    data_valid = 1;
    @(posedge clk);
    data_valid = 0;
    #10;
    assert(data_out_valid == 1) else $error("Test case 2 failed:
data_out_valid should be 1");

```

```

// Test case 3: Consecutive inputs
data_in = {16'd5, 16'd10, 16'd15, 16'd20};
data_valid = 1;
@(posedge clk);
data_in = {16'd25, 16'd30, 16'd35, 16'd40};
@(posedge clk);
data_valid = 0;
#10;
assert(data_out_valid == 1) else $error("Test case 3 failed:
data_out_valid should be 1");

// Add more test cases as needed

#100;
$display("Testbench completed");
$finish;
end

// Assertions
always @(posedge clk) begin
    if (data_out_valid) begin
        assert(data_out >= 0 && data_out < OUTPUT_SIZE) else $error("Output
out of range");
    end
end

endmodule

```

In this testbench:

1. We define the necessary parameters and signals to interface with the softmax module.
2. We instantiate the softmax module using the dut (device under test) instance.
3. We generate a clock signal using an always block.
4. In the initial block, we define the stimulus generation:
 - We initialize the inputs.
 - We assert the reset signal and wait for a positive clock edge.
 - We define multiple test cases:
 - Test case 1: We provide a valid input and check if data_out_valid is asserted correctly.
 - Test case 2: We provide an invalid input (all zeros) and check if data_out_valid is still asserted.
 - Test case 3: We provide consecutive inputs and check if data_out_valid is asserted correctly.
 - We can add more test cases as needed to cover different scenarios.
5. We use assert statements within the test cases to check if the expected behavior is met. If an assertion fails, an error message is displayed using \$error.
6. We have an always block with assertions that continuously monitor the output data_out whenever data_out_valid is asserted. It checks if the output value is within the valid range.

To run the testbench, you can simulate the design using a Verilog simulator. The testbench will generate the specified inputs, and the assertions will check if the softmax module behaves as expected. If any assertion fails, an error message will be displayed, indicating which test case failed and the reason for the failure.

You can extend the testbench by adding more test cases, covering different input scenarios, and incorporating additional assertions to validate the correctness of the softmax module comprehensively.

Copy

Claude does not have the ability to run the code it generates yet.

EB

Help me validate the maxpool2d module by utilizing a testbench that includes assertions and test cases.

Certainly! Here's an example testbench that includes assertions and test cases to validate the maxpool2d module:

verilog

Copy code

```
`timescale 1ns/1ps

module maxpool2d_tb;

    // Parameters
    parameter INPUT_WIDTH = 4;
    parameter INPUT_HEIGHT = 4;
    parameter INPUT_CHANNELS = 2;
    parameter KERNEL_WIDTH = 2;
    parameter KERNEL_HEIGHT = 2;
    parameter STRIDE = 2;

    // Inputs
    reg clk;
    reg rst_n;
    reg [INPUT_WIDTH*INPUT_HEIGHT*INPUT_CHANNELS-1:0] data_in;
    reg data_valid;

    // Outputs
    wire [(INPUT_WIDTH/STRIDE)*(INPUT_HEIGHT/STRIDE)*INPUT_CHANNELS-1:0]
data_out;
    wire data_out_valid;

    // Instantiate the maxpool2d module
    maxpool2d #(
        .INPUT_WIDTH(INPUT_WIDTH),
        .INPUT_HEIGHT(INPUT_HEIGHT),
        .INPUT_CHANNELS(INPUT_CHANNELS),
        .KERNEL_WIDTH(KERNEL_WIDTH),
        .KERNEL_HEIGHT(KERNEL_HEIGHT),
```

```

        .STRIDE(STRIDE)
    ) dut (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(data_in),
        .data_valid(data_valid),
        .data_out(data_out),
        .data_out_valid(data_out_valid)
    );

    // Clock generation
    always #5 clk = ~clk;

    // Stimulus generation
    initial begin
        // Initialize inputs
        clk = 0;
        rst_n = 0;
        data_in = 0;
        data_valid = 0;

        // Reset assertion
        #10 rst_n = 1;
        @(posedge clk);

        // Test case 1: Valid input
        data_in = {8'd1, 8'd2, 8'd3, 8'd4,
                   8'd5, 8'd6, 8'd7, 8'd8,
                   8'd9, 8'd10, 8'd11, 8'd12,
                   8'd13, 8'd14, 8'd15, 8'd16,
                   8'd17, 8'd18, 8'd19, 8'd20,
                   8'd21, 8'd22, 8'd23, 8'd24,
                   8'd25, 8'd26, 8'd27, 8'd28,
                   8'd29, 8'd30, 8'd31, 8'd32};
        data_valid = 1;
        @(posedge clk);
        data_valid = 0;
        #10;
        assert(data_out_valid == 1) else $error("Test case 1 failed:
data_out_valid should be 1");
        assert(data_out == {8'd6, 8'd8, 8'd22, 8'd24}) else $error("Test case 1
failed: data_out mismatch");

        // Test case 2: Invalid input
        data_in = 0;
        data_valid = 1;
        @(posedge clk);
        data_valid = 0;
        #10;
        assert(data_out_valid == 1) else $error("Test case 2 failed:
data_out_valid should be 1");
        assert(data_out == 0) else $error("Test case 2 failed: data_out
mismatch");
    end

```

```

// Add more test cases as needed

#100;
$display("Testbench completed");
$finish;
end

// Assertions
always @(posedge clk) begin
    if (data_out_valid) begin
        assert(data_out >= 0 && data_out < (1 << 8)) else $error("Output out
of range");
    end
end

endmodule

```

In this testbench:

1. We define the necessary parameters and signals to interface with the `maxpool2d` module.
2. We instantiate the `maxpool2d` module using the `dut` (device under test) instance.
3. We generate a clock signal using an `always` block.
4. In the `initial` block, we define the stimulus generation:
 - We initialize the inputs.
 - We assert the reset signal and wait for a positive clock edge.
 - We define multiple test cases:
 - Test case 1: We provide a valid input with a 4x4 feature map and 2 channels. We check if `data_out_valid` is asserted correctly and if the output `data_out` matches the expected max-pooled values.
 - Test case 2: We provide an invalid input (all zeros) and check if `data_out_valid` is still asserted and if the output `data_out` is zero.
 - We can add more test cases as needed to cover different scenarios.
5. We use `assert` statements within the test cases to check if the expected behavior is met. If an assertion fails, an error message is displayed using `$error`.
6. We have an `always` block with assertions that continuously monitor the output `data_out` whenever `data_out_valid` is asserted. It checks if the output values are within the valid range.

To run the testbench, you can simulate the design using a Verilog simulator. The testbench will generate the specified inputs, and the assertions will check if the `maxpool2d` module behaves as expected. If any assertion fails, an error message will be displayed, indicating which test case failed and the reason for the failure.

You can extend the testbench by adding more test cases, covering different input scenarios, and incorporating additional assertions to validate the correctness of the `maxpool2d` module comprehensively.

Copy

Claude does not have the ability to run the code it generates yet.

EB

Help me validate the `fully_connected` module by utilizing a testbench that includes assertions and test cases.

Certainly! Here's an example testbench that includes assertions and test cases to validate the `fully_connected` module:

verilog

Copy code

```
`timescale 1ns/1ps

module fully_connected_tb;

    // Parameters
    parameter INPUT_SIZE = 512;
    parameter OUTPUT_SIZE = 128;
    parameter ACTIVATION = "relu";

    // Inputs
    reg clk;
    reg rst_n;
    reg [INPUT_SIZE-1:0] data_in;
    reg data_valid;

    // Outputs
    wire [OUTPUT_SIZE-1:0] data_out;
    wire data_out_valid;

    // Instantiate the fully_connected module
    fully_connected #(
        .INPUT_SIZE(INPUT_SIZE),
        .OUTPUT_SIZE(OUTPUT_SIZE),
        .ACTIVATION(ACTIVATION)
    ) dut (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(data_in),
        .data_valid(data_valid),
        .data_out(data_out),
        .data_out_valid(data_out_valid)
    );

    // Clock generation
    always #5 clk = ~clk;

    // Stimulus generation
    initial begin
        // Initialize inputs
        clk = 0;
```



```

rst_n = 0;
data_in = 0;
data_valid = 0;

// Reset assertion
#10 rst_n = 1;
@(posedge clk);

// Test case 1: Valid input
data_in = {16'd1, 16'd2, 16'd3, 16'd4, 16'd5, 16'd6, 16'd7, 16'd8,
           16'd9, 16'd10, 16'd11, 16'd12, 16'd13, 16'd14, 16'd15,
16'd16,
           16'd17, 16'd18, 16'd19, 16'd20, 16'd21, 16'd22, 16'd23,
16'd24,
           16'd25, 16'd26, 16'd27, 16'd28, 16'd29, 16'd30, 16'd31,
16'd32};
data_valid = 1;
@(posedge clk);
data_valid = 0;
#10;
assert(data_out_valid == 1) else $error("Test case 1 failed:
data_out_valid should be 1");
// Add assertions for expected output values

// Test case 2: Invalid input
data_in = 0;
data_valid = 1;
@(posedge clk);
data_valid = 0;
#10;
assert(data_out_valid == 1) else $error("Test case 2 failed:
data_out_valid should be 1");
// Add assertions for expected output values

// Test case 3: Consecutive inputs
data_in = {16'd10, 16'd20, 16'd30, 16'd40, 16'd50, 16'd60, 16'd70,
16'd80,
           16'd90, 16'd100, 16'd110, 16'd120, 16'd130, 16'd140,
16'd150, 16'd160,
           16'd170, 16'd180, 16'd190, 16'd200, 16'd210, 16'd220,
16'd230, 16'd240,
           16'd250, 16'd260, 16'd270, 16'd280, 16'd290, 16'd300,
16'd310, 16'd320};
data_valid = 1;
@(posedge clk);
data_in = {16'd330, 16'd340, 16'd350, 16'd360, 16'd370, 16'd380,
16'd390, 16'd400,
           16'd410, 16'd420, 16'd430, 16'd440, 16'd450, 16'd460,
16'd470, 16'd480,
           16'd490, 16'd500, 16'd510, 16'd520, 16'd530, 16'd540,
16'd550, 16'd560,
           16'd570, 16'd580, 16'd590, 16'd600, 16'd610, 16'd620,
16'd630, 16'd640};

```

```

    @(posedge clk);
    data_valid = 0;
    #10;
    assert(data_out_valid == 1) else $error("Test case 3 failed:
data_out_valid should be 1");
    // Add assertions for expected output values

    // Add more test cases as needed

    #100;
    $display("Testbench completed");
    $finish;
end

// Assertions
always @(posedge clk) begin
    if (data_out_valid) begin
        assert(data_out >= 0) else $error("Output should be non-negative");
        // Add more assertions as needed
    end
end

endmodule

```

In this testbench:

1. We define the necessary parameters and signals to interface with the `fully_connected` module.
2. We instantiate the `fully_connected` module using the `dut` (device under test) instance.
3. We generate a clock signal using an `always` block.
4. In the `initial` block, we define the stimulus generation:
 - We initialize the inputs.
 - We assert the reset signal and wait for a positive clock edge.
 - We define multiple test cases:
 - Test case 1: We provide a valid input vector and check if `data_out_valid` is asserted correctly. You should add assertions for the expected output values based on your specific implementation.
 - Test case 2: We provide an invalid input (all zeros) and check if `data_out_valid` is still asserted. Add assertions for the expected output values.
 - Test case 3: We provide consecutive input vectors to test the module's behavior with continuous data. Add assertions for the expected output values.
 - You can add more test cases as needed to cover different scenarios.
5. We use `assert` statements within the test cases to check if the expected behavior is met. If an assertion fails, an error message is displayed using `$error`. You should add specific assertions for the expected output values based on your implementation.
6. We have an `always` block with assertions that continuously monitor the output `data_out` whenever `data_out_valid` is asserted. In this example, it checks if the

output values are non-negative. You can add more assertions as needed to validate the output.

To run the testbench, you can simulate the design using a Verilog simulator. The testbench will generate the specified inputs, and the assertions will check if the `fully_connected` module behaves as expected. If any assertion fails, an error message will be displayed, indicating which test case failed and the reason for the failure.

Remember to add specific assertions for the expected output values based on your implementation and the weights and biases used in the `fully_connected` module.

You can extend the testbench by adding more test cases, covering different input scenarios, and incorporating additional assertions to validate the correctness of the `fully_connected` module comprehensively.

Copy

Claude does not have the ability to run the code it generates yet.

EB

Help me validate the `cnn_kws_accel` module by utilizing a testbench that includes assertions and test cases.

Certainly! Here's an example testbench that includes assertions and test cases to validate the `cnn_kws_accel` module:

verilog

Copy code

```
`timescale 1ns/1ps

module cnn_kws_accel_tb;

    // Parameters
    parameter INPUT_WIDTH = 32;
    parameter OUTPUT_SIZE = 2;

    // Inputs
    reg clk;
    reg rst_n;
    reg [INPUT_WIDTH-1:0] mfcc_in;
    reg mfcc_valid;

    // Outputs
    wire [OUTPUT_SIZE-1:0] keyword_class;
    wire keyword_detected;

    // Instantiate the cnn_kws_accel module
    cnn_kws_accel #(
        .INPUT_WIDTH(INPUT_WIDTH),
        .OUTPUT_SIZE(OUTPUT_SIZE)
    ) dut (
```

```

        .clk(clk),
        .rst_n(rst_n),
        .mfcc_in(mfcc_in),
        .mfcc_valid(mfcc_valid),
        .keyword_class(keyword_class),
        .keyword_detected(keyword_detected)
    );

    // Clock generation
    always #5 clk = ~clk;

    // Stimulus generation
    initial begin
        // Initialize inputs
        clk = 0;
        rst_n = 0;
        mfcc_in = 0;
        mfcc_valid = 0;

        // Reset assertion
        #10 rst_n = 1;
        @(posedge clk);

        // Test case 1: Valid input sequence for keyword 1
        mfcc_in = 32'h1234_5678;
        mfcc_valid = 1;
        @(posedge clk);
        mfcc_in = 32'h2345_6789;
        @(posedge clk);
        mfcc_in = 32'h3456_789A;
        @(posedge clk);
        mfcc_in = 32'h4567_89AB;
        @(posedge clk);
        mfcc_valid = 0;
        @(posedge clk);
        #10;
        assert(keyword_detected == 1) else $error("Test case 1 failed:
keyword_detected should be 1");
        assert(keyword_class == 2'b01) else $error("Test case 1 failed:
keyword_class mismatch");

        // Test case 2: Valid input sequence for keyword 2
        mfcc_in = 32'hABCD_EF01;
        mfcc_valid = 1;
        @(posedge clk);
        mfcc_in = 32'hBCDE_F012;
        @(posedge clk);
        mfcc_in = 32'hCDEF_0123;
        @(posedge clk);
        mfcc_in = 32'hDEF0_1234;
        @(posedge clk);
        mfcc_valid = 0;
        @(posedge clk);

```

```

#10;
assert(keyword_detected == 1) else $error("Test case 2 failed:
keyword_detected should be 1");
assert(keyword_class == 2'b10) else $error("Test case 2 failed:
keyword_class mismatch");

// Test case 3: Invalid input sequence
mfcc_in = 32'h1111_1111;
mfcc_valid = 1;
@(posedge clk);
mfcc_in = 32'h2222_2222;
@(posedge clk);
mfcc_in = 32'h3333_3333;
@(posedge clk);
mfcc_in = 32'h4444_4444;
@(posedge clk);
mfcc_valid = 0;
@(posedge clk);
#10;
assert(keyword_detected == 0) else $error("Test case 3 failed:
keyword_detected should be 0");

// Add more test cases as needed

#100;
$display("Testbench completed");
$finish;
end

// Assertions
always @(posedge clk) begin
    if (keyword_detected) begin
        assert(keyword_class >= 0 && keyword_class < OUTPUT_SIZE) else
$error("Keyword class out of range");
    end
end

endmodule

```

In this testbench:

1. We define the necessary parameters and signals to interface with the `cnn_kws_accel` module.
2. We instantiate the `cnn_kws_accel` module using the `dut` (device under test) instance.
3. We generate a clock signal using an `always` block.
4. In the `initial` block, we define the stimulus generation:
 - We initialize the inputs.
 - We assert the reset signal and wait for a positive clock edge.
 - We define multiple test cases:
 - Test case 1: We provide a valid input sequence for keyword 1 and check if `keyword_detected` is asserted and `keyword_class` matches the expected value.

- Test case 2: We provide a valid input sequence for keyword 2 and check if `keyword_detected` is asserted and `keyword_class` matches the expected value.
- Test case 3: We provide an invalid input sequence and check if `keyword_detected` remains low.
 - You can add more test cases as needed to cover different scenarios.
- 5. We use `assert` statements within the test cases to check if the expected behavior is met. If an assertion fails, an error message is displayed using `$error`.
- 6. We have an `always` block with an assertion that continuously monitors the output `keyword_class` whenever `keyword_detected` is asserted. It checks if the keyword class is within the valid range.

To run the testbench, you can simulate the design using a Verilog simulator. The testbench will generate the specified inputs, and the assertions will check if the `cnn_kws_accel` module behaves as expected. If any assertion fails, an error message will be displayed, indicating which test case failed and the reason for the failure.

Please note that the test cases and input sequences provided in this testbench are for illustrative purposes only. You should modify them based on your specific implementation and the expected behavior of your `cnn_kws_accel` module.

You can extend the testbench by adding more test cases, covering different input scenarios, and incorporating additional assertions to validate the correctness of the `cnn_kws_accel` module comprehensively.