

Chat : <https://claude.ai/chat/99e87a5b-8e5c-4122-8b29-a2ee1fe0815f>

You are a senior chip design engineer who is an expert with Verilog HDL, especially designing AI accelerators for SoCs. I want to 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. We chose the CNN model that has been widely used for KWS, 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). I want you to help me refactor the code for the project. Top module is kws. Submodule code ``ifndef FULLY_CONNECTED_V

```
`define FULLY_CONNECTED_V

module fully_connected #(

parameter INPUT_SIZE = 160,

parameter OUTPUT_SIZE = 64,

parameter ACTIV_BITS = 8

) (

input wire clk,

input wire rst_n,

input wire [INPUT_SIZE*ACTIV_BITS-1:0] data_in,

input wire data_valid,

output reg [OUTPUT_SIZE*ACTIV_BITS-1:0] data_out,

output reg data_out_valid,

input wire [OUTPUT_SIZE*INPUT_SIZE*ACTIV_BITS-1:0] weights_in,

input wire [OUTPUT_SIZE*ACTIV_BITS-1:0] biases_in,

input wire load_weights,

input wire load_biases

);

// Declare weights and biases
```

```

reg [ACTIV_BITS-1:0] weights [0:OUTPUT_SIZE-1][0:INPUT_SIZE-1];

reg [ACTIV_BITS-1:0] biases [0:OUTPUT_SIZE-1];

// Declare internal signals

reg [2*ACTIV_BITS-1:0] acc_result [0:OUTPUT_SIZE-1];

reg [ACTIV_BITS-1:0] relu_result [0:OUTPUT_SIZE-1];

// Load weights and biases

integer i_load, j_load;

always @(posedge clk or negedge rst_n) begin

if (!rst_n) begin

// Reset weights and biases

for (i_load = 0; i_load < OUTPUT_SIZE; i_load = i_load + 1) begin

for (j_load = 0; j_load < INPUT_SIZE; j_load = j_load + 1) begin

weights[i_load][j_load] <= 0;

end

biases[i_load] <= 0;

end

end else begin

// Load weights when load_weights is asserted

if (load_weights) begin

for (i_load = 0; i_load < OUTPUT_SIZE; i_load = i_load + 1) begin

for (j_load = 0; j_load < INPUT_SIZE; j_load = j_load + 1) begin

weights[i_load][j_load] <= weights_in[(i_load*INPUT_SIZE + j_load)*ACTIV_BITS +:
ACTIV_BITS];

end

end

end

end

```

```

// Load biases when load_biases is asserted

if (load_biases) begin

for (i_load = 0; i_load < OUTPUT_SIZE; i_load = i_load + 1) begin

biases[i_load] <= biases_in[i_load*ACTIV_BITS +: ACTIV_BITS];

end

end

end

end

// Fully connected layer operation

integer i_fc, j_fc;

always @(posedge clk or negedge rst_n) begin

if (!rst_n) begin

// Reset internal signals and output

for (i_fc = 0; i_fc < OUTPUT_SIZE; i_fc = i_fc + 1) begin

acc_result[i_fc] <= 0;

relu_result[i_fc] <= 0;

end

data_out <= 0;

data_out_valid <= 0;

end else if (data_valid) begin

// Perform matrix multiplication

for (i_fc = 0; i_fc < OUTPUT_SIZE; i_fc = i_fc + 1) begin

acc_result[i_fc] = { {(2*ACTIV_BITS-ACTIV_BITS){1'b0}}, biases[i_fc]};

for (j_fc = 0; j_fc < INPUT_SIZE; j_fc = j_fc + 1) begin

acc_result[i_fc] = acc_result[i_fc] + weights[i_fc][j_fc] _data_in[j_fc_ACTIV_BITS +:
ACTIV_BITS];

```

```

end

end

// Apply ReLU activation

for (i_fc = 0; i_fc < OUTPUT_SIZE; i_fc = i_fc + 1) begin

relu_result[i_fc] <= (acc_result[i_fc][2*ACTIV_BITS-1] == 0) ?
acc_result[i_fc][ACTIV_BITS-1:0] : 0;

end

// Assign output

for (i_fc = 0; i_fc < OUTPUT_SIZE; i_fc = i_fc + 1) begin

data_out[i_fc*ACTIV_BITS +: ACTIV_BITS] <= relu_result[i_fc];

end

data_out_valid <= 1;

end else begin

data_out_valid <= 0;

end

end

endmodule

`endif", "`ifndef GOERTZEL_DFT_V

`define GOERTZEL_DFT_V

module goertzel_dft #(

parameter NUM_FREQS = 256

)(

input wire clk,

input wire rst_n,

input wire [15:0] framed_out,

input wire framed_valid,

```

```

input wire [4095:0] goertzel_coefs,

output reg [31:0] dft_out,

output reg dft_valid

);

// Goertzel algorithm variables

reg [31:0] q_prev [0:NUM_FREQS-1];

reg [31:0] q_curr [0:NUM_FREQS-1];

reg [$clog2(NUM_FREQS)-1:0] freq_idx;

// Goertzel algorithm implementation

integer j;

always @(posedge clk or negedge rst_n) begin

if (!rst_n) begin

for (j = 0; j < NUM_FREQS; j = j + 1) begin

q_prev[j] <= 32'h0;

q_curr[j] <= 32'h0;

end

freq_idx <= 'h0;

dft_out <= 32'h0;

dft_valid <= 1'b0;

end else if (framed_valid) begin

for (j = 0; j < NUM_FREQS; j = j + 1) begin

// Create temporary variables

reg [31:0] q_curr_temp;

reg [31:0] q_prev_temp;

// Compute Goertzel algorithm

```

```

q_curr_temp = (goertzel_coefs[j*16 +: 16] * q_prev[j] >>> 15) - q_curr[j] +
{{16{framed_out[15]}}, framed_out};

q_prev_temp = q_curr[j];

// Assign the updated values to the arrays

q_curr[j] <= q_curr_temp;

q_prev[j] <= q_prev_temp;

end

// Increment frequency index

freq_idx <= freq_idx + 1;

// Output DFT result when all frequencies are processed

if (freq_idx == NUM_FREQS[$clog2(NUM_FREQS)-1:0] - 1) begin

dft_out <= q_curr[freq_idx]*q_curr[freq_idx] + q_prev[freq_idx]*q_prev[freq_idx] -
(goertzel_coefs[freq_idx*16 +: 16] * q_curr[freq_idx] >>> 15);

dft_valid <= 1'b1;

freq_idx <= 'h0;

end else begin

dft_valid <= 1'b0;

end

end else begin

dft_valid <= 1'b0;

end

end

endmodule

`endif", "`ifndef MAXPOOL2D_V

`define MAXPOOL2D_V

module maxpool2d #(

```

```

parameter INPUT_WIDTH = 40,

parameter INPUT_HEIGHT = 1,

parameter INPUT_CHANNELS = 8,

parameter KERNEL_SIZE = 2,

parameter STRIDE = 2,

parameter ACTIV_BITS = 8

) (

input wire clk,

input wire rst_n,

input wire [INPUT_WIDTH*INPUT_HEIGHT*INPUT_CHANNELS*ACTIV_BITS-1:0]
data_in,

input wire data_valid,

output reg [(INPUT_WIDTH/STRIDE)*INPUT_CHANNELS*ACTIV_BITS-1:0] data_out,

output reg data_out_valid

);

localparam OUTPUT_WIDTH = INPUT_WIDTH / STRIDE;

localparam OUTPUT_HEIGHT = INPUT_HEIGHT / STRIDE;

// Declare internal signals

reg [ACTIV_BITS-1:0] input_buffer [0:INPUT_HEIGHT-1][0:INPUT_WIDTH-
1][0:INPUT_CHANNELS-1];

reg [ACTIV_BITS-1:0] max_value [0:INPUT_CHANNELS-1];

// 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

for (k = 0; k < INPUT_CHANNELS; k = k + 1) begin

input_buffer[i][j][k] <= 0;

end

end

end

data_out <= 0;

data_out_valid <= 0;

end else if (data_valid) begin

// Shift input data into buffer

for (i = 0; i < INPUT_HEIGHT; i = i + 1) begin

for (j = 0; j < INPUT_WIDTH; j = j + 1) begin

for (k = 0; k < INPUT_CHANNELS; k = k + 1) begin

if (j < INPUT_WIDTH - 1) begin

input_buffer[i][j][k] <= input_buffer[i][j+1][k];

end else begin

input_buffer[i][j][k] <= data_in[i*INPUT_WIDTH*INPUT_CHANNELS*ACTIV_BITS +
j*INPUT_CHANNELS*ACTIV_BITS + k*ACTIV_BITS +: ACTIV_BITS];

end

end

end

end

end

// Perform max pooling

for (i = 0; i < OUTPUT_HEIGHT; i = i + 1) begin

for (j = 0; j < OUTPUT_WIDTH; j = j + 1) begin

```



```

for (k = 0; k < INPUT_CHANNELS; k = k + 1) begin

max_value[k] = input_buffer[i*STRIDE][j*STRIDE][k];

for (m = 0; m < KERNEL_SIZE; m = m + 1) begin

for (n = 0; n < KERNEL_SIZE; n = n + 1) begin

if (i*STRIDE + m < INPUT_HEIGHT && j*STRIDE + n < INPUT_WIDTH) begin

max_value[k] = (input_buffer[i*STRIDE + m][j*STRIDE + n][k] > max_value[k]) ?
input_buffer[i*STRIDE + m][j*STRIDE + n][k] : max_value[k];

end

end

end

data_out[i*OUTPUT_WIDTH*INPUT_CHANNELS*ACTIV_BITS +
j*INPUT_CHANNELS*ACTIV_BITS + k*ACTIV_BITS +: ACTIV_BITS] <=
max_value[k];

end

end

end

data_out_valid <= 1;

end else begin

data_out_valid <= 0;

end

end

endmodule

`endif", "`ifndef MEL_FILTERBANK_V

`define MEL_FILTERBANK_V

module mel_filterbank #(

parameter DFT_SIZE = 256,

parameter NUM_MEL_FILTERS = 40,

```

```

parameter MEL_FBANK_OUT_BITS = 32

) (

input wire clk,

input wire rst_n,

input wire [31:0] dft_out,

input wire dft_valid,

output reg [MEL_FBANK_OUT_BITS-1:0] mel_fbank_out,

output reg mel_fbank_valid

);

// Mel-scale filter coefficients (stored in ROM)

reg [15:0] mel_filter_coefs [0:NUM_MEL_FILTERS-1][0:DFT_SIZE-1];

// Mel-scale filter center frequencies (stored in ROM)

reg [7:0] mel_filter_centers [0:NUM_MEL_FILTERS];

// Intermediate variables

reg [31:0] filtered_dft [0:NUM_MEL_FILTERS-1];

reg [$clog2(NUM_MEL_FILTERS)-1:0] filter_idx;

reg [$clog2(DFT_SIZE)-1:0] dft_idx;

// Initialize mel-scale filter center frequencies

initial begin

mel_filter_centers[0] = 8'd0;

mel_filter_centers[1] = 8'd0;

mel_filter_centers[2] = 8'd1;

mel_filter_centers[3] = 8'd2;

mel_filter_centers[4] = 8'd3;

mel_filter_centers[5] = 8'd4;

```

```
mel_filter_centers[6] = 8'd5;  
mel_filter_centers[7] = 8'd6;  
mel_filter_centers[8] = 8'd7;  
mel_filter_centers[9] = 8'd8;  
mel_filter_centers[10] = 8'd9;  
mel_filter_centers[11] = 8'd10;  
mel_filter_centers[12] = 8'd12;  
mel_filter_centers[13] = 8'd13;  
mel_filter_centers[14] = 8'd15;  
mel_filter_centers[15] = 8'd17;  
mel_filter_centers[16] = 8'd18;  
mel_filter_centers[17] = 8'd20;  
mel_filter_centers[18] = 8'd22;  
mel_filter_centers[19] = 8'd24;  
mel_filter_centers[20] = 8'd27;  
mel_filter_centers[21] = 8'd29;  
mel_filter_centers[22] = 8'd32;  
mel_filter_centers[23] = 8'd34;  
mel_filter_centers[24] = 8'd37;  
mel_filter_centers[25] = 8'd41;  
mel_filter_centers[26] = 8'd44;  
mel_filter_centers[27] = 8'd47;  
mel_filter_centers[28] = 8'd51;  
mel_filter_centers[29] = 8'd55;  
mel_filter_centers[30] = 8'd59;
```


[illegible][illegible][illegible][illegible][illegible][illegible]


```

mel_fbank_valid <= 1'b0;

filter_idx <= 'h0;

dft_idx <= 'h0;

end else if (dft_valid) begin

// Apply mel-scale filters to DFT output

for (int i = 0; i < NUM_MEL_FILTERS; i = i + 1) begin

if (dft_idx >= mel_filter_centers[i] && dft_idx < mel_filter_centers[i+1]) begin

filtered_dft[i] <= filtered_dft[i] + (dft_out * mel_filter_coefs[i][dft_idx]);

end

end

// Increment DFT index

dft_idx <= dft_idx + 1;

// Check if all DFT bins have been processed

if (dft_idx == DFT_SIZE[$clog2(DFT_SIZE)-1:0] - 1) begin

// Output the accumulated mel-frequency filterbank output

mel_fbank_out <= filtered_dft[filter_idx];

mel_fbank_valid <= 1'b1;

// Increment filter index

filter_idx <= filter_idx + 1;

// Reset DFT index and filtered DFT accumulator

dft_idx <= 'h0;

filtered_dft[filter_idx] <= 32'h0;

// Check if all filters have been processed

if (filter_idx == NUM_MEL_FILTERS - 1) begin

filter_idx <= 'h0;

```

```

end

end else begin

mel_fbank_valid <= 1'b0;

end

end else begin

mel_fbank_valid <= 1'b0;

end

end

endmodule

`endif", "`ifndef MFCC_ACCELERATOR_V

`define MFCC_ACCELERATOR_V

module mfcc_accelerator #(

parameter MFCC_FEATURES = 40,

parameter ACTIV_BITS = 8

) (

input wire clk,

input wire rst_n,

input wire [15:0] audio_in,

input wire audio_valid,

output reg [MFCC_FEATURES*ACTIV_BITS-1:0] mfcc_out,

output reg mfcc_valid,

input wire [7:0] frame_size,

input wire [7:0] frame_overlap,

input wire [7:0] num_mfcc_coeffs,

input wire [4095:0] goertzel_coefs

```

```

);

// 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 [MFCC_FEATURES*ACTIV_BITS-1:0] dct_out;

wire dct_valid;

// Instantiate sub-modules

// 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),
    .goertzel_coefs(goertzel_coefs),
    .dft_out(dft_out),
    .dft_valid(dft_valid)
);

mel_filterbank #(
    .DFT_SIZE(256),
    .NUM_MEL_FILTERS(40),
    .MEL_FBANK_OUT_BITS(32)
) 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[4:0]),

.dct_out(dct_out),

.dct_valid(dct_valid)

```



```

);

// Output assignment

always @(posedge clk or negedge rst_n) begin

if (!rst_n) begin

mfcc_out <= 'b0;

mfcc_valid <= 1'b0;

end else begin

mfcc_out <= dct_out;

mfcc_valid <= dct_valid;

end

end

endmodule

`endif", "`ifndef PREEMPHASIS_FILTER_V

`define PREEMPHASIS_FILTER_V

module preemphasis_filter (

input wire clk,

input wire rst_n,

input wire [15:0] audio_in,

input wire audio_valid,

output reg [15:0] preemph_out,

output reg preemph_valid

);

// Configurable pre-emphasis coefficient (0.97 by default)

localparam PREEMPH_COEF = 16'h7D71; // 0.97 in Q15 format

// Internal registers

```

```

reg [15:0] audio_delay;

// Pre-emphasis filtering logic

always @(posedge clk or negedge rst_n) begin

if (!rst_n) begin

audio_delay <= 16'h0000;

preemph_out <= 16'h0000;

preemph_valid <= 1'b0;

end else if (audio_valid) begin

audio_delay <= audio_in;

preemph_out <= $signed(audio_in) - $signed(($signed(audio_delay) *
$signed(PREEMPH_COEF)) >>> 15);

preemph_valid <= 1'b1;

end else begin

preemph_valid <= 1'b0;

end

end

endmodule

`endif", "`ifndef SOFTMAX_V

`define SOFTMAX_V

module softmax #(

parameter INPUT_SIZE = 10,

parameter ACTIV_BITS = 8

) (

input wire clk,

input wire rst_n,

input wire [INPUT_SIZE*ACTIV_BITS-1:0] data_in,

```

```

input wire data_valid,

output reg [INPUT_SIZE*ACTIV_BITS-1:0] data_out,

output reg data_out_valid

);

// Declare internal signals

reg [ACTIV_BITS-1:0] exp_values [0:INPUT_SIZE-1];

reg [2*ACTIV_BITS-1:0] sum_exp;

reg [ACTIV_BITS-1:0] 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 if (data_valid) begin

// Compute exponential values

for (i = 0; i < INPUT_SIZE; i = i + 1) begin

exp_values[i] <= data_in[i*ACTIV_BITS +: ACTIV_BITS];

end

```

```

// Compute sum of exponential values

sum_exp <= 0;

for (i = 0; i < INPUT_SIZE; i = i + 1) begin

sum_exp <= sum_exp + {{ACTIV_BITS{1'b0}}, exp_values[i]};

end

// Compute softmax values

for (i = 0; i < INPUT_SIZE; i = i + 1) begin

softmax_values[i] <= (exp_values[i] << ACTIV_BITS) / sum_exp[2*ACTIV_BITS-
1:ACTIV_BITS];

end

// Assign output

for (i = 0; i < INPUT_SIZE; i = i + 1) begin

data_out[i*ACTIV_BITS +: ACTIV_BITS] <= softmax_values[i];

end

data_out_valid <= 1;

end else begin

data_out_valid <= 0;

end

end

endmodule

`endif".

```

Here are the refactored and optimized Verilog modules for the Keyword Spotting (KWS) accelerator:

1. Top-level module: `kws.v`

verilog

Copy code

```
`include "preemphasis_filter.v"
```

```

`include "framing_windowing.v"
`include "goertzel_dft.v"
`include "mel_filterbank.v"
`include "logarithm_comp.v"
`include "dct_comp.v"
`include "mfcc_accelerator.v"
`include "conv2d.v"
`include "maxpool2d.v"
`include "fully_connected.v"
`include "softmax.v"

module kws #(
    parameter NUM_KEYWORDS = 10,
    parameter MFCC_FEATURES = 40,
    parameter ACTIV_BITS = 8,
    parameter FC1_INPUT_SIZE = (MFCC_FEATURES/2)*(CONV2_NUM_FILTERS),
    parameter FC1_OUTPUT_SIZE = 64,
    parameter FC2_INPUT_SIZE = 64,
    parameter FC2_OUTPUT_SIZE = NUM_KEYWORDS,
    parameter CONV1_KERNEL_SIZE = 3,
    parameter CONV1_NUM_FILTERS = 8,
    parameter CONV2_KERNEL_SIZE = 3,
    parameter CONV2_NUM_FILTERS = 8
)()
    input wire clk,
    input wire rst_n,
    input wire [15:0] audio_in,
    input wire audio_valid,
    output reg [NUM_KEYWORDS-1:0] kws_result,
    output reg kws_valid,
    input wire [7:0] frame_size,
    input wire [7:0] frame_overlap,
    input wire [7:0] num_mfcc_coefs,
    input wire [4095:0] goertzel_coefs,
    input wire
[CONV1_NUM_FILTERS*CONV1_KERNEL_SIZE*CONV1_KERNEL_SIZE*ACTIV_BITS-1:0]
conv1_weights,
    input wire [CONV1_NUM_FILTERS*ACTIV_BITS-1:0] conv1_biases,
    input wire conv1_load_weights,
    input wire conv1_load_biases,
    input wire
[CONV2_NUM_FILTERS*CONV1_NUM_FILTERS*CONV2_KERNEL_SIZE*CONV2_KERNEL_SIZE*AC
TIV_BITS-1:0] conv2_weights,
    input wire [CONV2_NUM_FILTERS*ACTIV_BITS-1:0] conv2_biases,
    input wire conv2_load_weights,
    input wire conv2_load_biases,
    input wire [FC1_OUTPUT_SIZE*FC1_INPUT_SIZE*ACTIV_BITS-1:0] fc1_weights,
    input wire [FC1_OUTPUT_SIZE*ACTIV_BITS-1:0] fc1_biases,
    input wire fc1_load_weights,
    input wire fc1_load_biases,
    input wire [FC2_OUTPUT_SIZE*FC2_INPUT_SIZE*ACTIV_BITS-1:0] fc2_weights,
    input wire [FC2_OUTPUT_SIZE*ACTIV_BITS-1:0] fc2_biases,
    input wire fc2_load_weights,

```

```

        input wire fc2_load_biases
    );

    // MFCC module signals
    wire [MFCC_FEATURES*ACTIV_BITS-1:0] mfcc_out;
    wire mfcc_valid;

    // CNN-KWS layers
    wire [MFCC_FEATURES*CONV1_NUM_FILTERS*ACTIV_BITS-1:0] conv1_out;
    wire conv1_valid;
    wire [MFCC_FEATURES*CONV2_NUM_FILTERS*ACTIV_BITS-1:0] conv2_out;
    wire conv2_valid;
    wire [(MFCC_FEATURES/2)*CONV2_NUM_FILTERS*ACTIV_BITS-1:0] maxpool_out;
    wire maxpool_valid;
    wire [FC1_OUTPUT_SIZE*ACTIV_BITS-1:0] fc1_out;
    wire fc1_valid;
    wire [FC2_OUTPUT_SIZE*ACTIV_BITS-1:0] fc2_out;
    wire fc2_valid;
    wire [FC2_OUTPUT_SIZE*ACTIV_BITS-1:0] softmax_out;
    wire softmax_valid;

    // MFCC module instantiation
    mfcc_accelerator #(
        .MFCC_FEATURES(MFCC_FEATURES),
        .ACTIV_BITS(ACTIV_BITS)
    ) mfcc (
        .clk(clk),
        .rst_n(rst_n),
        .audio_in(audio_in),
        .audio_valid(audio_valid),
        .mfcc_out(mfcc_out),
        .mfcc_valid(mfcc_valid),
        .frame_size(frame_size),
        .frame_overlap(frame_overlap),
        .num_mfcc_coeffs(num_mfcc_coeffs),
        .goertzel_coefs(goertzel_coefs)
    );

    // Convolutional layer 1
    conv2d #(
        .INPUT_WIDTH(MFCC_FEATURES),
        .INPUT_HEIGHT(1),
        .INPUT_CHANNELS(1),
        .KERNEL_SIZE(CONV1_KERNEL_SIZE),
        .NUM_FILTERS(CONV1_NUM_FILTERS),
        .PADDING(1),
        .ACTIV_BITS(ACTIV_BITS)
    ) conv1 (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(mfcc_out),
        .data_valid(mfcc_valid),
        .data_out(conv1_out),

```

```

        .data_out_valid(conv1_valid),
        .weights_in(conv1_weights),
        .biases_in(conv1_biases),
        .load_weights(conv1_load_weights),
        .load_biases(conv1_load_biases)
    );

    // Convolutional layer 2
    conv2d #(
        .INPUT_WIDTH(MFCC_FEATURES),
        .INPUT_HEIGHT(1),
        .INPUT_CHANNELS(CONV1_NUM_FILTERS),
        .KERNEL_SIZE(CONV2_KERNEL_SIZE),
        .NUM_FILTERS(CONV2_NUM_FILTERS),
        .PADDING(1),
        .ACTIV_BITS(ACTIV_BITS)
    ) conv2 (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(conv1_out),
        .data_valid(conv1_valid),
        .data_out(conv2_out),
        .data_out_valid(conv2_valid),
        .weights_in(conv2_weights),
        .biases_in(conv2_biases),
        .load_weights(conv2_load_weights),
        .load_biases(conv2_load_biases)
    );

    maxpool2d #(
        .INPUT_WIDTH(MFCC_FEATURES),
        .INPUT_HEIGHT(1),
        .INPUT_CHANNELS(CONV2_NUM_FILTERS),
        .KERNEL_SIZE(2),
        .STRIDE(2),
        .ACTIV_BITS(ACTIV_BITS)
    ) maxpool (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(conv2_out),
        .data_valid(conv2_valid),
        .data_out(maxpool_out),
        .data_out_valid(maxpool_valid)
    );

    fully_connected #(
        .INPUT_SIZE(FC1_INPUT_SIZE),
        .OUTPUT_SIZE(FC1_OUTPUT_SIZE),
        .ACTIV_BITS(ACTIV_BITS)
    ) fc1 (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(maxpool_out),

```

```

        .data_valid(maxpool_valid),
        .data_out(fc1_out),
        .data_out_valid(fc1_valid),
        .weights_in(fc1_weights),
        .biases_in(fc1_biases),
        .load_weights(fc1_load_weights),
        .load_biases(fc1_load_biases)
    );

    // Fully connected layer 2 (output layer)
    fully_connected #(
        .INPUT_SIZE(FC2_INPUT_SIZE),
        .OUTPUT_SIZE(FC2_OUTPUT_SIZE),
        .ACTIV_BITS(ACTIV_BITS)
    ) fc2 (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(fc1_out),
        .data_valid(fc1_valid),
        .data_out(fc2_out),
        .data_out_valid(fc2_valid),
        .weights_in(fc2_weights),
        .biases_in(fc2_biases),
        .load_weights(fc2_load_weights),
        .load_biases(fc2_load_biases)
    );

    // Softmax layer
    softmax #(
        .INPUT_SIZE(NUM_KEYWORDS),
        .ACTIV_BITS(ACTIV_BITS)
    ) softmax (
        .clk(clk),
        .rst_n(rst_n),
        .data_in(fc2_out),
        .data_valid(fc2_valid),
        .data_out(softmax_out),
        .data_out_valid(softmax_valid)
    );

    // Output assignment
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            kws_result <= 'b0;
            kws_valid <= 1'b0;
        end else begin
            kws_result <= softmax_out[NUM_KEYWORDS-1:0];
            kws_valid <= softmax_valid;
        end
    end

endmodule

```


2. Convolutional layer: conv2d.v

verilog

Copy code

```
`ifndef CONV2D_V
`define CONV2D_V

module conv2d #(
    parameter INPUT_WIDTH = 40,
    parameter INPUT_HEIGHT = 1,
    parameter INPUT_CHANNELS = 1,
    parameter KERNEL_SIZE = 3,
    parameter NUM_FILTERS = 8,
    parameter PADDING = 1,
    parameter ACTIV_BITS = 8
) (
    input wire clk,
    input wire rst_n,
    input wire [INPUT_WIDTH*INPUT_HEIGHT*INPUT_CHANNELS*ACTIV_BITS-1:0]
data_in,
    input wire data_valid,
    output reg [INPUT_WIDTH*INPUT_HEIGHT*NUM_FILTERS*ACTIV_BITS-1:0]
data_out,
    output reg data_out_valid,
    input wire
[ NUM_FILTERS*INPUT_CHANNELS*KERNEL_SIZE*KERNEL_SIZE*ACTIV_BITS-1:0 ]
weights_in,
    input wire [NUM_FILTERS*ACTIV_BITS-1:0] biases_in,
    input wire load_weights,
    input wire load_biases
);

    // Declare weights and biases
    reg [ACTIV_BITS-1:0] weights [0:NUM_FILTERS-1][0:INPUT_CHANNELS-
1][0:KERNEL_SIZE-1][0:KERNEL_SIZE-1];
    reg [ACTIV_BITS-1:0] biases [0:NUM_FILTERS-1];

    // Declare internal signals
    reg [ACTIV_BITS-1:0] input_buffer [0:INPUT_HEIGHT-1][0:INPUT_WIDTH-
1][0:INPUT_CHANNELS-1];
    reg [2*ACTIV_BITS-1:0] conv_result [0:INPUT_HEIGHT-1][0:INPUT_WIDTH-
1][0:NUM_FILTERS-1];
    reg [ACTIV_BITS-1:0] relu_result [0:INPUT_HEIGHT-1][0:INPUT_WIDTH-
1][0:NUM_FILTERS-1];

    // Load weights and biases
    integer i_load, j_load, k_load, l_load;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            // Reset weights and biases
            for (i_load = 0; i_load < NUM_FILTERS; i_load = i_load + 1)
begin
```

```

        for (j_load = 0; j_load < INPUT_CHANNELS; j_load = j_load +
1) begin
            for (k_load = 0; k_load < KERNEL_SIZE; k_load = k_load
+ 1) begin
                for (l_load = 0; l_load < KERNEL_SIZE; l_load =
l_load + 1) begin
                    weights[i_load][j_load][k_load][l_load] <= 0;
                end
            end
        end
        biases[i_load] <= 0;
    end
end else begin
    // Load weights when load_weights is asserted
    if (load_weights) begin
        for (i_load = 0; i_load < NUM_FILTERS; i_load = i_load + 1)
begin
            for (j_load = 0; j_load < INPUT_CHANNELS; j_load =
j_load + 1) begin
                for (k_load = 0; k_load < KERNEL_SIZE; k_load =
k_load + 1) begin
                    for (l_load = 0; l_load < KERNEL_SIZE; l_load =
l_load + 1) begin
                        weights[i_load][j_load][k_load][l_load] <=
weights_in[(i_load*INPUT_CHANNELS*KERNEL_SIZE*KERNEL_SIZE +
j_load*KERNEL_SIZE*KERNEL_SIZE + k_load*KERNEL_SIZE + l_load)*ACTIV_BITS +:
ACTIV_BITS];
                    end
                end
            end
        end
    end
    // Load biases when load_biases is asserted
    if (load_biases) begin
        for (i_load = 0; i_load < NUM_FILTERS; i_load = i_load + 1)
begin
            biases[i_load] <= biases_in[i_load*ACTIV_BITS +:
ACTIV_BITS];
        end
    end
end
end

// Convolution operation
integer m_conv, n_conv, p_conv, q_conv, i_conv, j_conv, k_conv;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        // Reset internal signals and output
        for (i_conv = 0; i_conv < INPUT_HEIGHT; i_conv = i_conv + 1)
begin
            for (j_conv = 0; j_conv < INPUT_WIDTH; j_conv = j_conv + 1)
begin

```

```

                                for (k_conv = 0; k_conv < INPUT_CHANNELS; k_conv =
k_conv + 1) begin
                                    input_buffer[i_conv][j_conv][k_conv] <= 0;
                                end
                                for (m_conv = 0; m_conv < NUM_FILTERS; m_conv = m_conv
+ 1) begin
                                    conv_result[i_conv][j_conv][m_conv] <= 0;
                                    relu_result[i_conv][j_conv][m_conv] <= 0;
                                end
                            end
                        end
                        data_out <= 0;
                        data_out_valid <= 0;
                    end else if (data_valid) begin
                        // Shift input data into buffer
                        for (i

```

Copy

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

EB

Do you understand the codebase?

Yes, I understand the provided codebase. The codebase implements a hardware accelerator for Keyword Spotting (KWS) using a Convolutional Neural Network (CNN) model called CNN-KWS or "Hello Edge" model.

The main components of the codebase are:

1. Top-level module (`kws.v`): This module instantiates and connects all the sub-modules of the KWS accelerator, including the MFCC feature extraction module, convolutional layers, max pooling layer, fully connected layers, and softmax layer.
2. MFCC feature extraction module (`mfcc_accelerator.v`): This module takes the input audio signal and performs pre-emphasis filtering, framing, windowing, Discrete Fourier Transform (DFT) using Goertzel's algorithm, mel-scale filterbank computation, logarithm computation, and Discrete Cosine Transform (DCT) to extract the Mel-Frequency Cepstral Coefficients (MFCC) features.
3. Convolutional layers (`conv2d.v`): These layers perform 2D convolution operations on the input feature maps using learned weights and biases, followed by ReLU activation.
4. Max pooling layer (`maxpool2d.v`): This layer performs max pooling operation to downsample the feature maps and reduce spatial dimensions.
5. Fully connected layers (`fully_connected.v`): These layers perform matrix multiplication and ReLU activation on the flattened feature maps to generate the final output predictions.
6. Softmax layer (`softmax.v`): This layer applies the softmax activation function to the output of the last fully connected layer to generate the probability distribution over the keywords.

The codebase also includes modules for pre-emphasis filtering (`preemphasis_filter.v`), framing and windowing (`framing_windowing.v`), Goertzel's DFT computation

(goertzel_dft.v), mel-scale filterbank computation (mel_filterbank.v), logarithm computation (logarithm_comp.v), and DCT computation (dct_comp.v).

Overall, the codebase implements a hardware-efficient CNN-based KWS accelerator that can be deployed on resource-constrained edge devices for real-time keyword spotting applications.

Copy
EB

We are highly constrained in space. In the top kws module we have a signal "input wire [4095:0] goertzel_coefs,". This signal is used to transfer data to the goertzel_dft module. Create a prompt to refactor all the necessary modules in order to convert this to an 8bit signal. Then add a shift register inside the goertzel_dft module that will load the values into a LUT. There should be a signal to start the transfer and a valid signal for proper transfer and finally a signal for transfer done.

To refactor the necessary modules and convert the goertzel_coefs signal to an 8-bit signal, and add a shift register inside the goertzel_dft module for loading the values into a LUT, follow these steps:

1. Modify the kws module:

- Change the goertzel_coefs signal to an 8-bit signal:

verilog

Copy code

```
input wire [7:0] goertzel_coefs,
```

- Add signals for transfer control:

verilog

Copy code

```
output reg goertzel_coefs_start,  
input wire goertzel_coefs_valid,  
input wire goertzel_coefs_done,
```

2. Modify the goertzel_dft module:

- Change the goertzel_coefs signal to an 8-bit signal:

verilog

Copy code

```
input wire [7:0] goertzel_coefs,
```

- Add signals for transfer control:

verilog

Copy code

```
input wire goertzel_coefs_start,
output reg goertzel_coefs_valid,
output reg goertzel_coefs_done,
```

- Add a shift register and LUT for storing the Goertzel coefficients:

verilog

Copy code

```
localparam GOERTZEL_COEFS_SIZE = 512; // Adjust the size as
needed
reg [7:0] goertzel_coefs_reg [0:GOERTZEL_COEFS_SIZE-1];
reg [$clog2(GOERTZEL_COEFS_SIZE)-1:0] goertzel_coefs_cnt;
```

- Implement the shift register and LUT loading process:

verilog

Copy code

```
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        goertzel_coefs_cnt <= 0;
        goertzel_coefs_valid <= 0;
        goertzel_coefs_done <= 0;
    end else if (goertzel_coefs_start) begin
        goertzel_coefs_reg[goertzel_coefs_cnt] <=
goertzel_coefs;
        goertzel_coefs_cnt <= goertzel_coefs_cnt + 1;
        goertzel_coefs_valid <= 1;
        if (goertzel_coefs_cnt == GOERTZEL_COEFS_SIZE-1) begin
            goertzel_coefs_done <= 1;
        end
    end else begin
        goertzel_coefs_valid <= 0;
        goertzel_coefs_done <= 0;
    end
end
```

- Use the loaded Goertzel coefficients from the LUT in the Goertzel algorithm:

verilog

Copy code

```
// Replace the usage of goertzel_coefs with
goertzel_coefs_reg[index]
// Example:
// q_curr_temp = (goertzel_coefs_reg[j] * q_prev[j] >>> 15) -
q_curr[j] + {{16{framed_out[15]}}}, framed_out};
```

3. Instantiate the modified goertzel_dft module in the kws module:

verilog

Copy code

```
goertzel_dft dft (
    .clk(clk),
    .rst_n(rst_n),
    .framed_out(framed_out),
    .framed_valid(framed_valid),
    .goertzel_coefs(goertzel_coefs),
    .goertzel_coefs_start(goertzel_coefs_start),
    .goertzel_coefs_valid(goertzel_coefs_valid),
    .goertzel_coefs_done(goertzel_coefs_done),
    .dft_out(dft_out),
    .dft_valid(dft_valid)
);
```

4. Modify the `kws` module to control the transfer of Goertzel coefficients:
 - Add a state machine or control logic to initiate the transfer and wait for the `goertzel_coefs_done` signal.
 - Set the `goertzel_coefs_start` signal to start the transfer.
 - Monitor the `goertzel_coefs_valid` signal to ensure proper transfer.
 - Wait for the `goertzel_coefs_done` signal to indicate the completion of the transfer.

These modifications will allow you to convert the `goertzel_coefs` signal to an 8-bit signal and use a shift register inside the `goertzel_dft` module to load the values into a LUT. The transfer process will be controlled by the `goertzel_coefs_start`, `goertzel_coefs_valid`, and `goertzel_coefs_done` signals.

Copy

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

EB

To refactor the necessary modules and convert the `goertzel_coefs` signal to an 8-bit signal, and add a shift register inside the `goertzel_dft` module for loading the values into a LUT, follow these steps:

1. Modify the `kws` module:
 - Change the `goertzel_coefs` signal to an 8-bit signal:

Copy code

```
verilogCopy code
```

```
input wire [7:0] goertzel_coefs,
```

- Add signals for transfer control:

Copy code

```
verilogCopy code
```

```
output reg goertzel_coefs_start, input wire
goertzel_coefs_valid, input wire goertzel_coefs_done,
```

2. Modify the `goertzel_dft` module:

- Change the `goertzel_coefs` signal to an 8-bit signal:

Copy code

```
verilogCopy code
```

```
input wire [7:0] goertzel_coefs,
```

- Add signals for transfer control:

Copy code

```
verilogCopy code
```

```
input wire goertzel_coefs_start, output reg  
goertzel_coefs_valid, output reg goertzel_coefs_done,
```

- Add a shift register and LUT for storing the Goertzel coefficients:

Copy code

```
verilogCopy code
```

```
localparam GOERTZEL_COEFS_SIZE = 512; // Adjust the size as  
needed reg [7:0] goertzel_coefs_reg [0:GOERTZEL_COEFS_SIZE-1];  
reg [$clog2(GOERTZEL_COEFS_SIZE)-1:0] goertzel_coefs_cnt;
```

- Implement the shift register and LUT loading process:

Copy code

```
verilogCopy code
```

```
always @(posedge clk or negedge rst_n) begin if (!rst_n) begin  
goertzel_coefs_cnt <= 0; goertzel_coefs_valid <= 0;  
goertzel_coefs_done <= 0; end else if (goertzel_coefs_start)  
begin goertzel_coefs_reg[goertzel_coefs_cnt] <= goertzel_coefs;  
goertzel_coefs_cnt <= goertzel_coefs_cnt + 1;  
goertzel_coefs_valid <= 1; if (goertzel_coefs_cnt ==  
GOERTZEL_COEFS_SIZE-1) begin goertzel_coefs_done <= 1; end end  
else begin goertzel_coefs_valid <= 0; goertzel_coefs_done <= 0;  
end end
```

- Use the loaded Goertzel coefficients from the LUT in the Goertzel algorithm:

Copy code

```
verilogCopy code
```

```
// Replace the usage of goertzel_coefs with  
goertzel_coefs_reg[index] // Example: // q_curr_temp =  
(goertzel_coefs_reg[j] * q_prev[j] >>> 15) - q_curr[j] +  
{16{framed_out[15]}}, framed_out};
```

3. Instantiate the modified `goertzel_dft` module in the `kws` module:

Copy code

```
verilogCopy code
```

```

goertzel_dft dft ( .clk(clk), .rst_n(rst_n), .framed_out(framed_out),
    .framed_valid(framed_valid), .goertzel_coefs(goertzel_coefs),
    .goertzel_coefs_start(goertzel_coefs_start),
    .goertzel_coefs_valid(goertzel_coefs_valid),
    .goertzel_coefs_done(goertzel_coefs_done), .dft_out(dft_out),
    .dft_valid(dft_valid) );

```

4. Modify the `kws` module to control the transfer of Goertzel coefficients:
 - Add a state machine or control logic to initiate the transfer and wait for the `goertzel_coefs_done` signal.
 - Set the `goertzel_coefs_start` signal to start the transfer.
 - Monitor the `goertzel_coefs_valid` signal to ensure proper transfer.
 - Wait for the `goertzel_coefs_done` signal to indicate the completion of the transfer.

These modifications will allow you to convert the `goertzel_coefs` signal to an 8-bit signal and use a shift register inside the `goertzel_dft` module to load the values into a LUT. The transfer process will be controlled by the `goertzel_coefs_start`, `goertzel_coefs_valid`, and `goertzel_coefs_done` signals.

Here's the refactored code for the `kws` and `goertzel_dft` modules based on the provided steps:

1. Refactored `kws` module:

verilog

Copy code

```

module kws #(
    // ... (other parameters)
)()
    // ... (other ports)
    input wire [7:0] goertzel_coefs,
    output reg goertzel_coefs_start,
    input wire goertzel_coefs_valid,
    input wire goertzel_coefs_done,
    // ... (other ports)
);

    // ... (other code)

    // Instantiate the modified goertzel_dft module
    goertzel_dft dft (
        .clk(clk),
        .rst_n(rst_n),
        .framed_out(framed_out),
        .framed_valid(framed_valid),
        .goertzel_coefs(goertzel_coefs),
        .goertzel_coefs_start(goertzel_coefs_start),
        .goertzel_coefs_valid(goertzel_coefs_valid),
        .goertzel_coefs_done(goertzel_coefs_done),
        .dft_out(dft_out),

```



```

        .dft_valid(dft_valid)
    );

    // ... (other code)

    // Control logic for Goertzel coefficients transfer
    localparam IDLE = 2'b00;
    localparam START_TRANSFER = 2'b01;
    localparam WAIT_TRANSFER = 2'b10;

    reg [1:0] state;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= IDLE;
            goertzel_coefs_start <= 0;
        end else begin
            case (state)
                IDLE: begin
                    // Initiate the transfer
                    state <= START_TRANSFER;
                end
                START_TRANSFER: begin
                    // Set the start signal
                    goertzel_coefs_start <= 1;
                    state <= WAIT_TRANSFER;
                end
                WAIT_TRANSFER: begin
                    // Clear the start signal
                    goertzel_coefs_start <= 0;
                    // Wait for the transfer to complete
                    if (goertzel_coefs_done) begin
                        state <= IDLE;
                    end
                end
            endcase
        end
    end
endmodule

```

2. Refactored goertzel_dft module:

verilog

Copy code

```

module goertzel_dft #(
    parameter NUM_FREQS = 256
)(
    // ... (other ports)
    input wire [7:0] goertzel_coefs,
    input wire goertzel_coefs_start,
    output reg goertzel_coefs_valid,

```

```

output reg goertzel_coefs_done,
// ... (other ports)
);

// ... (other code)

// Shift register and LUT for storing Goertzel coefficients
localparam GOERTZEL_COEFS_SIZE = 512; // Adjust the size as needed
reg [7:0] goertzel_coefs_reg [0:GOERTZEL_COEFS_SIZE-1];
reg [$clog2(GOERTZEL_COEFS_SIZE)-1:0] goertzel_coefs_cnt;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        goertzel_coefs_cnt <= 0;
        goertzel_coefs_valid <= 0;
        goertzel_coefs_done <= 0;
    end else if (goertzel_coefs_start) begin
        goertzel_coefs_reg[goertzel_coefs_cnt] <= goertzel_coefs;
        goertzel_coefs_cnt <= goertzel_coefs_cnt + 1;
        goertzel_coefs_valid <= 1;
        if (goertzel_coefs_cnt == GOERTZEL_COEFS_SIZE-1) begin
            goertzel_coefs_done <= 1;
        end
    end else begin
        goertzel_coefs_valid <= 0;
        goertzel_coefs_done <= 0;
    end
end

// ... (other code)

// Use the loaded Goertzel coefficients from the LUT
always @(posedge clk or negedge rst_n) begin
    // ... (other code)
    for (j = 0; j < NUM_FREQS; j = j + 1) begin
        // ... (other code)
        q_curr_temp = (goertzel_coefs_reg[j] * q_prev[j] >>> 15) -
q_curr[j] + {{16{framed_out[15]}}, framed_out};
        // ... (other code)
    end
    // ... (other code)
end

// ... (other code)

endmodule

```

In the refactored `kws` module, the `goertzel_coefs` signal is changed to an 8-bit signal, and additional signals (`goertzel_coefs_start`, `goertzel_coefs_valid`, `goertzel_coefs_done`) are added for transfer control. The modified `goertzel_dft` module is instantiated with the new signals.

A simple state machine is added to control the transfer of Goertzel coefficients. When the state machine enters the `START_TRANSFER` state, it sets the `goertzel_coefs_start` signal to initiate the transfer. It then waits for the `goertzel_coefs_done` signal to indicate the completion of the transfer before returning to the `IDLE` state.

In the refactored `goertzel_dft` module, the `goertzel_coefs` signal is changed to an 8-bit signal, and additional signals (`goertzel_coefs_start`, `goertzel_coefs_valid`, `goertzel_coefs_done`) are added for transfer control. A shift register and LUT (`goertzel_coefs_reg`) are added to store the Goertzel coefficients.

When the `goertzel_coefs_start` signal is asserted, the module starts loading the Goertzel coefficients into the shift register and LUT. It sets the `goertzel_coefs_valid` signal to indicate valid data transfer and increments the `goertzel_coefs_cnt` counter. When the counter reaches the size of the LUT (`GOERTZEL_COEFS_SIZE-1`), it asserts the `goertzel_coefs_done` signal to indicate the completion of the transfer.

Finally, in the Goertzel algorithm, the usage of `goertzel_coefs` is replaced with `goertzel_coefs_reg[j]` to use the loaded coefficients from the LUT.

These modifications allow the `goertzel_coefs` signal to be converted to an 8-bit signal and use a shift register and LUT inside the `goertzel_dft` module for loading the coefficients. The transfer process is controlled by the added signals (`goertzel_coefs_start`, `goertzel_coefs_valid`, `goertzel_coefs_done`).

Copy

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

EB

The instantiation of `goertzel_dft` *// Instantiate the modified goertzel_dft module* `goertzel_dft dft (.clk(clk), .rst_n(rst_n), .framed_out(framed_out), .framed_valid(framed_valid), .goertzel_coefs(goertzel_coefs), .goertzel_coefs_start(goertzel_coefs_start), .goertzel_coefs_valid(goertzel_coefs_valid), .goertzel_coefs_done(goertzel_coefs_done), .dft_out(dft_out), .dft_valid(dft_valid));` should be done inside the `mfcc_accelerator` module. The `kws` top design should be refactored for the changed signals and `mfcc_accelerator` module instantiation.

I apologize for the confusion. Here's the refactored code for the `kws` and `mfcc_accelerator` modules based on your instructions:

1. Refactored `kws` module:

verilog

Copy code

```
module kws #(
    // ... (other parameters)
)()
    // ... (other ports)
    input wire [7:0] goertzel_coefs,
    output reg goertzel_coefs_start,
```

```

    input wire goertzel_coefs_valid,
    input wire goertzel_coefs_done,
    // ... (other ports)
);

// ... (other code)

// MFCC module signals
wire [MFCC_FEATURES*ACTIV_BITS-1:0] mfcc_out;
wire mfcc_valid;

// MFCC module instantiation
mfcc_accelerator #(
    .MFCC_FEATURES(MFCC_FEATURES),
    .ACTIV_BITS(ACTIV_BITS)
) mfcc (
    .clk(clk),
    .rst_n(rst_n),
    .audio_in(audio_in),
    .audio_valid(audio_valid),
    .mfcc_out(mfcc_out),
    .mfcc_valid(mfcc_valid),
    .frame_size(frame_size),
    .frame_overlap(frame_overlap),
    .num_mfcc_coeffs(num_mfcc_coeffs),
    .goertzel_coefs(goertzel_coefs),
    .goertzel_coefs_start(goertzel_coefs_start),
    .goertzel_coefs_valid(goertzel_coefs_valid),
    .goertzel_coefs_done(goertzel_coefs_done)
);

// ... (other code)

endmodule

```

2. Refactored mfcc_accelerator module:

verilog

Copy code

```

module mfcc_accelerator #(
    parameter MFCC_FEATURES = 40,
    parameter ACTIV_BITS = 8
) (
    // ... (other ports)
    input wire [7:0] goertzel_coefs,
    input wire goertzel_coefs_start,
    output wire goertzel_coefs_valid,
    output wire goertzel_coefs_done,
    // ... (other ports)
);

// ... (other code)

```

```

// Instantiate the modified goertzel_dft module
goertzel_dft dft (
    .clk(clk),
    .rst_n(rst_n),
    .framed_out(framed_out),
    .framed_valid(framed_valid),
    .goertzel_coefs(goertzel_coefs),
    .goertzel_coefs_start(goertzel_coefs_start),
    .goertzel_coefs_valid(goertzel_coefs_valid),
    .goertzel_coefs_done(goertzel_coefs_done),
    .dft_out(dft_out),
    .dft_valid(dft_valid)
);

// ... (other code)

endmodule

```

In the refactored `kws` module:

- The `goertzel_coefs` signal is changed to an 8-bit signal.
- Additional signals (`goertzel_coefs_start`, `goertzel_coefs_valid`, `goertzel_coefs_done`) are added for transfer control.
- The `mfcc_accelerator` module is instantiated with the new signals.

In the refactored `mfcc_accelerator` module:

- The `goertzel_coefs` signal is changed to an 8-bit signal.
- Additional signals (`goertzel_coefs_start`, `goertzel_coefs_valid`, `goertzel_coefs_done`) are added for transfer control.
- The modified `goertzel_dft` module is instantiated with the new signals.

Now, the instantiation of the `goertzel_dft` module is done inside the `mfcc_accelerator` module, and the `kws` module is refactored to accommodate the changed signals and the `mfcc_accelerator` module instantiation.

Please note that you still need to implement the control logic for the Goertzel coefficients transfer in the `kws` module, as shown in the previous response.

Copy

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

EB

The contest requires all code to be provided by the LLM. Please provide the code for the `goertzel_dft` module

Here's the refactored code for the `goertzel_dft` module with the 8-bit `goertzel_coefs` signal and the shift register for loading the coefficients into a LUT:

verilog

Copy code

```
`ifndef GOERTZEL_DFT_V
`define GOERTZEL_DFT_V

module goertzel_dft #(
    parameter NUM_FREQS = 256
)(
    input wire clk,
    input wire rst_n,
    input wire [15:0] framed_out,
    input wire framed_valid,
    input wire [7:0] goertzel_coefs,
    input wire goertzel_coefs_start,
    output reg goertzel_coefs_valid,
    output reg goertzel_coefs_done,
    output reg [31:0] dft_out,
    output reg dft_valid
);

    // Goertzel algorithm variables
    reg [31:0] q_prev [0:NUM_FREQS-1];
    reg [31:0] q_curr [0:NUM_FREQS-1];
    reg [$clog2(NUM_FREQS)-1:0] freq_idx;

    // Shift register and LUT for storing Goertzel coefficients
    localparam GOERTZEL_COEFS_SIZE = 512; // Adjust the size as needed
    reg [7:0] goertzel_coefs_reg [0:GOERTZEL_COEFS_SIZE-1];
    reg [$clog2(GOERTZEL_COEFS_SIZE)-1:0] goertzel_coefs_cnt;

    // Goertzel coefficients loading process
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            goertzel_coefs_cnt <= 0;
            goertzel_coefs_valid <= 0;
            goertzel_coefs_done <= 0;
        end else if (goertzel_coefs_start) begin
            goertzel_coefs_reg[goertzel_coefs_cnt] <= goertzel_coefs;
            goertzel_coefs_cnt <= goertzel_coefs_cnt + 1;
            goertzel_coefs_valid <= 1;
            if (goertzel_coefs_cnt == GOERTZEL_COEFS_SIZE-1) begin
                goertzel_coefs_done <= 1;
            end
        end else begin
            goertzel_coefs_valid <= 0;
            goertzel_coefs_done <= 0;
        end
    end

    // Goertzel algorithm implementation
    integer j;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            for (j = 0; j < NUM_FREQS; j = j + 1) begin
```

```

        q_prev[j] <= 32'h0;
        q_curr[j] <= 32'h0;
    end
    freq_idx <= 'h0;
    dft_out <= 32'h0;
    dft_valid <= 1'b0;
end else if (framed_valid) begin
    for (j = 0; j < NUM_FREQS; j = j + 1) begin
        // Create temporary variables
        reg [31:0] q_curr_temp;
        reg [31:0] q_prev_temp;

        // Compute Goertzel algorithm using loaded coefficients
        from LUT
        q_curr_temp = (goertzel_coefs_reg[j] * q_prev[j] >>> 15) -
q_curr[j] + {{16{framed_out[15]}}, framed_out};
        q_prev_temp = q_curr[j];

        // Assign the updated values to the arrays
        q_curr[j] <= q_curr_temp;
        q_prev[j] <= q_prev_temp;
    end

    // Increment frequency index
    freq_idx <= freq_idx + 1;

    // Output DFT result when all frequencies are processed
    if (freq_idx == NUM_FREQS[$clog2(NUM_FREQS)-1:0] - 1) begin
        dft_out <= q_curr[freq_idx]*q_curr[freq_idx] +
q_prev[freq_idx]*q_prev[freq_idx] - (goertzel_coefs_reg[freq_idx] *
q_curr[freq_idx] >>> 15);
        dft_valid <= 1'b1;
        freq_idx <= 'h0;
    end else begin
        dft_valid <= 1'b0;
    end
end else begin
    dft_valid <= 1'b0;
end
end

endmodule

`endif

```

In this refactored `goertzel_dft` module:

1. The `goertzel_coefs` input signal is changed to an 8-bit signal.
2. Additional input signals `goertzel_coefs_start`, `goertzel_coefs_valid`, and `goertzel_coefs_done` are added for controlling the transfer of Goertzel coefficients.
3. A shift register `goertzel_coefs_reg` and a counter `goertzel_coefs_cnt` are introduced to store and manage the loading of Goertzel coefficients into a LUT.

4. The loading process is controlled by the `goertzel_coefs_start` signal. When asserted, the coefficients are loaded into the shift register, and the `goertzel_coefs_valid` signal is set to indicate valid data transfer. The `goertzel_coefs_done` signal is asserted when all coefficients have been loaded.
5. In the Goertzel algorithm implementation, the loaded coefficients are accessed from the LUT using `goertzel_coefs_reg[j]` instead of directly using the `goertzel_coefs` input signal.
6. The rest of the module remains the same, with the Goertzel algorithm being computed for each frequency and the DFT output being generated when all frequencies have been processed.

This refactored module allows for the efficient transfer and storage of Goertzel coefficients using an 8-bit signal and a shift register, while maintaining the functionality of the Goertzel algorithm for DFT computation.

Copy

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

EB

Synthesis failed with error "%Error-ASSIGNIN: /openlane/designs/kws/src/kws.v:91:31: Assigning to input/const variable: 'goertzel_coefs_valid'"

```
91 | .goertzel_coefs_valid(goertzel_coefs_valid),
```

```
| ^~~~~~
```

... For error description see <https://verilator.org/warn/ASSIGNIN?v=5.009>

%Error-ASSIGNIN: /openlane/designs/kws/src/kws.v:92:30: Assigning to input/const variable: 'goertzel_coefs_done'

```
92 | .goertzel_coefs_done(goertzel_coefs_done)"
```

Claude does not have internet access and cannot get up to date info from web links.

I apologize for the oversight. It seems that the `goertzel_coefs_valid` and `goertzel_coefs_done` signals were incorrectly assigned as outputs in the `kws` module instantiation of the `mfcc_accelerator`. Let me provide the corrected code for the `kws` module:

verilog

Copy code

```
module kws #(
    // ... (other parameters)
)()
    // ... (other ports)
    input wire [7:0] goertzel_coefs,
    output reg goertzel_coefs_start,
    output wire goertzel_coefs_valid,
```



```

output wire goertzel_coefs_done,
// ... (other ports)
);

// ... (other code)

// MFCC module signals
wire [MFCC_FEATURES*ACTIV_BITS-1:0] mfcc_out;
wire mfcc_valid;

// MFCC module instantiation
mfcc_accelerator #(
    .MFCC_FEATURES(MFCC_FEATURES),
    .ACTIV_BITS(ACTIV_BITS)
) mfcc (
    .clk(clk),
    .rst_n(rst_n),
    .audio_in(audio_in),
    .audio_valid(audio_valid),
    .mfcc_out(mfcc_out),
    .mfcc_valid(mfcc_valid),
    .frame_size(frame_size),
    .frame_overlap(frame_overlap),
    .num_mfcc_coefs(num_mfcc_coefs),
    .goertzel_coefs(goertzel_coefs),
    .goertzel_coefs_start(goertzel_coefs_start),
    .goertzel_coefs_valid(goertzel_coefs_valid),
    .goertzel_coefs_done(goertzel_coefs_done)
);

// ... (other code)

endmodule

```

In this corrected version of the `kws` module:

- The `goertzel_coefs_valid` and `goertzel_coefs_done` signals are declared as output wires instead of inputs.
- The `mfcc_accelerator` module instantiation connects these signals correctly as outputs.

Please update the `kws` module with this corrected code and re-run the synthesis. It should resolve the assignment errors related to `goertzel_coefs_valid` and `goertzel_coefs_done`.

I apologize for any confusion caused by my previous response. Let me know if you have any further questions or if there are any other issues with the code.

Copy

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

EB

Synthesis found these warnings "%Warning-WIDTHEXPAND:
/openlane/designs/kws/src/goertzel_dft.v:39:36: Operator EQ expects 32 or 10 bits on the LHS, but LHS's VARREF 'goertzel_coefs_cnt' generates 9 bits: In instance kws.mfcc.dft

39 | if (goertzel_coefs_cnt == GOERTZEL_COEFS_SIZE-1) begin", "%Warning-WIDTHEXPAND: /openlane/designs/kws/src/goertzel_dft.v:79:119: Bit extraction of array[511:0] requires 9 bit index, not 8 bits : In instance kws.mfcc.dft

79 | dft_out <= q_curr[freq_idx]*q_curr[freq_idx] + q_prev[freq_idx]*q_prev[freq_idx] - (goertzel_coefs_reg[freq_idx] * q_curr[freq_idx] >>> 15);". Based on the CNN-KWS model, also known as the "Hello Edge" model that was introduced in the paper "Hello Edge: Keyword Spotting on Microcontrollers" by Zhang et al. (2017) and the top module kws, what are the requirements for the goertzel_dft module. write a prompt for it.

Based on the CNN-KWS model, also known as the "Hello Edge" model introduced in the paper "Hello Edge: Keyword Spotting on Microcontrollers" by Zhang et al. (2017), and the top module kws, the requirements for the goertzel_dft module are as follows:

1. The goertzel_dft module should perform the Discrete Fourier Transform (DFT) on the framed audio data using the Goertzel algorithm.
2. The module should have the following inputs:
 - o clk: Clock signal
 - o rst_n: Active-low reset signal
 - o framed_out: 16-bit framed audio data
 - o framed_valid: Valid signal for the framed audio data
 - o goertzel_coefs: 8-bit Goertzel coefficients
 - o goertzel_coefs_start: Start signal for loading Goertzel coefficients
 - o num_freqs: Number of frequencies to compute DFT for (e.g., 256)
3. The module should have the following outputs:
 - o dft_out: 32-bit DFT output
 - o dft_valid: Valid signal for the DFT output
 - o goertzel_coefs_done: Done signal indicating the completion of Goertzel coefficients loading
4. The module should use a shift register and a lookup table (LUT) to store and manage the Goertzel coefficients. The shift register size should be determined based on the number of frequencies (num_freqs) and the coefficient size (8 bits).
5. The loading of Goertzel coefficients should be controlled by the goertzel_coefs_start signal. When asserted, the coefficients should be loaded into the shift register, and the goertzel_coefs_done signal should be asserted when all coefficients have been loaded.
6. The Goertzel algorithm should be implemented to compute the DFT for each frequency. The algorithm should use the loaded Goertzel coefficients from the LUT.
7. The DFT output should be calculated as the sum of the squared values of the real and imaginary parts of the Goertzel algorithm output, minus the product of the corresponding Goertzel coefficient and the real part of the Goertzel algorithm output, right-shifted by 15 bits.
8. The dft_valid signal should be asserted when the DFT output is valid, which occurs after all frequencies have been processed.
9. The module should be parameterized to allow flexibility in the number of frequencies (num_freqs) and the coefficient size (8 bits).

10. The module should handle the necessary bit widths and arithmetic operations correctly to avoid warnings during synthesis.

Based on these requirements, please write a Verilog module for `goertzel_dft` that efficiently implements the Goertzel algorithm for DFT computation, with proper handling of Goertzel coefficients loading and output generation. The module should be optimized for resource usage and compatible with the CNN-KWS model and the top module `kws`.

Copy
EB

1. The `goertzel_dft` module should perform the Discrete Fourier Transform (DFT) on the framed audio data using the Goertzel algorithm.
2. The module should have the following inputs:
 - o `clk`: Clock signal
 - o `rst_n`: Active-low reset signal
 - o `framed_out`: 16-bit framed audio data
 - o `framed_valid`: Valid signal for the framed audio data
 - o `goertzel_coefs`: 8-bit Goertzel coefficients
 - o `goertzel_coefs_start`: Start signal for loading Goertzel coefficients
 - o `num_freqs`: Number of frequencies to compute DFT for (e.g., 256)
3. The module should have the following outputs:
 - o `dft_out`: 32-bit DFT output
 - o `dft_valid`: Valid signal for the DFT output
 - o `goertzel_coefs_done`: Done signal indicating the completion of Goertzel coefficients loading
4. The module should use a shift register and a lookup table (LUT) to store and manage the Goertzel coefficients. The shift register size should be determined based on the number of frequencies (`num_freqs`) and the coefficient size (8 bits).
5. The loading of Goertzel coefficients should be controlled by the `goertzel_coefs_start` signal. When asserted, the coefficients should be loaded into the shift register, and the `goertzel_coefs_done` signal should be asserted when all coefficients have been loaded.
6. The Goertzel algorithm should be implemented to compute the DFT for each frequency. The algorithm should use the loaded Goertzel coefficients from the LUT.
7. The DFT output should be calculated as the sum of the squared values of the real and imaginary parts of the Goertzel algorithm output, minus the product of the corresponding Goertzel coefficient and the real part of the Goertzel algorithm output, right-shifted by 15 bits.
8. The `dft_valid` signal should be asserted when the DFT output is valid, which occurs after all frequencies have been processed.
9. The module should be parameterized to allow flexibility in the number of frequencies (`num_freqs`) and the coefficient size (8 bits).
10. The module should handle the necessary bit widths and arithmetic operations correctly to avoid warnings during synthesis.

Based on these requirements, please write a Verilog module for `goertzel_dft` that efficiently implements the Goertzel algorithm for DFT computation, with proper handling of

Goertzel coefficients loading and output generation. The module should be optimized for resource usage and compatible with the CNN-KWS model and the top module `kws`.

Here's the Verilog code for the `goertzel_dft` module based on the given requirements:

verilog

Copy code

```
`ifndef GOERTZEL_DFT_V
`define GOERTZEL_DFT_V

module goertzel_dft #(
    parameter NUM_FREQS = 256,
    parameter COEF_SIZE = 8
)(
    input wire clk,
    input wire rst_n,
    input wire [15:0] framed_out,
    input wire framed_valid,
    input wire [COEF_SIZE-1:0] goertzel_coefs,
    input wire goertzel_coefs_start,
    output reg goertzel_coefs_done,
    output reg [31:0] dft_out,
    output reg dft_valid
);

    // Goertzel algorithm variables
    reg [31:0] q_prev [0:NUM_FREQS-1];
    reg [31:0] q_curr [0:NUM_FREQS-1];
    reg [$clog2(NUM_FREQS)-1:0] freq_idx;

    // Shift register and LUT for storing Goertzel coefficients
    reg [COEF_SIZE-1:0] goertzel_coefs_reg [0:NUM_FREQS-1];
    reg [$clog2(NUM_FREQS)-1:0] goertzel_coefs_cnt;

    // Goertzel coefficients loading process
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            goertzel_coefs_cnt <= 0;
            goertzel_coefs_done <= 0;
        end else if (goertzel_coefs_start) begin
            goertzel_coefs_reg[goertzel_coefs_cnt] <= goertzel_coefs;
            if (goertzel_coefs_cnt == NUM_FREQS-1) begin
                goertzel_coefs_done <= 1;
                goertzel_coefs_cnt <= 0;
            end else begin
                goertzel_coefs_cnt <= goertzel_coefs_cnt + 1;
            end
        end else begin
            goertzel_coefs_done <= 0;
        end
    end
end
```

```

// Goertzel algorithm implementation
integer j;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        for (j = 0; j < NUM_FREQS; j = j + 1) begin
            q_prev[j] <= 32'h0;
            q_curr[j] <= 32'h0;
        end
        freq_idx <= 0;
        dft_out <= 32'h0;
        dft_valid <= 1'b0;
    end else if (framed_valid) begin
        for (j = 0; j < NUM_FREQS; j = j + 1) begin
            // Create temporary variables
            reg [31:0] q_curr_temp;
            reg [31:0] q_prev_temp;

            // Compute Goertzel algorithm using loaded coefficients
            from LUT
            q_curr_temp = ({24{goertzel_coefs_reg[j][COEF_SIZE-1]}},
            goertzel_coefs_reg[j]} * q_prev[j] >>> 15) - q_curr[j] +
            {{16{framed_out[15]}}, framed_out};
            q_prev_temp = q_curr[j];

            // Assign the updated values to the arrays
            q_curr[j] <= q_curr_temp;
            q_prev[j] <= q_prev_temp;
        end

        // Increment frequency index
        freq_idx <= freq_idx + 1;

        // Output DFT result when all frequencies are processed
        if (freq_idx == NUM_FREQS-1) begin
            dft_out <= q_curr[freq_idx]*q_curr[freq_idx] +
            q_prev[freq_idx]*q_prev[freq_idx] -
            ({24{goertzel_coefs_reg[freq_idx][COEF_SIZE-1]}},
            goertzel_coefs_reg[freq_idx]} * q_curr[freq_idx] >>> 15);
            dft_valid <= 1'b1;
            freq_idx <= 0;
        end else begin
            dft_valid <= 1'b0;
        end
    end else begin
        dft_valid <= 1'b0;
    end
end

endmodule

`endif

```

Explanation:

1. The `goertzel_dft` module is parameterized with `NUM_FREQS` (number of frequencies) and `COEF_SIZE` (coefficient size, default is 8 bits).
2. The module has the required inputs: `clk`, `rst_n`, `framed_out`, `framed_valid`, `goertzel_coefs`, and `goertzel_coefs_start`. The `num_freqs` parameter is used instead of an input.
3. The module has the required outputs: `dft_out`, `dft_valid`, and `goertzel_coefs_done`.
4. The Goertzel coefficients are stored in a lookup table (`goertzel_coefs_reg`) of size `NUM_FREQS`.
5. The loading of Goertzel coefficients is controlled by the `goertzel_coefs_start` signal. When asserted, the coefficients are loaded into the LUT, and the `goertzel_coefs_done` signal is asserted when all coefficients have been loaded.
6. The Goertzel algorithm is implemented using the loaded coefficients from the LUT. The algorithm updates the `q_prev` and `q_curr` arrays for each frequency.
7. The DFT output is calculated as the sum of the squared values of `q_curr` and `q_prev`, minus the product of the corresponding Goertzel coefficient and `q_curr`, right-shifted by 15 bits.
8. The `dft_valid` signal is asserted when the DFT output is valid, which occurs after all frequencies have been processed.
9. The module is parameterized with `NUM_FREQS` and `COEF_SIZE` to allow flexibility.
10. The necessary bit widths and arithmetic operations are handled correctly to avoid synthesis warnings. Sign extension is used when multiplying the Goertzel coefficients with `q_prev` and `q_curr`.

This `goertzel_dft` module efficiently implements the Goertzel algorithm for DFT computation, with proper handling of Goertzel coefficients loading and output generation. It is optimized for resource usage and compatible with the CNN-KWS model and the top module `kws`.

Copy

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

EB

You forgot the "`goertzel_coefs_valid`" signal. The logic is shift register `goertzel_coefs_reg` and a counter `goertzel_coefs_cnt` are introduced to store and manage the loading of Goertzel coefficients into a LUT.

- The loading process is controlled by the `goertzel_coefs_start` signal. When asserted, the coefficients are loaded into the shift register, and the `goertzel_coefs_valid` signal is set to indicate valid data transfer. The `goertzel_coefs_done` signal is asserted when all coefficients have been loaded.

I apologize for the oversight. Here's the updated Verilog code for the `goertzel_dft` module with the `goertzel_coefs_valid` signal added:

verilog

Copy code

```
`ifndef GOERTZEL_DFT_V
`define GOERTZEL_DFT_V
```

```

module goertzel_dft #(
    parameter NUM_FREQS = 256,
    parameter COEF_SIZE = 8
)(
    input wire clk,
    input wire rst_n,
    input wire [15:0] framed_out,
    input wire framed_valid,
    input wire [COEF_SIZE-1:0] goertzel_coefs,
    input wire goertzel_coefs_start,
    output reg goertzel_coefs_valid,
    output reg goertzel_coefs_done,
    output reg [31:0] dft_out,
    output reg dft_valid
);

    // Goertzel algorithm variables
    reg [31:0] q_prev [0:NUM_FREQS-1];
    reg [31:0] q_curr [0:NUM_FREQS-1];
    reg [$clog2(NUM_FREQS)-1:0] freq_idx;

    // Shift register and LUT for storing Goertzel coefficients
    reg [COEF_SIZE-1:0] goertzel_coefs_reg [0:NUM_FREQS-1];
    reg [$clog2(NUM_FREQS)-1:0] goertzel_coefs_cnt;

    // Goertzel coefficients loading process
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            goertzel_coefs_cnt <= 0;
            goertzel_coefs_valid <= 0;
            goertzel_coefs_done <= 0;
        end else if (goertzel_coefs_start) begin
            goertzel_coefs_reg[goertzel_coefs_cnt] <= goertzel_coefs;
            goertzel_coefs_valid <= 1;
            if (goertzel_coefs_cnt == NUM_FREQS-1) begin
                goertzel_coefs_done <= 1;
                goertzel_coefs_cnt <= 0;
            end else begin
                goertzel_coefs_cnt <= goertzel_coefs_cnt + 1;
            end
        end else begin
            goertzel_coefs_valid <= 0;
            goertzel_coefs_done <= 0;
        end
    end

    // Goertzel algorithm implementation
    integer j;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            for (j = 0; j < NUM_FREQS; j = j + 1) begin
                q_prev[j] <= 32'h0;
            end
        end
    end

```

```

        q_curr[j] <= 32'h0;
    end
    freq_idx <= 0;
    dft_out <= 32'h0;
    dft_valid <= 1'b0;
end else if (framed_valid) begin
    for (j = 0; j < NUM_FREQS; j = j + 1) begin
        // Create temporary variables
        reg [31:0] q_curr_temp;
        reg [31:0] q_prev_temp;

        // Compute Goertzel algorithm using loaded coefficients
        from LUT
        q_curr_temp = ({24{goertzel_coefs_reg[j][COEF_SIZE-1]}},
goertzel_coefs_reg[j]} * q_prev[j] >>> 15) - q_curr[j] +
{{16{framed_out[15]}}, framed_out};
        q_prev_temp = q_curr[j];

        // Assign the updated values to the arrays
        q_curr[j] <= q_curr_temp;
        q_prev[j] <= q_prev_temp;
    end

    // Increment frequency index
    freq_idx <= freq_idx + 1;

    // Output DFT result when all frequencies are processed
    if (freq_idx == NUM_FREQS-1) begin
        dft_out <= q_curr[freq_idx]*q_curr[freq_idx] +
q_prev[freq_idx]*q_prev[freq_idx] -
({24{goertzel_coefs_reg[freq_idx][COEF_SIZE-1]}},
goertzel_coefs_reg[freq_idx]} * q_curr[freq_idx] >>> 15);
        dft_valid <= 1'b1;
        freq_idx <= 0;
    end else begin
        dft_valid <= 1'b0;
    end
end else begin
    dft_valid <= 1'b0;
end
end
endmodule

`endif

```

In this updated code:

1. The goertzel_coefs_valid signal is declared as an output reg.
2. During the Goertzel coefficients loading process, the goertzel_coefs_valid signal is set to 1 when the goertzel_coefs_start signal is asserted, indicating valid data transfer.

3. The `goertzel_coefs_valid` signal is set to 0 when the loading process is not active or when it has completed (i.e., when `goertzel_coefs_done` is asserted).

With these changes, the `goertzel_coefs_valid` signal correctly indicates the valid data transfer during the loading of Goertzel coefficients into the shift register and LUT.

Copy

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

EB

During the compilation these errors were found "%Warning-WIDTHEXPAND: /openlane/designs/kws/src/goertzel_dft.v:38:36: Operator EQ expects 32 or 9 bits on the LHS, but LHS's VARREF 'goertzel_coefs_cnt' generates 8 bits.

: ... In instance kws.mfcc.dft

```
38 | if (goertzel_coefs_cnt == NUM_FREQS) begin
```

```
| ^~
```

... For warning description see <https://verilator.org/warn/WIDTHEXPAND?v=5.009>

... Use `/* verilator lint_off WIDTHEXPAND */` and `lint_on` around source to disable this message.

%Warning-WIDTHEXPAND: /openlane/designs/kws/src/goertzel_dft.v:80:26: Operator EQ expects 32 or 9 bits on the LHS, but LHS's VARREF 'freq_idx' generates 8 bits.

: ... In instance kws.mfcc.dft

```
80 | if (freq_idx == NUM_FREQS) begin
```

```
| ^~
```

%Warning-UNDRIVEN: /openlane/designs/kws/src/kws.v:36:16: Signal is not driven: 'goertzel_coefs_start'

: ... In instance kws

```
36 | output reg goertzel_coefs_start,".
```

Claude does not have internet access and cannot get up to date info from web links.

To address the compilation warnings and errors, let's update the `goertzel_dft` module and the `kws` module accordingly:

1. Update the `goertzel_dft` module to fix the `WIDTHEXPAND` warnings:

verilog

Copy code

```
`ifndef GOERTZEL_DFT_V
`define GOERTZEL_DFT_V

module goertzel_dft #(
    parameter NUM_FREQS = 256,
    parameter COEF_SIZE = 8
)(
    input wire clk,
    input wire rst_n,
    input wire [15:0] framed_out,
    input wire framed_valid,
    input wire [COEF_SIZE-1:0] goertzel_coefs,
    input wire goertzel_coefs_start,
    output reg goertzel_coefs_valid,
    output reg goertzel_coefs_done,
    output reg [31:0] dft_out,
    output reg dft_valid
);

    // Goertzel algorithm variables
    reg [31:0] q_prev [0:NUM_FREQS-1];
    reg [31:0] q_curr [0:NUM_FREQS-1];
    reg [$clog2(NUM_FREQS)-1:0] freq_idx;

    // Shift register and LUT for storing Goertzel coefficients
    reg [COEF_SIZE-1:0] goertzel_coefs_reg [0:NUM_FREQS-1];
    reg [$clog2(NUM_FREQS)-1:0] goertzel_coefs_cnt;

    // Goertzel coefficients loading process
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            goertzel_coefs_cnt <= 0;
            goertzel_coefs_valid <= 0;
            goertzel_coefs_done <= 0;
        end else if (goertzel_coefs_start) begin
            goertzel_coefs_reg[goertzel_coefs_cnt] <= goertzel_coefs;
            goertzel_coefs_valid <= 1;
            if (goertzel_coefs_cnt == NUM_FREQS[$clog2(NUM_FREQS)-1:0] - 1)
begin
                goertzel_coefs_done <= 1;
                goertzel_coefs_cnt <= 0;
            end else begin
                goertzel_coefs_cnt <= goertzel_coefs_cnt + 1;
            end
        end else begin
            goertzel_coefs_valid <= 0;
            goertzel_coefs_done <= 0;
        end
    end

    // Goertzel algorithm implementation
    integer j;
```

```

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        for (j = 0; j < NUM_FREQS; j = j + 1) begin
            q_prev[j] <= 32'h0;
            q_curr[j] <= 32'h0;
        end
        freq_idx <= 0;
        dft_out <= 32'h0;
        dft_valid <= 1'b0;
    end else if (framed_valid) begin
        for (j = 0; j < NUM_FREQS; j = j + 1) begin
            // Create temporary variables
            reg [31:0] q_curr_temp;
            reg [31:0] q_prev_temp;

            // Compute Goertzel algorithm using loaded coefficients
            from LUT
            q_curr_temp = ({24{goertzel_coefs_reg[j][COEF_SIZE-1]}},
goertzel_coefs_reg[j]} * q_prev[j] >>> 15) - q_curr[j] +
{{16{framed_out[15]}}, framed_out};
            q_prev_temp = q_curr[j];

            // Assign the updated values to the arrays
            q_curr[j] <= q_curr_temp;
            q_prev[j] <= q_prev_temp;
        end

        // Increment frequency index
        freq_idx <= freq_idx + 1;

        // Output DFT result when all frequencies are processed
        if (freq_idx == NUM_FREQS[$clog2(NUM_FREQS)-1:0] - 1) begin
            dft_out <= q_curr[freq_idx]*q_curr[freq_idx] +
q_prev[freq_idx]*q_prev[freq_idx] -
({24{goertzel_coefs_reg[freq_idx][COEF_SIZE-1]}},
goertzel_coefs_reg[freq_idx]} * q_curr[freq_idx] >>> 15);
            dft_valid <= 1'b1;
            freq_idx <= 0;
        end else begin
            dft_valid <= 1'b0;
        end
    end else begin
        dft_valid <= 1'b0;
    end
end
endmodule

`endif

```

In this updated code:

- The comparisons with `NUM_FREQS` are updated to use the appropriate bit width by indexing `NUM_FREQS` with `[$clog2(NUM_FREQS)-1:0]` to avoid the `WIDTHEXPAND` warnings.

2. Update the `kws` module to fix the `UNDRIVEN` warning:

verilog

Copy code

```
module kws #(
    // ... (other parameters)
)(
    // ... (other ports)
    input wire [7:0] goertzel_coefs,
    output reg goertzel_coefs_start,
    output wire goertzel_coefs_valid,
    output wire goertzel_coefs_done,
    // ... (other ports)
);

// ... (other code)

// Control logic for Goertzel coefficients transfer
localparam IDLE = 2'b00;
localparam START_TRANSFER = 2'b01;
localparam WAIT_TRANSFER = 2'b10;

reg [1:0] state;

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= IDLE;
        goertzel_coefs_start <= 0;
    end else begin
        case (state)
            IDLE: begin
                // Initiate the transfer
                state <= START_TRANSFER;
            end
            START_TRANSFER: begin
                // Set the start signal
                goertzel_coefs_start <= 1;
                state <= WAIT_TRANSFER;
            end
            WAIT_TRANSFER: begin
                // Clear the start signal
                goertzel_coefs_start <= 0;
                // Wait for the transfer to complete
                if (goertzel_coefs_done) begin
                    state <= IDLE;
                end
            end
        end
    end
end
```

```

        endcase
    end
end

// ... (other code)

endmodule

```

In this updated code:

- The missing control logic for the `goertzel_coefs_start` signal is added using a state machine to drive the signal properly.

With these updates, the `WIDTHEXPAND` warnings in the `goertzel_dft` module and the `UNDRIVEN` warning in the `kws` module should be resolved.

Copy

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

EB

The state machine should be inside the `mfcc_accelerator` not the `kws` top design, right?

You are absolutely right! I apologize for my mistake. The state machine for controlling the Goertzel coefficients transfer should indeed be inside the `mfcc_accelerator` module, not the `kws` top module.

Here's the updated `mfcc_accelerator` module with the state machine:

verilog

Copy code

```

module mfcc_accelerator #(
    parameter MFCC_FEATURES = 40,
    parameter ACTIV_BITS = 8
) (
    // ... (other ports)
    input wire [7:0] goertzel_coefs,
    output reg goertzel_coefs_start,
    output wire goertzel_coefs_valid,
    output wire goertzel_coefs_done,
    // ... (other ports)
);

// ... (other code)

// Control logic for Goertzel coefficients transfer
localparam IDLE = 2'b00;
localparam START_TRANSFER = 2'b01;
localparam WAIT_TRANSFER = 2'b10;

reg [1:0] state;

```

```

always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state <= IDLE;
        goertzel_coefs_start <= 0;
    end else begin
        case (state)
            IDLE: begin
                // Initiate the transfer
                state <= START_TRANSFER;
            end
            START_TRANSFER: begin
                // Set the start signal
                goertzel_coefs_start <= 1;
                state <= WAIT_TRANSFER;
            end
            WAIT_TRANSFER: begin
                // Clear the start signal
                goertzel_coefs_start <= 0;
                // Wait for the transfer to complete
                if (goertzel_coefs_done) begin
                    state <= IDLE;
                end
            end
        endcase
    end
end

// Instantiate the modified goertzel_dft module
goertzel_dft dft (
    .clk(clk),
    .rst_n(rst_n),
    .framed_out(framed_out),
    .framed_valid(framed_valid),
    .goertzel_coefs(goertzel_coefs),
    .goertzel_coefs_start(goertzel_coefs_start),
    .goertzel_coefs_valid(goertzel_coefs_valid),
    .goertzel_coefs_done(goertzel_coefs_done),
    .dft_out(dft_out),
    .dft_valid(dft_valid)
);

// ... (other code)

endmodule

```

And here's the updated `kws` top module without the state machine:

verilog

Copy code

```

module kws #(
    // ... (other parameters)
)(

```

```

// ... (other ports)
input wire [7:0] goertzel_coefs,
output wire goertzel_coefs_valid,
output wire goertzel_coefs_done,
// ... (other ports)
);

// ... (other code)

// MFCC module instantiation
mfcc_accelerator #(
    .MFCC_FEATURES(MFCC_FEATURES),
    .ACTIV_BITS(ACTIV_BITS)
) mfcc (
    .clk(clk),
    .rst_n(rst_n),
    .audio_in(audio_in),
    .audio_valid(audio_valid),
    .mfcc_out(mfcc_out),
    .mfcc_valid(mfcc_valid),
    .frame_size(frame_size),
    .frame_overlap(frame_overlap),
    .num_mfcc_coeffs(num_mfcc_coeffs),
    .goertzel_coefs(goertzel_coefs),
    .goertzel_coefs_valid(goertzel_coefs_valid),
    .goertzel_coefs_done(goertzel_coefs_done)
);

// ... (other code)

endmodule

```

Now, the state machine for controlling the Goertzel coefficients transfer is correctly placed inside the `mfcc_accelerator` module, and the `kws` top module simply instantiates the `mfcc_accelerator` module without the state machine logic.

Thank you for pointing out the mistake, and I apologize for any confusion caused.

Copy

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

EB

Top `kws` design is missing pins. Here are the errors "%Warning-PINMISSING: /openlane/designs/kws/src/kws.v:78:7: Cell has missing pin: 'goertzel_coefs_start'"

```
78 | ) mfcc (
```

```
| ^~~~
```

... For warning description see <https://verilator.org/warn/PINMISSING?v=5.009>

... Use `/* verilator lint_off PINMISSING */` and `lint_on` around source to disable this message.

%Error-ASSIGNIN: /openlane/designs/kws/src/mfcc_accelerator.v:107:13: Assigning to input/const variable: 'goertzel_coefs_start'

```
107 | goertzel_coefs_start <= 0;
```

```
| ^~~~~~
```

%Error-ASSIGNIN: /openlane/designs/kws/src/mfcc_accelerator.v:116:21: Assigning to input/const variable: 'goertzel_coefs_start'

```
116 | goertzel_coefs_start <= 1;
```

```
| ^~~~~~
```

%Error-ASSIGNIN: /openlane/designs/kws/src/mfcc_accelerator.v:121:21: Assigning to input/const variable: 'goertzel_coefs_start'

```
121 | goertzel_coefs_start <= 0;
```

```
| ^~~~~~"
```

Claude does not have internet access and cannot get up to date info from web links.

Apologies for the oversight. Let's update the `kws` top module and the `mfcc_accelerator` module to fix the missing pin and the `ASSIGNIN` errors.

Here's the updated `kws` top module with the missing `goertzel_coefs_start` pin:

verilog

Copy code

```
module kws #(
    // ... (other parameters)
)()
    // ... (other ports)
    input wire [7:0] goertzel_coefs,
    output wire goertzel_coefs_start,
    output wire goertzel_coefs_valid,
    output wire goertzel_coefs_done,
    // ... (other ports)
);

    // ... (other code)

    // MFCC module instantiation
    mfcc_accelerator #(
        .MFCC_FEATURES(MFCC_FEATURES),
        .ACTIV_BITS(ACTIV_BITS)
```



```

    ) mfcc (
        .clk(clk),
        .rst_n(rst_n),
        .audio_in(audio_in),
        .audio_valid(audio_valid),
        .mfcc_out(mfcc_out),
        .mfcc_valid(mfcc_valid),
        .frame_size(frame_size),
        .frame_overlap(frame_overlap),
        .num_mfcc_coefs(num_mfcc_coefs),
        .goertzel_coefs(goertzel_coefs),
        .goertzel_coefs_start(goertzel_coefs_start),
        .goertzel_coefs_valid(goertzel_coefs_valid),
        .goertzel_coefs_done(goertzel_coefs_done)
    );

    // ... (other code)

endmodule

```

And here's the updated `mfcc_accelerator` module to fix the ASSIGNIN errors:

verilog

Copy code

```

module mfcc_accelerator #(
    parameter MFCC_FEATURES = 40,
    parameter ACTIV_BITS = 8
) (
    // ... (other ports)
    input wire [7:0] goertzel_coefs,
    output reg goertzel_coefs_start,
    output wire goertzel_coefs_valid,
    output wire goertzel_coefs_done,
    // ... (other ports)
);

    // ... (other code)

    // Control logic for Goertzel coefficients transfer
    localparam IDLE = 2'b00;
    localparam START_TRANSFER = 2'b01;
    localparam WAIT_TRANSFER = 2'b10;

    reg [1:0] state;
    reg goertzel_coefs_start_reg;

    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state <= IDLE;
            goertzel_coefs_start_reg <= 0;
        end else begin
            case (state)

```

```

        IDLE: begin
            // Initiate the transfer
            state <= START_TRANSFER;
        end
        START_TRANSFER: begin
            // Set the start signal
            goertzel_coefs_start_reg <= 1;
            state <= WAIT_TRANSFER;
        end
        WAIT_TRANSFER: begin
            // Clear the start signal
            goertzel_coefs_start_reg <= 0;
            // Wait for the transfer to complete
            if (goertzel_coefs_done) begin
                state <= IDLE;
            end
        end
    endcase
end
end

assign goertzel_coefs_start = goertzel_coefs_start_reg;

// Instantiate the modified goertzel_dft module
goertzel_dft dft (
    .clk(clk),
    .rst_n(rst_n),
    .framed_out(framed_out),
    .framed_valid(framed_valid),
    .goertzel_coefs(goertzel_coefs),
    .goertzel_coefs_start(goertzel_coefs_start),
    .goertzel_coefs_valid(goertzel_coefs_valid),
    .goertzel_coefs_done(goertzel_coefs_done),
    .dft_out(dft_out),
    .dft_valid(dft_valid)
);

// ... (other code)

endmodule

```

In the updated `kws_top` module, the `goertzel_coefs_start` pin is added to the port list and connected to the `mfcc_accelerator` module instantiation.

In the updated `mfcc_accelerator` module:

- A new register `goertzel_coefs_start_reg` is introduced to drive the `goertzel_coefs_start` signal.
- The state machine logic is updated to assign values to `goertzel_coefs_start_reg` instead of directly assigning to `goertzel_coefs_start`.
- The `goertzel_coefs_start` signal is assigned the value of `goertzel_coefs_start_reg` using a continuous assignment statement.

These changes should resolve the missing pin error in the `kws` top module and the `ASSIGNIN` errors in the `mfcc_accelerator` module.

Copy

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

EB

Synthesis found these warnings "Warning: Wire `mfcc_accelerator.mel_fbank_out` [31] is used but has no driver.". Based on the CNN-KWS model, also known as the "Hello Edge" model that was introduced in the paper "Hello Edge: Keyword Spotting on Microcontrollers" by Zhang et al. (2017) and the top module `kws`, what are the requirements for the `mel_filterbank` module. write a prompt for it.

Based on the CNN-KWS model, also known as the "Hello Edge" model introduced in the paper "Hello Edge: Keyword Spotting on Microcontrollers" by Zhang et al. (2017), and the top module `kws`, the requirements for the `mel_filterbank` module are as follows:

1. The `mel_filterbank` module should perform mel-scale filterbank computation on the output of the Discrete Fourier Transform (DFT) module.
2. The module should have the following inputs:
 - o `clk`: Clock signal
 - o `rst_n`: Active-low reset signal
 - o `dft_out`: 32-bit DFT output from the `goertzel_dft` module
 - o `dft_valid`: Valid signal for the DFT output
 - o `num_mel_filters`: Number of mel-scale filters to apply (e.g., 40)
 - o `num_dft_points`: Number of DFT points (e.g., 256)
3. The module should have the following outputs:
 - o `mel_fbank_out`: 32-bit mel-scale filterbank output
 - o `mel_fbank_valid`: Valid signal for the mel-scale filterbank output
4. The module should use a set of triangular mel-scale filters to compute the filterbank energies. The filter coefficients should be stored in a lookup table (LUT) or memory.
5. The mel-scale filterbank computation should be performed by multiplying the DFT output with the corresponding filter coefficients and accumulating the results for each mel-scale filter.
6. The mel-scale filter coefficients should be precomputed and stored in the LUT or memory. The coefficients should be quantized to a fixed-point representation (e.g., Q15 format) to optimize hardware resources.
7. The module should handle the necessary bit widths and arithmetic operations to ensure proper computation of the mel-scale filterbank energies.
8. The `mel_fbank_valid` signal should be asserted when the mel-scale filterbank output is valid, which occurs after the DFT output has been processed by all the mel-scale filters.
9. The module should be designed to minimize resource usage and latency, considering the constraints of the target hardware platform.
10. The module should handle any potential edge cases or exceptional conditions gracefully, such as invalid input signals or out-of-range values.

Based on these requirements, please write a Verilog module for `mel_filterbank` that efficiently computes the mel-scale filterbank energies from the DFT output. The module should be optimized for resource usage and latency, and it should be compatible with the

CNN-KWS model and the top module `kws`. Consider the specific details provided in the paper, such as the number of mel-scale filters and DFT points, and adapt the module accordingly.

Please also ensure that the module addresses the synthesis warning related to the undriven wire `mel_fbank_out[31]`. Make sure that all the bits of the `mel_fbank_out` output are properly driven based on the mel-scale filterbank computation.

Copy
EB

1. The `mel_filterbank` module should perform mel-scale filterbank computation on the output of the Discrete Fourier Transform (DFT) module.
2. The module should have the following inputs:
 - o `clk`: Clock signal
 - o `rst_n`: Active-low reset signal
 - o `dft_out`: 32-bit DFT output from the `goertzel_dft` module
 - o `dft_valid`: Valid signal for the DFT output
 - o `num_mel_filters`: Number of mel-scale filters to apply (e.g., 40)
 - o `num_dft_points`: Number of DFT points (e.g., 256)
3. The module should have the following outputs:
 - o `mel_fbank_out`: 32-bit mel-scale filterbank output
 - o `mel_fbank_valid`: Valid signal for the mel-scale filterbank output
4. The module should use a set of triangular mel-scale filters to compute the filterbank energies. The filter coefficients should be stored in a lookup table (LUT) or memory.
5. The mel-scale filterbank computation should be performed by multiplying the DFT output with the corresponding filter coefficients and accumulating the results for each mel-scale filter.
6. The mel-scale filter coefficients should be precomputed and stored in the LUT or memory. The coefficients should be quantized to a fixed-point representation (e.g., Q15 format) to optimize hardware resources.
7. The module should handle the necessary bit widths and arithmetic operations to ensure proper computation of the mel-scale filterbank energies.
8. The `mel_fbank_valid` signal should be asserted when the mel-scale filterbank output is valid, which occurs after the DFT output has been processed by all the mel-scale filters.
9. The module should be designed to minimize resource usage and latency, considering the constraints of the target hardware platform.
10. The module should handle any potential edge cases or exceptional conditions gracefully, such as invalid input signals or out-of-range values.

Based on these requirements, please write a Verilog module for `mel_filterbank` that efficiently computes the mel-scale filterbank energies from the DFT output. The module should be optimized for resource usage and latency, and it should be compatible with the CNN-KWS model and the top module `kws`. Consider the specific details provided in the paper, such as the number of mel-scale filters and DFT points, and adapt the module accordingly.

Please also ensure that the module addresses the synthesis warning related to the undriven wire `mel_fbank_out[31]`. Make sure that all the bits of the `mel_fbank_out` output are properly driven based on the mel-scale filterbank computation.

Here's the Verilog code for the `mel_filterbank` module based on the given requirements:

verilog

Copy code

```
`ifndef MEL_FILTERBANK_V
`define MEL_FILTERBANK_V

module mel_filterbank #(
    parameter NUM_MEL_FILTERS = 40,
    parameter NUM_DFT_POINTS = 256,
    parameter COEF_WIDTH = 16,
    parameter ACCUMULATOR_WIDTH = 32
) (
    input wire clk,
    input wire rst_n,
    input wire [31:0] dft_out,
    input wire dft_valid,
    output reg [ACCUMULATOR_WIDTH-1:0] mel_fbank_out,
    output reg mel_fbank_valid
);

    // Mel-scale filter coefficients (stored in LUT or memory)
    reg [COEF_WIDTH-1:0] mel_filter_coefs [0:NUM_MEL_FILTERS-1][0:NUM_DFT_POINTS-1];

    // Registers for accumulating filterbank energies
    reg [ACCUMULATOR_WIDTH-1:0] mel_accumulators [0:NUM_MEL_FILTERS-1];

    // Counters for iterating over mel-scale filters and DFT points
    reg [$clog2(NUM_MEL_FILTERS)-1:0] mel_filter_cnt;
    reg [$clog2(NUM_DFT_POINTS)-1:0] dft_point_cnt;

    // Initialize mel-scale filter coefficients (precomputed)
    initial begin
        // Assuming coefficients are stored in an external file or memory
        // Replace this with the actual coefficient initialization code
        $readmemh("mel_filter_coefs.mem", mel_filter_coefs);
    end

    // Mel-scale filterbank computation
    integer i;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            mel_filter_cnt <= 0;
            dft_point_cnt <= 0;
            mel_fbank_out <= 0;
            mel_fbank_valid <= 0;
        end
    end
endmodule
```

```

        for (i = 0; i < NUM_MEL_FILTERS; i = i + 1) begin
            mel_accumulators[i] <= 0;
        end
    end else if (dft_valid) begin
        // Multiply DFT output with the corresponding filter
        coefficient
        mel_accumulators[mel_filter_cnt] <=
mel_accumulators[mel_filter_cnt] +
            (dft_out *
mel_filter_coefs[mel_filter_cnt][dft_point_cnt]);

        // Increment DFT point counter
        if (dft_point_cnt == NUM_DFT_POINTS - 1) begin
            dft_point_cnt <= 0;

            // Increment mel-scale filter counter
            if (mel_filter_cnt == NUM_MEL_FILTERS - 1) begin
                mel_filter_cnt <= 0;

                // Output the accumulated filterbank energies
                mel_fbank_out <= mel_accumulators[NUM_MEL_FILTERS - 1];
                mel_fbank_valid <= 1;

                // Reset the accumulators for the next frame
                for (i = 0; i < NUM_MEL_FILTERS; i = i + 1) begin
                    mel_accumulators[i] <= 0;
                end
            end else begin
                mel_filter_cnt <= mel_filter_cnt + 1;
                mel_fbank_valid <= 0;
            end
        end else begin
            dft_point_cnt <= dft_point_cnt + 1;
            mel_fbank_valid <= 0;
        end
    end else begin
        mel_fbank_valid <= 0;
    end
end
endmodule

`endif

```

Explanation:

1. The `mel_filterbank` module is parameterized with `NUM_MEL_FILTERS` (number of mel-scale filters), `NUM_DFT_POINTS` (number of DFT points), `COEF_WIDTH` (width of the filter coefficients), and `ACCUMULATOR_WIDTH` (width of the accumulator for filterbank energies).
2. The module has the required inputs: `clk`, `rst_n`, `dft_out`, `dft_valid`.
3. The module has the required outputs: `mel_fbank_out` and `mel_fbank_valid`.

4. The mel-scale filter coefficients are stored in a 2D register array `mel_filter_coefs`. The coefficients are assumed to be precomputed and stored in an external file or memory. The actual initialization code needs to be replaced based on the specific implementation.
5. The mel-scale filterbank computation is performed by multiplying the DFT output with the corresponding filter coefficients and accumulating the results for each mel-scale filter. This is done using the `mel_accumulators` register array.
6. The filter coefficients are stored in the `mel_filter_coefs` array, which is initialized with precomputed values. The coefficients are assumed to be in a fixed-point representation (e.g., Q15 format) to optimize hardware resources.
7. The module handles the necessary bit widths and arithmetic operations for the mel-scale filterbank computation. The `COEF_WIDTH` and `ACCUMULATOR_WIDTH` parameters can be adjusted based on the required precision and range.
8. The `mel_fbank_valid` signal is asserted when the mel-scale filterbank output is valid, which occurs after the DFT output has been processed by all the mel-scale filters.
9. The module is designed to minimize resource usage by using fixed-point arithmetic and storing the filter coefficients in a memory-efficient manner. The latency is determined by the number of mel-scale filters and DFT points.
10. The module handles the case when `rst_n` is asserted by resetting the internal counters, accumulators, and outputs.

The module addresses the synthesis warning related to the undriven wire `mel_fbank_out[31]` by properly assigning the accumulated filterbank energies to the `mel_fbank_out` output register.

Note: The actual initialization of the mel-scale filter coefficients (`mel_filter_coefs`) needs to be replaced with the specific code to load the precomputed coefficients from an external file or memory based on the implementation requirements.

Copy

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

EB

This code will not be implemented in an FPGA but rather compiled to an ASIC. Instructions like `"$readmemh("mel_filter_coefs.mem", mel_filter_coefs);"` are not preferred. Instead I prefer to create a lut.

I apologize for the confusion. Here's the updated Verilog code for the `mel_filterbank` module with the mel-scale filter coefficients stored in a lookup table (LUT) instead of using `$readmemh`:

verilog

Copy code

```
`ifndef MEL_FILTERBANK_V
`define MEL_FILTERBANK_V

module mel_filterbank #(
    parameter NUM_MEL_FILTERS = 40,
```

```

parameter NUM_DFT_POINTS = 256,
parameter COEF_WIDTH = 16,
parameter ACCUMULATOR_WIDTH = 32
) (
    input wire clk,
    input wire rst_n,
    input wire [31:0] dft_out,
    input wire dft_valid,
    output reg [ACCUMULATOR_WIDTH-1:0] mel_fbank_out,
    output reg mel_fbank_valid
);

    // Mel-scale filter coefficients (stored in LUT)
    reg [COEF_WIDTH-1:0] mel_filter_coefs [0:NUM_MEL_FILTERS-1][0:NUM_DFT_POINTS-1];

    // Registers for accumulating filterbank energies
    reg [ACCUMULATOR_WIDTH-1:0] mel_accumulators [0:NUM_MEL_FILTERS-1];

    // Counters for iterating over mel-scale filters and DFT points
    reg [$clog2(NUM_MEL_FILTERS)-1:0] mel_filter_cnt;
    reg [$clog2(NUM_DFT_POINTS)-1:0] dft_point_cnt;

    // Initialize mel-scale filter coefficients (precomputed)
    initial begin
        // Assuming coefficients are precomputed and stored in the LUT
        // Replace this with the actual coefficient values
        mel_filter_coefs[0][0] = 16'h1234;
        mel_filter_coefs[0][1] = 16'h5678;
        // ...
        mel_filter_coefs[NUM_MEL_FILTERS-1][NUM_DFT_POINTS-2] = 16'hABCD;
        mel_filter_coefs[NUM_MEL_FILTERS-1][NUM_DFT_POINTS-1] = 16'hEF01;
    end

    // Mel-scale filterbank computation
    integer i;
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            mel_filter_cnt <= 0;
            dft_point_cnt <= 0;
            mel_fbank_out <= 0;
            mel_fbank_valid <= 0;
            for (i = 0; i < NUM_MEL_FILTERS; i = i + 1) begin
                mel_accumulators[i] <= 0;
            end
        end else if (dft_valid) begin
            // Multiply DFT output with the corresponding filter
            // coefficient
            mel_accumulators[mel_filter_cnt] <=
mel_accumulators[mel_filter_cnt] +
                (dft_out *
mel_filter_coefs[mel_filter_cnt][dft_point_cnt]);

```



```

        // Increment DFT point counter
        if (dft_point_cnt == NUM_DFT_POINTS - 1) begin
            dft_point_cnt <= 0;

            // Increment mel-scale filter counter
            if (mel_filter_cnt == NUM_MEL_FILTERS - 1) begin
                mel_filter_cnt <= 0;

                // Output the accumulated filterbank energies
                mel_fbank_out <= mel_accumulators[NUM_MEL_FILTERS - 1];
                mel_fbank_valid <= 1;

                // Reset the accumulators for the next frame
                for (i = 0; i < NUM_MEL_FILTERS; i = i + 1) begin
                    mel_accumulators[i] <= 0;
                end
            end else begin
                mel_filter_cnt <= mel_filter_cnt + 1;
                mel_fbank_valid <= 0;
            end
        end else begin
            dft_point_cnt <= dft_point_cnt + 1;
            mel_fbank_valid <= 0;
        end
    end else begin
        mel_fbank_valid <= 0;
    end
end
endmodule

`endif

```

In this updated code:

1. The mel-scale filter coefficients are stored in a lookup table (LUT) represented by the 2D register array `mel_filter_coefs`.
2. The initialization of the filter coefficients is done using an `initial` block, where the actual coefficient values need to be provided. Replace the placeholder values (16'h1234, 16'h5678, etc.) with the precomputed coefficient values for each filter and DFT point.
3. The rest of the module remains the same, with the mel-scale filterbank computation being performed by multiplying the DFT output with the corresponding filter coefficients from the LUT and accumulating the results for each mel-scale filter.

By storing the mel-scale filter coefficients in a LUT, the module can be synthesized and compiled to an ASIC without relying on external files or memory initialization.

Remember to provide the actual precomputed coefficient values in the `initial` block based on your specific implementation requirements.

Copy

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

EB

create a python script to initialize mel-scale filter coefficients

Certainly! Here's a Python script that generates and initializes the mel-scale filter coefficients:

python

Copy code

```
import numpy as np
import math

# Mel-scale filterbank parameters
NUM_MEL_FILTERS = 40
NUM_DFT_POINTS = 256
SAMPLE_RATE = 16000 # Assuming a sample rate of 16 kHz
LOW_FREQ_MEL = 0
HIGH_FREQ_MEL = 2595 * np.log10(1 + (SAMPLE_RATE / 2) / 700) # Convert Hz to Mel

# Generate mel-scale filter center frequencies
mel_filter_centers = np.linspace(LOW_FREQ_MEL, HIGH_FREQ_MEL,
NUM_MEL_FILTERS + 2)

# Convert mel-scale filter center frequencies to linear scale
linear_filter_centers = 700 * (10 ** (mel_filter_centers / 2595) - 1)

# Compute the DFT bin frequencies
dft_bin_freqs = np.arange(NUM_DFT_POINTS) * (SAMPLE_RATE / 2) /
(NUM_DFT_POINTS - 1)

# Initialize the mel-scale filter coefficients
mel_filter_coefs = np.zeros((NUM_MEL_FILTERS, NUM_DFT_POINTS))

# Compute the mel-scale filter coefficients
for m in range(1, NUM_MEL_FILTERS + 1):
    left_freq = linear_filter_centers[m - 1]
    center_freq = linear_filter_centers[m]
    right_freq = linear_filter_centers[m + 1]

    for k in range(NUM_DFT_POINTS):
        if dft_bin_freqs[k] >= left_freq and dft_bin_freqs[k] <=
center_freq:
            mel_filter_coefs[m - 1, k] = (dft_bin_freqs[k] - left_freq) /
(center_freq - left_freq)
        elif dft_bin_freqs[k] > center_freq and dft_bin_freqs[k] <=
right_freq:
            mel_filter_coefs[m - 1, k] = (right_freq - dft_bin_freqs[k]) /
(right_freq - center_freq)
```

```

# Scale the filter coefficients to a fixed-point representation (Q15
format)
Q15_SCALE = 2 ** 15
mel_filter_coefs_fixed = np.round(mel_filter_coefs * Q15_SCALE).astype(int)

# Generate Verilog code for the mel-scale filter coefficients
verilog_code = ""
for m in range(NUM_MEL_FILTERS):
    for k in range(NUM_DFT_POINTS):
        verilog_code += f"mel_filter_coefs[{m}][{k}] =
16'h{mel_filter_coefs_fixed[m, k]:04X};\n"

# Print the Verilog code
print(verilog_code)

```

This Python script does the following:

1. It defines the necessary parameters for the mel-scale filterbank, such as the number of mel-scale filters (NUM_MEL_FILTERS), the number of DFT points (NUM_DFT_POINTS), and the sample rate (SAMPLE_RATE).
2. It generates the mel-scale filter center frequencies using `np.linspace()` and converts them to the linear scale using the formula $700 * (10 ** (\text{mel_freq} / 2595) - 1)$.
3. It computes the DFT bin frequencies based on the sample rate and the number of DFT points.
4. It initializes the `mel_filter_coefs` array to store the mel-scale filter coefficients.
5. It computes the mel-scale filter coefficients using triangular filters. For each mel-scale filter, it determines the left, center, and right frequencies and computes the filter coefficients using linear interpolation.
6. It scales the filter coefficients to a fixed-point representation (Q15 format) by multiplying them by 2^{15} and rounding to the nearest integer.
7. It generates the Verilog code for initializing the `mel_filter_coefs` array in the lookup table (LUT) format. Each coefficient is represented as a 16-bit hexadecimal value.
8. Finally, it prints the generated Verilog code.

To use this script, you can run it in a Python environment. The script will generate the Verilog code for initializing the mel-scale filter coefficients. You can then copy and paste the generated code into the `initial` block of the `mel_filterbank` module, replacing the placeholder values.

Note: Make sure you have the NumPy library installed (`import numpy as np`) before running the script.

Copy

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

arning: Wire

\$paramod\$35c923e6eddd4ad833178706e63a81a2798d4b07\mfcc_accelerator.\mel_fbank_out [31] is used but has no driver. Encountered check error: Warning: Wire
\$paramod\$35c923e6eddd4ad833178706

pasted

EB

Apologies for the oversight. It seems that the `mel_fbank_out` and `mel_fbank_valid` signals are not being properly driven in the `mel_filterbank` module. Let's update the module to ensure that all bits of `mel_fbank_out` are driven and the `mel_fbank_valid` signal is correctly assigned.

Here's the updated Verilog code for the `mel_filterbank` module:

verilog

Copy code

```
`ifndef MEL_FILTERBANK_V
`define MEL_FILTERBANK_V

module mel_filterbank #(
    parameter NUM_MEL_FILTERS = 40,
    parameter NUM_DFT_POINTS = 256,
    parameter COEF_WIDTH = 16,
    parameter ACCUMULATOR_WIDTH = 32
) (
    input wire clk,
    input wire rst_n,
    input wire [31:0] dft_out,
    input wire dft_valid,
    output reg [ACCUMULATOR_WIDTH-1:0] mel_fbank_out,
    output reg mel_fbank_valid
);

    // Mel-scale filter coefficients (stored in LUT)
    reg [COEF_WIDTH-1:0] mel_filter_coefs [0:NUM_MEL_FILTERS-1][0:NUM_DFT_POINTS-1];

    // Registers for accumulating filterbank energies
    reg [ACCUMULATOR_WIDTH-1:0] mel_accumulators [0:NUM_MEL_FILTERS-1];

    // Counters for iterating over mel-scale filters and DFT points
    reg [$clog2(NUM_MEL_FILTERS)-1:0] mel_filter_cnt;
    reg [$clog2(NUM_DFT_POINTS)-1:0] dft_point_cnt;

    // Initialize mel-scale filter coefficients (precomputed)
    initial begin
        // Assuming coefficients are precomputed and stored in the LUT
        // Replace this with the actual coefficient values generated by the
        Python script
        mel_filter_coefs[0][0] = 16'h1234;
        mel_filter_coefs[0][1] = 16'h5678;
        // ...
        mel_filter_coefs[NUM_MEL_FILTERS-1][NUM_DFT_POINTS-2] = 16'hABCD;
        mel_filter_coefs[NUM_MEL_FILTERS-1][NUM_DFT_POINTS-1] = 16'hEF01;
    end
end
```

```

// Mel-scale filterbank computation
integer i;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        mel_filter_cnt <= 0;
        dft_point_cnt <= 0;
        mel_fbank_out <= 0;
        mel_fbank_valid <= 0;
        for (i = 0; i < NUM_MEL_FILTERS; i = i + 1) begin
            mel_accumulators[i] <= 0;
        end
    end else if (dft_valid) begin
        // Multiply DFT output with the corresponding filter
        // coefficient
        mel_accumulators[mel_filter_cnt] <=
mel_accumulators[mel_filter_cnt] +
            (dft_out *
mel_filter_coefs[mel_filter_cnt][dft_point_cnt]);

        // Increment DFT point counter
        if (dft_point_cnt == NUM_DFT_POINTS - 1) begin
            dft_point_cnt <= 0;

            // Increment mel-scale filter counter
            if (mel_filter_cnt == NUM_MEL_FILTERS - 1) begin
                mel_filter_cnt <= 0;

                // Output the accumulated filterbank energies
                mel_fbank_out <= mel_accumulators[NUM_MEL_FILTERS - 1];
                mel_fbank_valid <= 1;

                // Reset the accumulators for the next frame
                for (i = 0; i < NUM_MEL_FILTERS; i = i + 1) begin
                    mel_accumulators[i] <= 0;
                end
            end else begin
                mel_filter_cnt <= mel_filter_cnt + 1;
                mel_fbank_out <= mel_accumulators[mel_filter_cnt];
                mel_fbank_valid <= 0;
            end
        end else begin
            dft_point_cnt <= dft_point_cnt + 1;
            mel_fbank_out <= mel_accumulators[mel_filter_cnt];
            mel_fbank_valid <= 0;
        end
    end else begin
        mel_fbank_out <= 0;
        mel_fbank_valid <= 0;
    end
end
endmodule

```

```
`endif
```

In this updated code:

1. The `mel_fbank_out` signal is assigned the value of `mel_accumulators[mel_filter_cnt]` in each iteration of the mel-scale filterbank computation. This ensures that all bits of `mel_fbank_out` are driven based on the current mel-scale filter being processed.
2. When the computation for all mel-scale filters is complete (i.e., `mel_filter_cnt` reaches `NUM_MEL_FILTERS - 1`), the `mel_fbank_out` signal is assigned the final accumulated value from `mel_accumulators[NUM_MEL_FILTERS - 1]`, and the `mel_fbank_valid` signal is set to 1 to indicate valid output.
3. In the `else` conditions (when `dft_valid` is not asserted or when the computation is not complete), the `mel_fbank_out` signal is assigned 0 to ensure that all bits are driven, and the `mel_fbank_valid` signal is set to 0.

With these changes, the `mel_fbank_out` signal should have all its bits properly driven, and the `mel_fbank_valid` signal should be correctly assigned based on the computation status.

Remember to replace the placeholder coefficient values in the `initial` block with the actual values generated by the Python script.

Copy

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

arning: Wire

\$paramod\$35c923e6eddd4ad833178706e63a81a2798d4b07\mfcc_accelerator.\mel_fbank_out [31] is used but has no driver. Encountered check error: Warning: Wire
\$paramod\$35c923e6eddd4ad833178706

pasted

EB

I apologize for the confusion. It seems that the previous changes did not fully resolve the issues with the `mel_fbank_out` and `mel_fbank_valid` signals. Let's make further modifications to ensure that all bits of `mel_fbank_out` are properly driven and `mel_fbank_valid` is correctly assigned.

Here's the updated Verilog code for the `mel_filterbank` module:

verilog

Copy code

```
`ifndef MEL_FILTERBANK_V
`define MEL_FILTERBANK_V

module mel_filterbank #(
    parameter NUM_MEL_FILTERS = 40,
    parameter NUM_DFT_POINTS = 256,
    parameter COEF_WIDTH = 16,
    parameter ACCUMULATOR_WIDTH = 32
) (
```

```

input wire clk,
input wire rst_n,
input wire [31:0] dft_out,
input wire dft_valid,
output reg [ACCUMULATOR_WIDTH-1:0] mel_fbank_out,
output reg mel_fbank_valid
);

// Mel-scale filter coefficients (stored in LUT)
reg [COEF_WIDTH-1:0] mel_filter_coefs [0:NUM_MEL_FILTERS-
1][0:NUM_DFT_POINTS-1];

// Registers for accumulating filterbank energies
reg [ACCUMULATOR_WIDTH-1:0] mel_accumulators [0:NUM_MEL_FILTERS-1];

// Counters for iterating over mel-scale filters and DFT points
reg [$clog2(NUM_MEL_FILTERS)-1:0] mel_filter_cnt;
reg [$clog2(NUM_DFT_POINTS)-1:0] dft_point_cnt;

// Initialize mel-scale filter coefficients (precomputed)
initial begin
    // Assuming coefficients are precomputed and stored in the LUT
    // Replace this with the actual coefficient values generated by the
Python script
    mel_filter_coefs[0][0] = 16'h1234;
    mel_filter_coefs[0][1] = 16'h5678;
    // ...
    mel_filter_coefs[NUM_MEL_FILTERS-1][NUM_DFT_POINTS-2] = 16'hABCD;
    mel_filter_coefs[NUM_MEL_FILTERS-1][NUM_DFT_POINTS-1] = 16'hEF01;
end

// Mel-scale filterbank computation
integer i;
always @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        mel_filter_cnt <= 0;
        dft_point_cnt <= 0;
        mel_fbank_out <= 0;
        mel_fbank_valid <= 0;
        for (i = 0; i < NUM_MEL_FILTERS; i = i + 1) begin
            mel_accumulators[i] <= 0;
        end
    end else begin
        if (dft_valid) begin
            // Multiply DFT output with the corresponding filter
coefficient
            mel_accumulators[mel_filter_cnt] <=
mel_accumulators[mel_filter_cnt] +
                (dft_out *
mel_filter_coefs[mel_filter_cnt][dft_point_cnt]);

            // Increment DFT point counter
            if (dft_point_cnt == NUM_DFT_POINTS - 1) begin

```

```

        dft_point_cnt <= 0;

        // Increment mel-scale filter counter
        if (mel_filter_cnt == NUM_MEL_FILTERS - 1) begin
            mel_filter_cnt <= 0;

            // Output the accumulated filterbank energies
            mel_fbank_out <= mel_accumulators[NUM_MEL_FILTERS -
1];

            mel_fbank_valid <= 1;

            // Reset the accumulators for the next frame
            for (i = 0; i < NUM_MEL_FILTERS; i = i + 1) begin
                mel_accumulators[i] <= 0;
            end
        end else begin
            mel_filter_cnt <= mel_filter_cnt + 1;
            mel_fbank_out <= 0;
            mel_fbank_valid <= 0;
        end
    end else begin
        dft_point_cnt <= dft_point_cnt + 1;
        mel_fbank_out <= 0;
        mel_fbank_valid <= 0;
    end
end else begin
    mel_fbank_out <= 0;
    mel_fbank_valid <= 0;
end
end
endmodule
`endif

```

In this updated code:

1. The assignment of `mel_fbank_out` is moved inside the `else` block of the `if (dft_valid)` condition. This ensures that `mel_fbank_out` is assigned a value only when `dft_valid` is asserted.
2. When the computation for all mel-scale filters is complete (i.e., `mel_filter_cnt` reaches `NUM_MEL_FILTERS - 1`), the `mel_fbank_out` signal is assigned the final accumulated value from `mel_accumulators[NUM_MEL_FILTERS - 1]`, and the `mel_fbank_valid` signal is set to 1 to indicate valid output.
3. In all other cases (when `dft_valid` is not asserted or when the computation is not complete), the `mel_fbank_out` signal is assigned 0 to ensure that all bits are driven, and the `mel_fbank_valid` signal is set to 0.

With these modifications, the `mel_fbank_out` signal should have all its bits properly driven based on the mel-scale filterbank computation, and the `mel_fbank_valid` signal should be correctly assigned to indicate valid output only when the computation is complete.

Remember to replace the placeholder coefficient values in the `initial` block with the actual values generated by the Python script.

Copy

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

EB

Why is not `mel_filterbank` instantiated in `mfcc` module?

I apologize for the confusion in my previous responses. You are correct that the `mel_filterbank` module is not instantiated in the `mfcc_accelerator` module. It seems that I overlooked this important detail.

To properly integrate the `mel_filterbank` module into the MFCC accelerator, we need to instantiate it within the `mfcc_accelerator` module and connect the appropriate signals.

Here's the updated `mfcc_accelerator` module with the instantiation of the `mel_filterbank` module:

verilog

Copy code

```
module mfcc_accelerator #(
    parameter MFCC_FEATURES = 40,
    parameter ACTIV_BITS = 8
) (
    // ... (other ports)
    input wire [31:0] dft_out,
    input wire dft_valid,
    // ... (other ports)
);

// ... (other code)

// Mel-scale filterbank signals
wire [31:0] mel_fbank_out;
wire mel_fbank_valid;

// Instantiate the mel_filterbank module
mel_filterbank #(
    .NUM_MEL_FILTERS(40),
    .NUM_DFT_POINTS(256),
    .COEF_WIDTH(16),
    .ACCUMULATOR_WIDTH(32)
) 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)
    );

    // ... (other code)

    // Instantiate the logarithm_comp module
    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)
    );

    // ... (other code)

endmodule

```

In this updated `mfcc_accelerator` module:

1. The `mel_filterbank` module is instantiated with the appropriate parameters (`NUM_MEL_FILTERS`, `NUM_DFT_POINTS`, `COEF_WIDTH`, `ACCUMULATOR_WIDTH`).
2. The `dft_out` and `dft_valid` signals from the `goertzel_dft` module are connected to the `dft_out` and `dft_valid` inputs of the `mel_filterbank` module.
3. The `mel_fbank_out` and `mel_fbank_valid` signals from the `mel_filterbank` module are connected to the `mel_fbank_out` and `mel_fbank_valid` inputs of the `logarithm_comp` module.
4. The rest of the MFCC accelerator pipeline remains the same.

With this modification, the `mel_filterbank` module is properly integrated into the MFCC accelerator, and the mel-scale filterbank computation will be performed on the DFT output before passing the results to the logarithm computation module.

I apologize for any confusion caused by my previous responses, and thank you for pointing out this important aspect of the MFCC accelerator design.