ECE559 Fall2018, Constituent Coder Interleaver Group

Group Design Report

xxx

December 9, 2018

1 A high-level description of the function

I wonder if vinith can do this 1. A high-level description of the function for which the team was responsible, including: o how this function relates to the overall function of the encoder stack o how this set of functions relates to the relevant 3GPP standards

2 Description of the design

The design overview is shown in the following block diagram, which includes functional blocks, interfaces (with more description later), and datapath:

Figure 1: Block diagram of functional blocks

Here are the enumeration & description of interfaces to other functions in the encoder stack:

- *input* **clock**: The clock.
- *input* **rst**: The reset input.
- *input* **k_size_6144**: The indication of the block size being transferred into our module from the code block segmentation module. This bit should be asserted 0 if block size is 1056, and asserted 1 if block size is 6144 when the ready_in signal was asserted.
- *input* **[7:0] databyte_in**: The input data byte of the data block from the code block segmentation module. This data is shifted into the "shift register". The right-most bit, or LSB, is the earliest bit in the sequence or bit with smaller index.
- *input* **shift_en**: This input control signal comes from code block segmentation module. At the clock rise, assertion of this makes the input byte shifted into the input shift register buffer.
- *input* **ready_in**: This input control signal comes from code block segmentation module. It indicates the finish of transfering of the data block. Once this signal is asserted, the output FSM starts operating and outputting to the turbo constituent coder module.
- *output* **k_size_out**: This output signal goes to the turbo constituent coder module. It tells the coder the size of the data block being output right now.
- *output* **ready_out**: This output signal goes to the turbo constituent coder module. It tells the coder the start of outputting the current data block. Only asserted for one cycle.
- *output* **outi**: This output signal goes to the turbo constituent coder module. It's the c_i output in the 3GPP document. It's also the output data block in original sequence.
- *output* **outpii**: This output signal goes to the turbo constituent coder module. It's the $c_{\pi i}$ output in the document. It's also the output data block in a remapped sequence indicated by the 3GPP document.

In terms of the datapath inside our module, there's not much combination logics but a bunch of wiring between functional blocks or even inside certain block.

1. The data is firstly shifted into the "shiftreg" buffer most of the time one byte per 8 cycle from the code block segmentation module. In terms of the endianess, the right-most bit, or LSB, is the earliest bit in the sequence or bit with smaller index. The code block segmentation module owns the control signal of "**shift_en**" to adjust timing to their output.

2. Once the whole data block is in, the "ready_in" signal is asserted. Then the data block in the "shiftreg" gets latched/held into the secondary "register" buffer, after which the "shiftreg" buffer can start taking next data block from the code block segmentation module while the current data block can be processed and output to the "turbo constituent coder". Simultaneously, the block size "**k_size_in**" of the current data block gets latched into the FSM and outputs as "**k_size_out**" such that the "turbo constituent coder" knows the size of the code block they are processing.

3. The output of the secondary "register" buffer, the whole data block, goes through the remapping module to generate the sequence specified by the 3GPP document purely by rewiring the bits. For example, $c_{\pi i}$ is just $c_{i,1056}$ given k=1056 or $c_{i,6144}$ given k=6144, where $_{i,1056}$ and $_{i,6144}$ are indices pre-calculated and hardcoded according to the 3GPP document. There's no combinational logic other than a mux that selects the corresponding bit given k inside the remapping module.

4. The original and the remapped data blocks are fed into two muxes seperately and the FSM outputs an ascending sequence to select the corresponding output bits to the "turbo constituent coder" module.

s
]iew from Quartus, with a brief prose description; this can be used to describe the block diagram mentioned in the previous bullet

Below is the state transition diagram for the counter module, which is a very simple FSM.

Figure 2: State transition diagram of output FSM

(!!!!!cheng can you talk abt the fsm?) -o FSM designs, with pictorial description (s) and text, clearly indicating what each state represents, how inputs determine the next state and Moore and Mealy outputs for each state, and also an indication of how the FSM is related to data flow through the function

3 Static timing analysis

(mostly cheng) 3. Static timing analysis o from the static timing analysis results, identify (a) worst-case slack against setup violations; (b) maximum frequency at which the design can be clocked; (c) the critical path corresponding to the worst-case slack. o based on the max frequency, comment on the max throughput that can be sustained by the function given the current design

4 Simulation results

Firstly, there's a python script that calculates the output sequence give the input data block. Below is a pair of console outputs generated by the python script:

Figure 3: c_i

Figure 4: c_i

On the left is the original data block and on the right is the remapped data block. The orignial data block is just the repeating pattern byte "11111011". Again, the right-most bit, or LSB, is the earliest bit in the sequence or bit with smaller index. In the console output, the right-most bit, or LSB, is also the earliest bit in the sequence or bit with smaller index. The left most bit is the last output bit. This script is used to calculate the output sequence.

The functional correctness of the overall module is verified with a testbench mimicing inputs from the code block segmentation module and monitoring the outputs to the turbo constituent coder.

In this testbench, the size k is 6144. The **databyte_in** was asserted as "11111011", which was the input pattern the python script ran on. The **shift_en** was asserted for 768 clock cycles to pass in the

whole 6144-sized data block. Then the **shift_en** was desserted and the **ready_in** was asserted. Here's the console output generated by the testbench monitoring the two output bits:

Figure 5: Console output

The mux index, which indicates which bit is being output (starting from 0 instead of 1), and the counter FSM state were also printed, in additional to the two output bits. It can be seen from the terminal output above that the mux index stopped increasing at 6143 and counter FSM state increased to 2, so this is the end of output sequence. Shown in the terminal are the last 6 bits for both outputs. With the LSB being the bit with smallest index, the last 6 bits for output c_i and $c_{\pi i}$ are "110111" and "111101" respectively, matching the outputs generated by the python script.

This simulated test verified the FSM transition and the functional correctness of the "constitunt coder interleaver", in terms of the output sequences and interfaces with other functional modules in the code stack.

!!!!little confused with these
and in a prose discription, what the simulation results indicate with respect to operation and performance of the subsystem

5 Hardware test results

(mostly vinith) 5. Hardware test results o what was run on the hardware, was it run at speed or "statically"? o show appropriate indication of the test results (e.g. prints of logic analyzer displays), with clear annotation on the plots and accompanying prose decription o what were the test conditions, what is the significance of the test in the context of the subsystem function o relationship to simulation test results

6 A1

The circuit in Figure 6.8 was and simulated to find A1. Here're the simulation result:

Figure 6: Log magnitude and phase of A1

7 Closed-loop gain of non-inverting mode amplifier

The non-inverting mode amplifier circuit was built and simulated to find closed-loop gain Af. Here're the simulation result:

8 Design an inverting mode amplifier

Here are the conditions required:

$$VS/RS = -VO/RF$$
$$VO/VS = -RF/RS = -4$$