Duke University

ECE 559: Advanced Digital Systems

Turbo Coder Internal Interleaver

Vinith Sharma, Yao Yuan, Cheng Lyu

Fall 2018

1 3GPP LTE Advanced Wireless System

1.1 Introduction

The next advancement on wireless communication was proposed by the 3rd Generation Partnership Program (3GPP). Formally a candidate 4G to ITU-T in 2009, it was standardized in March of 2011. The biggest benefit of LTE Advanced is the ability to take advantage of avdanced topology networks. The usage of optimized heterogeneous networks and utilizing macrocells with low power nodes such as pico and femto cells create a much better network compared to macrocells (wide area high power base stations that covers a large radius). LTE Advanced also introduces multicarrier so we can utilize wider bandwidth, up to 100 MHz, which will support high data rates.

Our focus this semester lied on the coding, multiplexing and mapping to physical channels for E-UTRA section of the LTE Advanced specification as seen in 3GPP TS 36.212 V10.5.0. The sub-systems extracted from this document that our class worked on is as follows:

- Code block segmentation and CRC
- Convolutional encoder
- Turbo-code constituent encoder
- Turbo-code interleaver
- Sub-block interleaver and multiplexer

1.2 Internal Interleaver

Our group was responsible for the turbo coder internal interleaver. The purpose of this subsystem is to block interleave the data, frame quality indicator (CRC), and the reserve bits input to the turbo encoder. This interleaving happens through a index generator function defined in section 5.1.3.2.3 of the document. First we define an index generator function that selects the certain index of output bits from a certain input bit index. We will call this $\Pi(i)$:

$$\Pi(i) = (f_1 \cdot i + f_2 \cdot i^2) \mod K$$

where f_1, f_2 are defined in Table 5.1.3-3 in the 3GPP document. These values depend on the block size K. In our case, K could only be two different sizes 1056 and 6144. This leads us to the following values for f_1 and f_2:

K	f_1	f_2
1056	17	66
6144	263	480

Using these values, we can generate the output bits using the following function:

$$c'_i = c_{\Pi(i)} \quad \forall i \in \{0, 1, 2, ..., K\}$$

where c_i is the i^{th} bit of the input sequence and c'_i is the i^{th} bit of the output sequence defined by the function $\Pi(i)$.

This sub-module is part of the larger sub-system, turbo-code constituent encoder. The turbo encoder's purpose is to encode the incoming data, CRC, and the two reserved bits. During encoding, the output tail sequence is also added. The encoder generates the output based on the code rate. In our case, the code rate was 1/3. It uses two systematic, recursive, convolutional encoders connected in parallel with an interleaver for the second encoder. The two recursive convolutional codes are called the constituents codes of the turbo code. Since there is delay after the interleaver processes the data, the second encoder must account for this delay of the incoming data sequence.

2 Description of the design

One of the important changes we want to highlight is the `outi` port. According to the 3GPP document's block diagram as seen in Figure 2, the first constituent encoder takes in the sequence directly from the CSU, where as the other encoder has to wait some amount of cycles determined by the bit-serial vs. byte-serial. Trying to implement this exact design would create multiple ports in both ours and constituent encoder's design. Thus, we decided that the interleaver group would be responsible for serializing and feeding both the original and interleaved data directly to the constituent encoders.

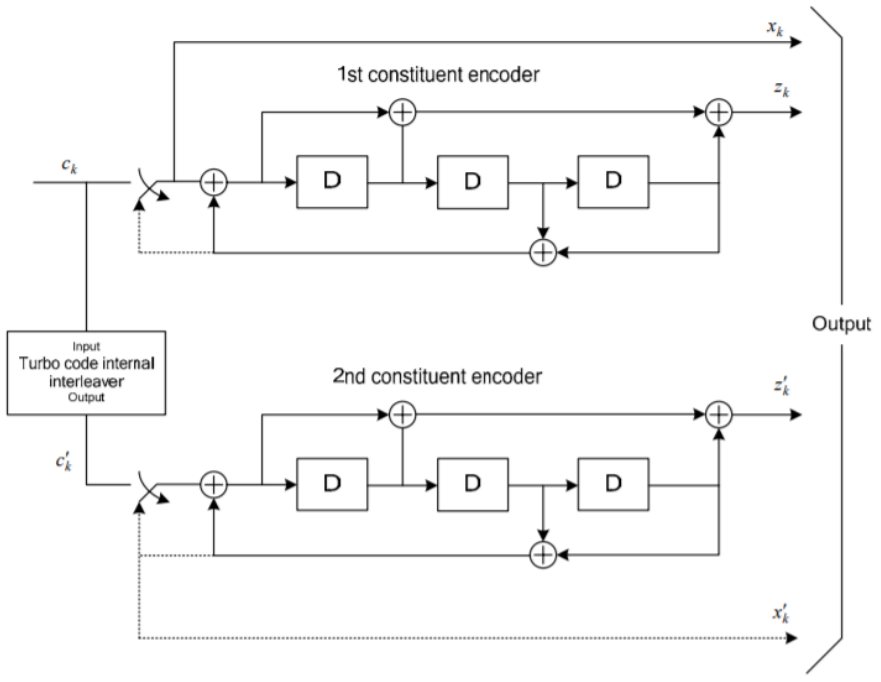


Figure 1: The high level design shown in the 3GPP document.

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

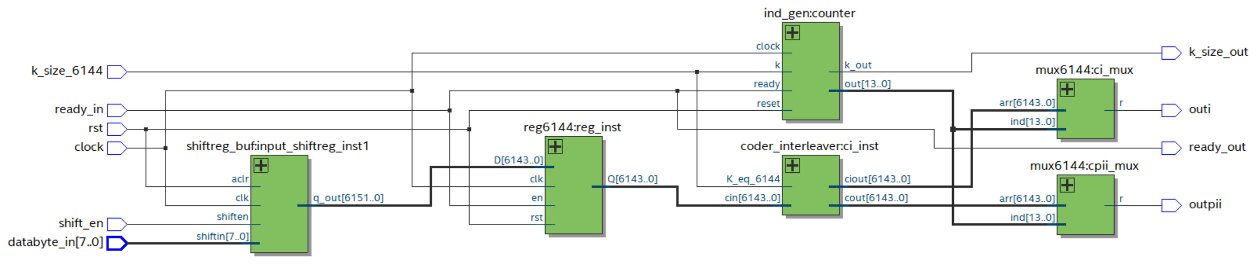


Figure 2: Block diagram of functional blocks

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 transferring 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 logic 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 endianness, the rightmost 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 hard coded 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 separately and the FSM outputs an ascending sequence to select the corresponding output bits to the "turbo constituent coder" module.

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

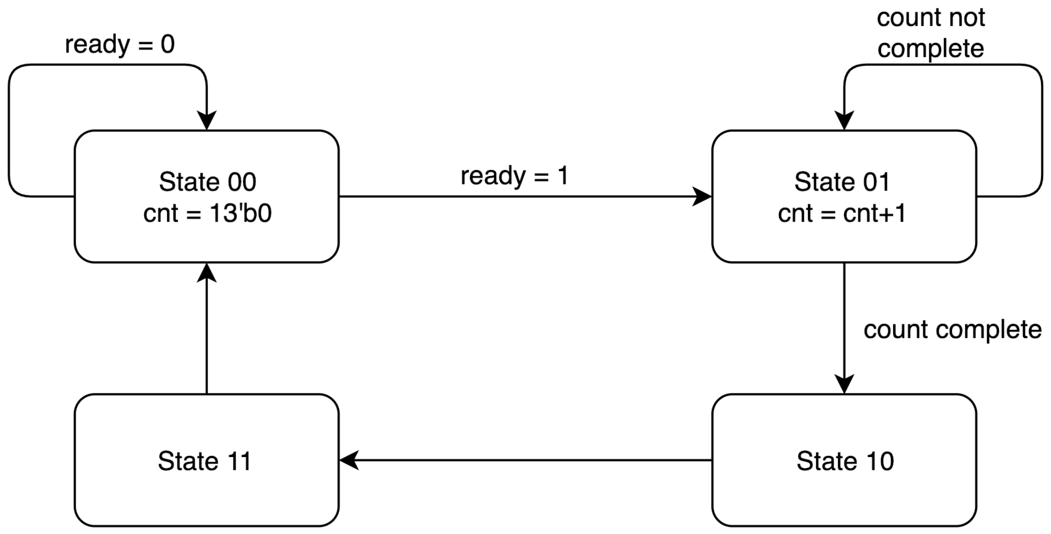


Figure 3: State transition diagram of output FSM

This FSM contains four states. The state in this FSM doesn't make output, but manipulate the 13-bit register `cnt`, which relates to the output of the module. The function of each state is listed below.

- State00 is the waiting state. It waits for a `ready` signal to be pulled high to go to State01. If the `ready` signal stays low, State00 keeps going back to itself. At State00, `cnt` is set to 0.
- State01 is the counting state. This state goes to State10 is the counting is complete, otherwise it goes to itself. The counting is considered complete if either the `kout` signal is 0 and `cnt` is 1055 or the `kout` signal is 1 and `cnt` is 6143. At State01, `cnt` is incremented by 1.
- State10 is a reserved state to accommodate with any signal indicating data readiness required in the future.
- State11 is a reserved state to accommodate with any signal indicating data readiness required in the future.

3 Static Timing Analysis

A comprehensive static timing analysis includes analysis of register-to-register, I/O, and asynchronous reset paths. It verifies circuit performance and detect possible timing violations by investigating data required times, data arrival times, and clock arrival times. The Timing Analyzer determines the timing relationships that must be met for the design to correctly function, and checks arrival times against required times to verify timing. In our design, we focus on the timing constraints on internal register-to-register paths and are not worrying about the I/O constraints. The Timing Analysis is operated with the Base Clock at 50 MHz. We used `Post-fit` and `Fast-corner` for a full timing analysis.

3.1 Identifications

3.1.1 Worst Case Slack

The Timing Analyzer reports the result of clock setup checks as slack values. Slack is the margin by which a timing requirement is met or not met. Positive slack indicates the margin by which a requirement is met, and negative slack indicates the margin by which a requirement is not met. The Timing Analyzer determines clock setup slack as shown in the following equation for internal register-to-register paths, where *slack* is Clock Setup Slack Time, $t_{require}$ is Data Required Time and t_{arrive} is Data Arrival Time.

$$slack = t_{require} t_{arrive} \tag{1}$$

Since slack indicates that the clock meets the requirement with some margin, larger slack means better performance. According to the `Report All Core Time` page in our design, The Worst Case Slack against Setup Violations is 13.307 ns. From this result, we are confident that we don't have any failing path since there's no negative slack for the clock domain.

3.1.2 Maximum Frequency

According to `Report FMax Summary`, we find that the maximum frequency at which our design can be clocked is 149.41 MHz.

3.1.3 Critical Path

By using `Report Worst Case Path` function, we find that the worst data delay is 6.528 ns and the path of Worst Case Slack is:

```
shiftreg_buf:input_shiftreg_inst1|mem[1464] :  (0.232 ns)
input_shiftreg_inst1|mem[1464]|q :  (0 ns)
reg_inst|Q[1464]|asdata :  (5.89 ns)
reg6144:comb_6149|Q[1464] :  (0.406 ns)
```

According to the Path above, `reg_inst|Q[1464]|asdata` is the Critical Path since it takes the longest delay which is 5.89 ns. According to the path name, we locate this path in the second buffer of our design.

The plot of Worst Case Path is shown in the plot below. The green region represents the slack area. We can see that with the clock period of 20 ns, our design is fairly fast since the worst slack period is longer than half of the clock cycle.

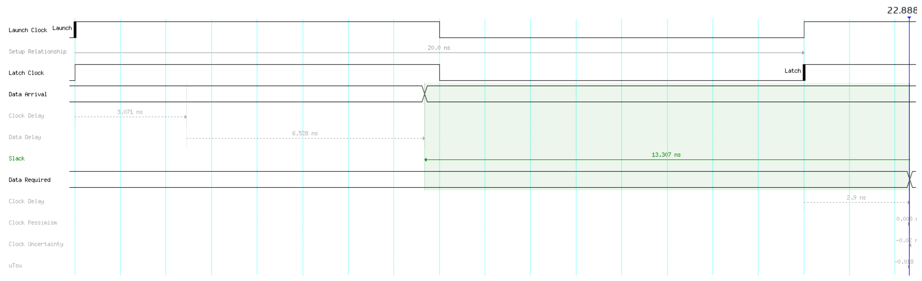


Figure 4: Worst Case Path

3.2 Max Throughput

We are using the following equation to calculate Throughput, where $data_{period}$ represents Data Per Clock Period and $freq$ represents Clock Frequency.

$$throughput = data_{period} * freq \tag{2}$$

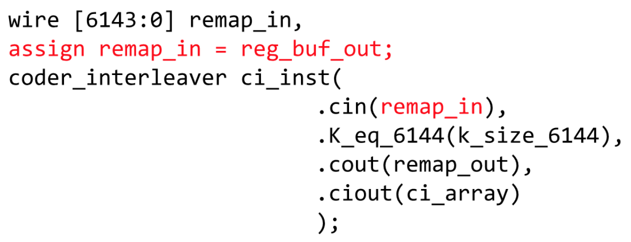
According to the equation above, with Max Frequency as 149.41 MHz and data per clock period as 1 bits, we have

$Throughput = 1bits * 149.41MHz = 0.15bits/ns = 150MB/sec.$

According to the calculation above, the Max Throughput is estimated to be 150 MB/sec.

3.3 Modification

The initial design includes redundant nodes, which yields a Worst Case Slack as 8.079 ns and Max Frequency as 83.89 MHz. An example of the redundant nodes is show in the picture below where the declaration of `remap_in` is not necessary.



```
wire [6143:0] remap_in,
assign remap_in = reg_buf_out;
coder_interleaver ci_inst(
                         .cin(remap_in),
                         .K_eq_6144(k_size_6144),
                         .cout(remap_out),
                         .ciout(ci_array)
                         );
```

Figure 5: Redundant Node

After removing the redundant wires, the Worst Case Slack is 13.307 ns and the Max Frequency increases to 149.41 MHz. We find huge time improvement after removing the redundant nodes which is very surprising that wiring between two nodes may take such a huge amount of extra time. The trade-off here is between the easiness in understanding the code and the time performance of the design.

3.3.1 Worst Path Comparison

The worst slowest path before node reduction is 11.216 ns, and after the node reduction is 5.89 ns. We see huge timing improvements.

- Before Wire Reduction:
  ```
  shiftreg_buf:input_shiftreg_inst1|mem[4653] :  (0.232 ns)
  input_shiftreg_inst1|mem[4653]|q :  (0 ns)
  comb_6149|Q[4653]|asdata :  (11.216 ns)
  reg6144:comb_6149|Q[4653] :  (0.406 ns)
  ```

- After Wire Reduction:
  ```
  shiftreg_buf:input_shiftreg_inst1|mem[1464] :  (0.232 ns)
  input_shiftreg_inst1|mem[1464]|q :  (0 ns)
  reg_inst|Q[1464]|asdata :  (5.89 ns)
  reg6144:comb_6149|Q[1464] :  (0.406 ns)
  ```

3.3.2 All Clock Histogram Comparison

According to the All Clock Histogram shown below, most of the slack are between 16 ns and 19 ns which doesn't change a lot before and after node reduction. However a samll amount of slack less than 13 ns are improved because of the node reduction. We can see that the node reduction greatly improved the worst slacks while doesn't have big influence on other slacks.

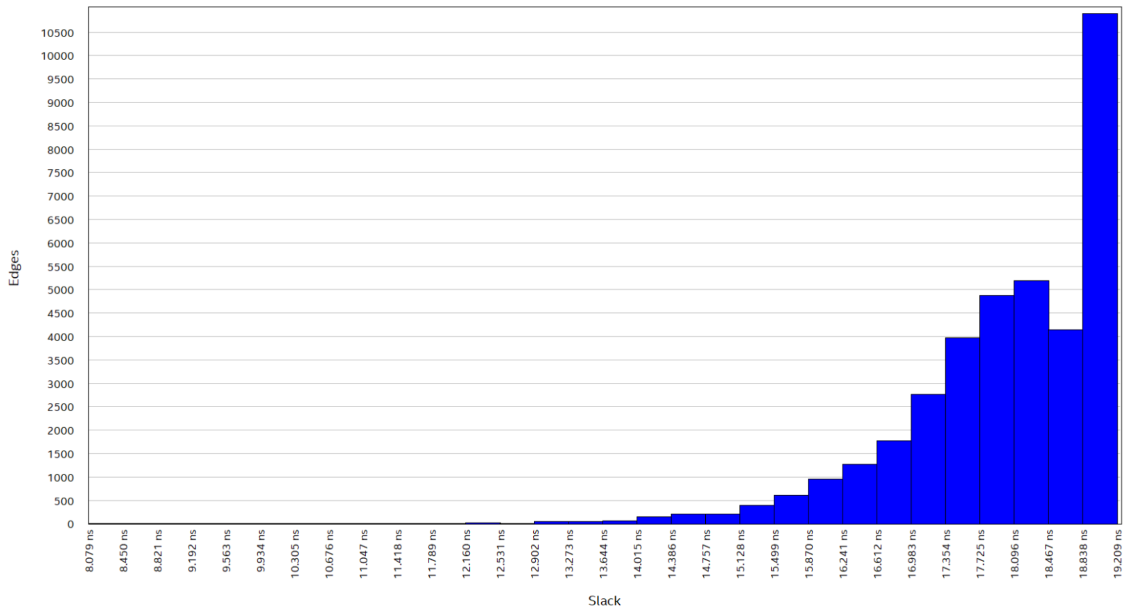


Figure 6: All Clock Histogram after node reduction

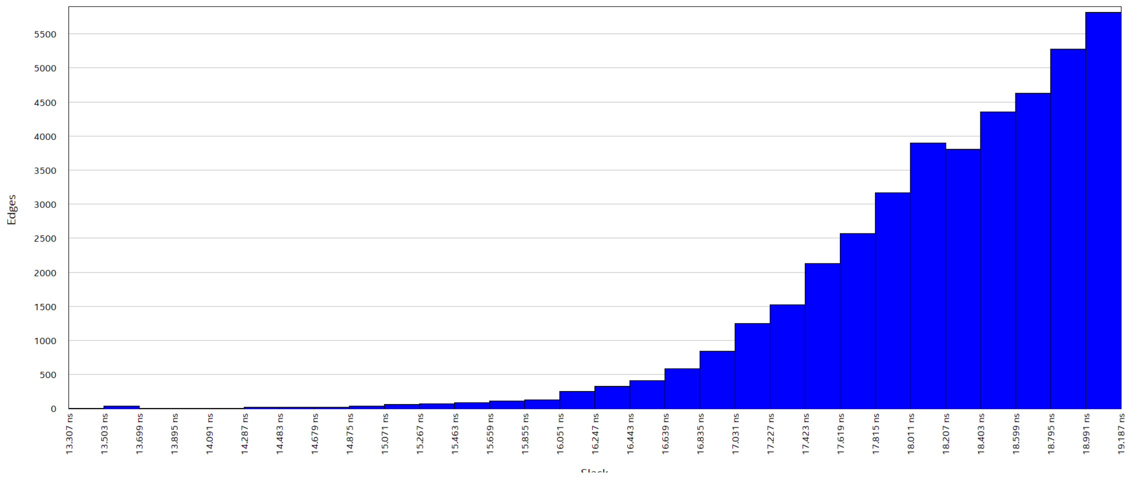
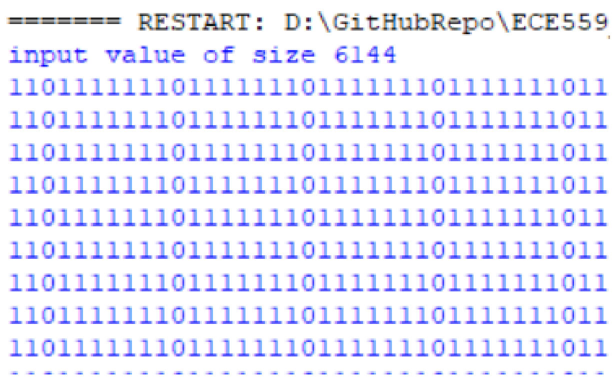


Figure 7: All Clock Histogram after node reduction

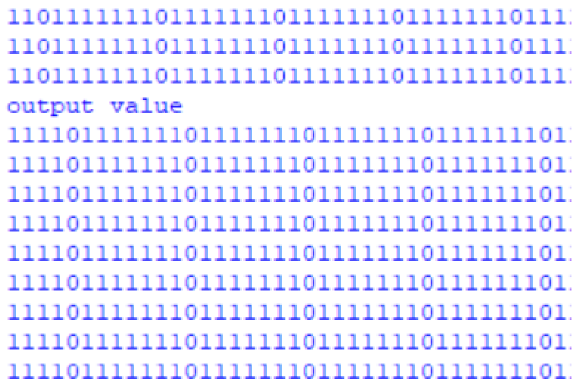
4 Simulation results

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:



```
======= RESTART: D:\GitHubRepo\ECE559
input value of size 6144
1101111111011111110111111101111111011
1101111111011111110111111101111111011
1101111111011111110111111101111111011
1101111111011111110111111101111111011
1101111111011111110111111101111111011
1101111111011111110111111101111111011
1101111111011111110111111101111111011
1101111111011111110111111101111111011
1101111111011111110111111101111111011
```

Figure 8: c_i



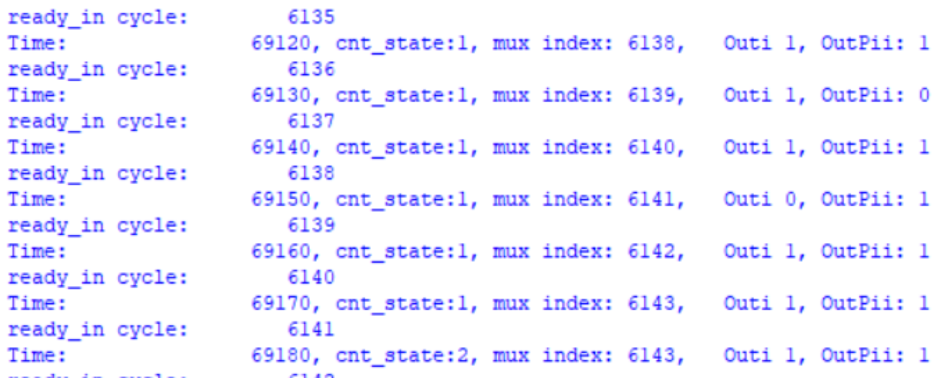
```
11011111110111111101111111011111110111
11011111110111111101111111011111110111
11011111110111111101111111011111110111
output value
11110111111101111111011111110111111101
11110111111101111111011111110111111101
11110111111101111111011111110111111101
11110111111101111111011111110111111101
11110111111101111111011111110111111101
11110111111101111111011111110111111101
11110111111101111111011111110111111101
11110111111101111111011111110111111101
11110111111101111111011111110111111101
```

Figure 9: $c_{\pi i}$

On the left is the original data block and on the right is the remapped data block. The original 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 mimicking 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 de-asserted and the **ready_in** was asserted. Here's the console output generated by the testbench monitoring the two output bits:



```
ready_in cycle:        6135
Time:               69120, cnt_state:1, mux index: 6138,   Outi 1, OutPii: 1
ready_in cycle:        6136
Time:               69130, cnt_state:1, mux index: 6139,   Outi 1, OutPii: 0
ready_in cycle:        6137
Time:               69140, cnt_state:1, mux index: 6140,   Outi 1, OutPii: 1
ready_in cycle:        6138
Time:               69150, cnt_state:1, mux index: 6141,   Outi 0, OutPii: 1
ready_in cycle:        6139
Time:               69160, cnt_state:1, mux index: 6142,   Outi 1, OutPii: 1
ready_in cycle:        6140
Time:               69170, cnt_state:1, mux index: 6143,   Outi 1, OutPii: 1
ready_in cycle:        6141
Time:               69180, cnt_state:2, mux index: 6143,   Outi 1, OutPii: 1
```

Figure 10: 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 "constituent coder interleaver", in terms of the output sequences and interfaces with other functional modules in the code stack.

5 Hardware test results

Initial concerns about the simulation was the size of the blocks and the number of bits we have to deal with. Trying to check every bit in a block of size 6144 would take an unnecessary amount of time. Thus, Vinith wrote a python script that simulated this submodule. Listing 1 is a python function that takes in two parameters `input_num`, the input sequence represented as a bit-string, and `flag_6144`, a one bit number that represents the size of the block. Writing 6144 bits would also take a tremendous amount of time and any sort of recurring pattern in the input will be present in the output, thus Listing 2 is another function that generates a random bit-string of size specified inside the `range()` function in the `for` loop.

The initial hardware testing was done using a `reg` with a constant value and passing all the bits in block side K into our interleaving module. This was the one and only test run statically. The clock did not matter for this because everything was done instantaneously as this test was essentially rewiring the wires based on the input values. We then took the bottom 10 bits of the output sequence and displayed it using the LEDs in the FPGA. After verifying that the output sequence matched, we moved on to the next part in hardware testing: simulating an incoming bit sequence. The test above did not really tell us how it would be used in real life, thus we had to incorporate a mechanism that simulated an incoming n-bit wide sequence.

We skipped the bit-serial implementation and tested a byte-serial implementation. We created a byte-shift register that is 6144 bit wide. Since we did not have the code segmentation unit block, we used a read-only memory to hold the input sequence. Using a counter up to $\frac{6144}{8} = 768$, we took the values out of the ROM and fed it into the shift register. After 768 cycles, we latched the value onto a buffer register and fed it into our interleaving module. The output value was once again latched onto another register and displayed using the LEDs and 7-segment displayed found on the FPGA. This test was done with a 50 Hz clock so we could visibly see as the counter increments and the values from ROM being shifted into the shift register. The first and last three bytes of `mif` file that was loaded onto the ROM is as follows:

Table 1: First and last three bytes in the `mif` file

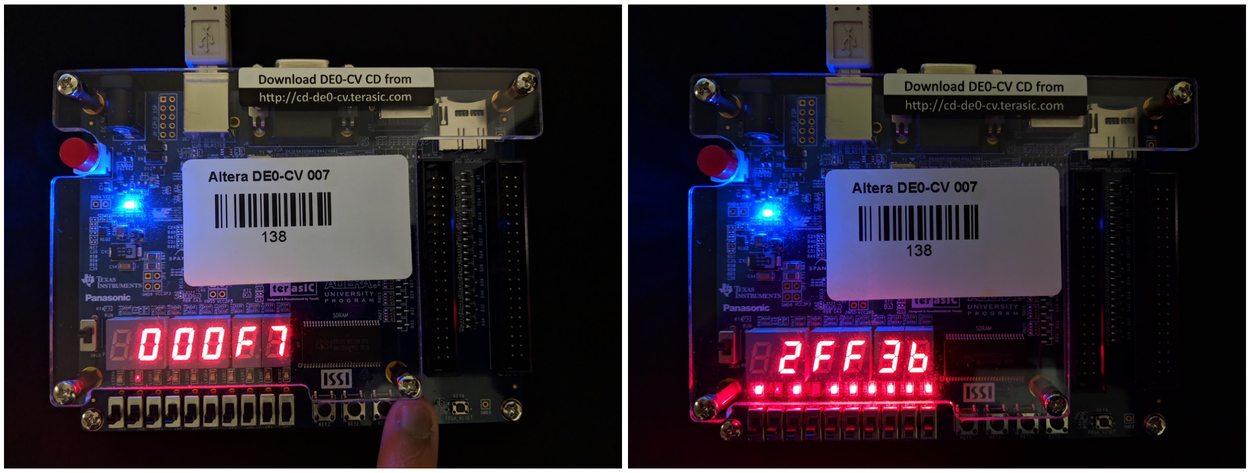
(a)

Address	Binary	Hex
0	11110111	F7
1	00111100	3C
2	11000110	C6

(b)

Address	Binary	Hex
765	11111000	F8
766	01111110	7E
767	00111011	3B

where the list index is the address in memory. Since the value is shifted right, we had to invert the original input and slice it into 8 bits. This was verified as seen in Figure 11a and Figure 11b. Of the five seven segment displays, the left three are used to show the counter as it increments from 000 to 2FF in base 16. The right two displays are used to show the output byte as we shift it into the byte-shift register. We then took the 8 LSB and displayed it using the LEDs present in the FPGA. According to the output from the python script, the least significant byte for that input is: 10111111. This is also verified from Figure 11b as the right most 8 LEDs are correctly turned on in that order.



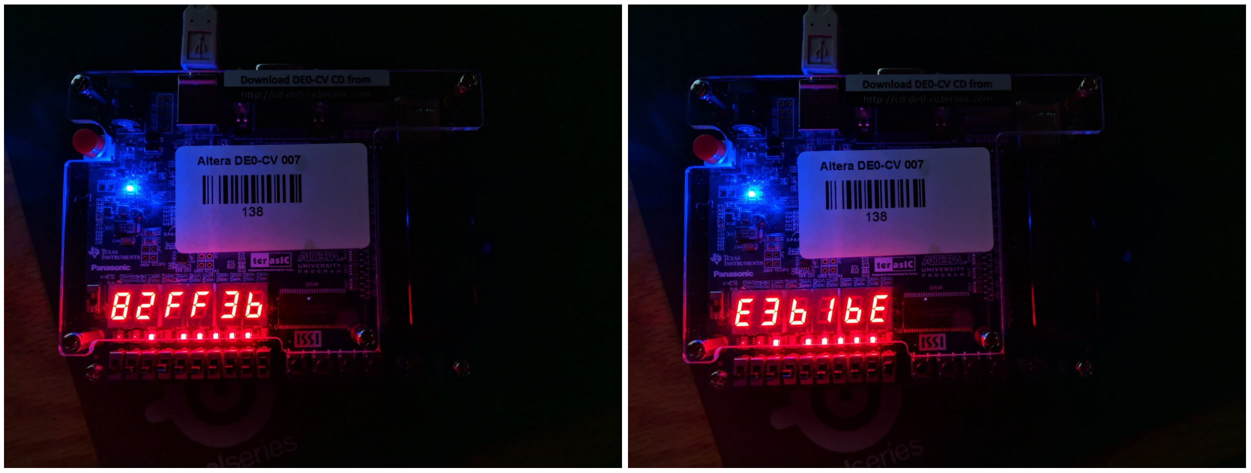
(a) Initial state

(b) Final state with LEDs

Figure 11: The initial and final state of the simulation as the data from ROM is put into the shift register. The LEDs represent the 8 output LSB bits.

Since the logic analyzer did not work, we were unable to test extensively using the display. Thus, to do the best test possible, we decided to use a slide switch to transition between two states of the seven segment decoder. The first state shown in Figure 12a shows us that when the switch is down (logical '0') the display shows us exactly what used to be. The new addition is when the switch is up (logical '1'), the seven segment display switches to show us three least significant bytes: `E3B1BF`. This matches exactly what the python script outputs as the least significant 24 bits: `1110 0011 1011 0001 1011 1111`.

These hardware results prove that even when combined with the CSU, our module correctly outputs the data as expected. There is always the chance of 'what if the software tests are wrong?', but I am confident in my python script. These results go alongside the software simulation as both software and hardware match the testbench simulation.



(a) State with switch down   (b) State with switch up

Figure 12: A slightly extensive test to check for the outputs.

Appendix

Listing 1: The function that simulates the interleaver.

```
def interleaver(input_num, flag_6144):
    f1 = 263 if flag_6144 == 1 else 17
    f2 = 480 if flag_6144 == 1 else 66
    k = 6144 if flag_6144 == 1 else 1056
    input_inv = input_num[::-1]

    output = [''] * k
    for i in range(k):
        pi_i = (f1 * i + f2 * i ** 2) % k
        output[i] = input_inv[pi_i]
    return ''.join(output)[::-1]
```

Listing 2: The function that generates a random block with uniform random bits as well as input last bit.

```
def create_val(last_number):
    out_str = ''
    for x in range(6143):
        out_str += str(random.randint(0, 1))
    out_str += last_number
    return out_str
```