Final project -Half Rate Convolutional Code

Diego Corna, Filippo Corna
Computer Science Engineering Department
Politecnico di Milano, Milan (MI), Italy

Academic Year 2021/2022

1 Introduction

The project consists of the creation of a hardware module described in VHDL, which interfaces with the memory according to the following specification. The module receives a continuous stream of 8-bit words and outputs a continuous sequence of words, each 8 bits long, encoded as follows: each input word is serialized, producing a continuous stream of 1 bit, to which the convolutional code $\frac{1}{2}$ is applied, encoding each bit with 2 bits, according to the scheme in the figure. This generates a continuous stream Y.

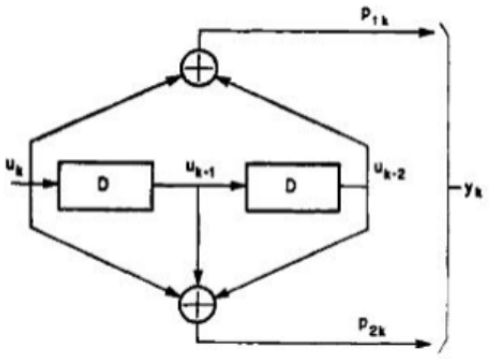


Figure 1: HR Convolutional Code

According to the notation in the figure 1, the Y stream is obtained as an alternating concatenation of the two output bits. The U_k bit generates the p_{1k} and p_{2k} bits, which are concatenated to generate the 1-bit stream y_k. The final output sequence is the parallelization, on 8 bits, of the continuous stream y_k.

The module to be implemented must read the sequence to be encoded from a memory with addressing for the byte in which it is stored. The number of words to be encoded is stored at address 0, and words of the input stream are saved starting from memory cell 1. The output stream must be stored starting from address 1000, and the maximum size of the input sequence is 255 bytes.

Processing will start when a START input signal is set to 1. This signal will remain high until the DONE signal is set to high at the end of the computation. DONE must remain at 1 until START is set to 0. START cannot be set to high until DONE returns to 0, allowing the encoding to restart. It is, therefore, possible to encode multiple flows one after the other without having to "reset" the module with the RESET signal. The module must be designed considering that before the first encoding, a RESET is always given to the module, while in case of a second processing, it is not necessary to wait for the RESET, but simply for the termination of the processing

2 Operation Diagram

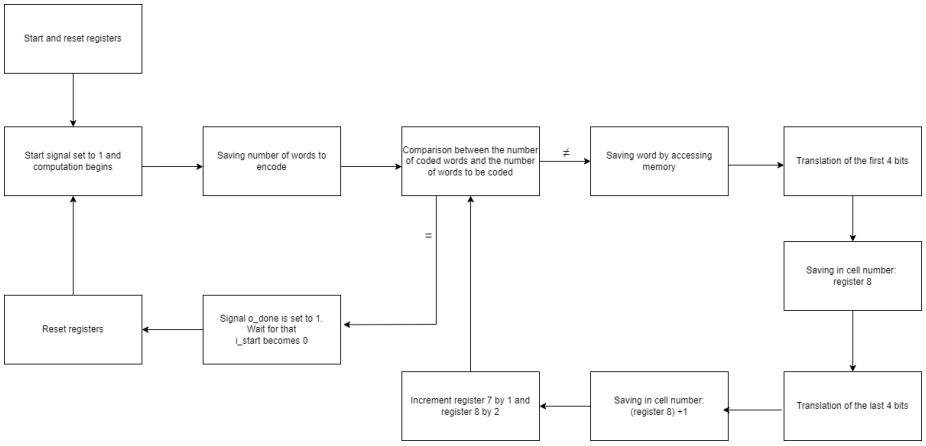
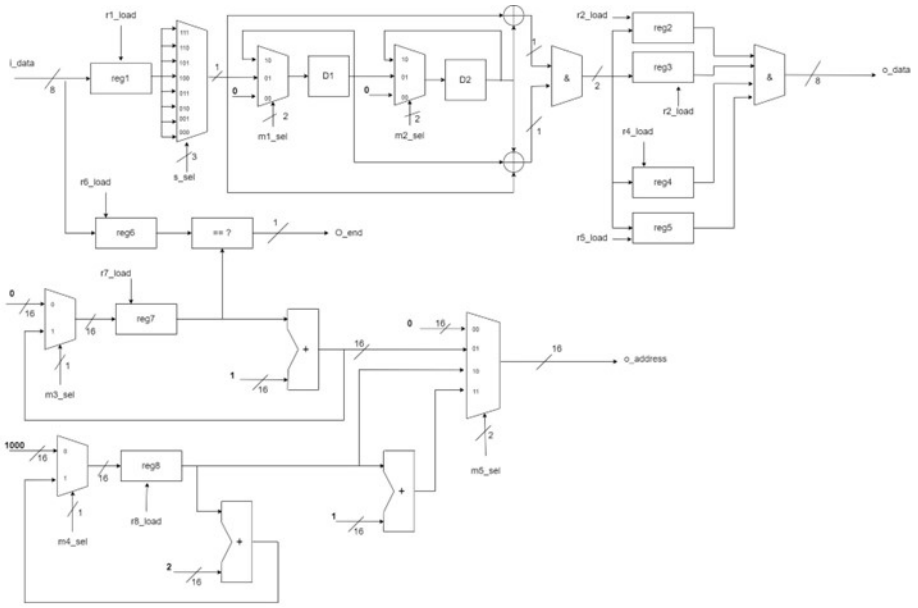


Figure 2: Operation Schema

The translation of the word is as follows: one bit is passed at a time, and two bits are generated in signals PK1 and PK2. The concatenation of these signals is then saved in registers 2, 3, 4, and 5; the first bit of the word to be encoded will generate 2 bits that are saved in register 2, the second bit will generate 2 bits that will be saved in register 3, and so on. Subsequently, the concatenation of these four pairs of bits will compose an 8-bit word. This procedure is repeated with the second 4 bits of the word to be encoded, resulting in another 8-bit word obtained.

3 Register and Component Descriptions



- **Reg1**: Register where the word to be encoded in 8 bits is saved, driven by the $r1\_load$ signal. It is connected to a MUX m_sel that individually selects the bits of the word that will then be processed according to the specification.

- **Msel**: MUX that selects through the m_sel signal (3 bits) the individual bits to be encoded of the word recorded in $r1$.

- **Reg2 – Reg5**: Registers where the bit pairs encoded according to the specification are saved, which are then concatenated to compose the 8 bits of o_data. The first 4 bits of the word are processed initially (encoded one at a time and the resulting bit pair is saved in the correct register) and then the last 4 (similarly) to obtain a 16-bit encoded word divided into two 8-bit parts, saved starting from memory address 1000.

- **Reg6**: Register where the number of words to be computed is saved, driven by the $r6\_load$ signal. It is connected to a comparator that compares it with the contents of $Reg7$.

- **Reg7**: Register where the number of computed words is saved, driven by the $r7\_load$ signal. The content saved is selected by the MUX $m3$.

- **M3**: MUX that controls with the $m3\_sel$ signal which address to save in $Reg7$:

  - 0: 0

    - 1: contents of $Reg7$ incremented by 1. As words flow into the code, the contents of $Reg7$ continuously grow by 1.

- **Reg8**: Register where the 16-bit memory address to write the result is saved, driven by the $r8\_load$ signal. The address saved is selected by the MUX $m4$ and starts at 1000.

- **M4**: MUX that controls with the $m4\_sel$ signal which address to save in $Reg8$:

    - 0: 1000

    - 1: contents of $Reg8$ incremented by 2. It is incremented by 2 because the result is divided into two 8-bit parts.

- **M5**: MUX that controls the value of $o\_address$ through the $m5\_sel$ signal:

    - 00: 0

    - 01: contents of $Reg7 + 1$

    - 10: content of $Reg8$

    - 11: content of $Reg8 + 2$

- **D1**: D flip-flop as described in the specification: connected to one of the two XOR gates and to the MUX $m1$.

- **M1**: MUX that controls the input signal of the D1 flip-flop with the $m1\_sel$ signal:

    - 00: D1

    - 01: the bit selected by $Msel$

    - 10: 0

- **D2**: D flip-flop as described in the specification: connected to the two XOR gates and the MUX $m2$.

- **M2**: MUX that controls the input signal of the D2 flip-flop with the $m2\_sel$ signal:

    - 00: D2

    - 01: D1

    - 10: 0

- **M1 and M2**: Used to ensure correct resetting of the D1 and D2 flip-flops.

- **Comparator**: Compares the number of words found with the number of words to translate. If they are equal, it sets o_end to 1.

4 State Machine

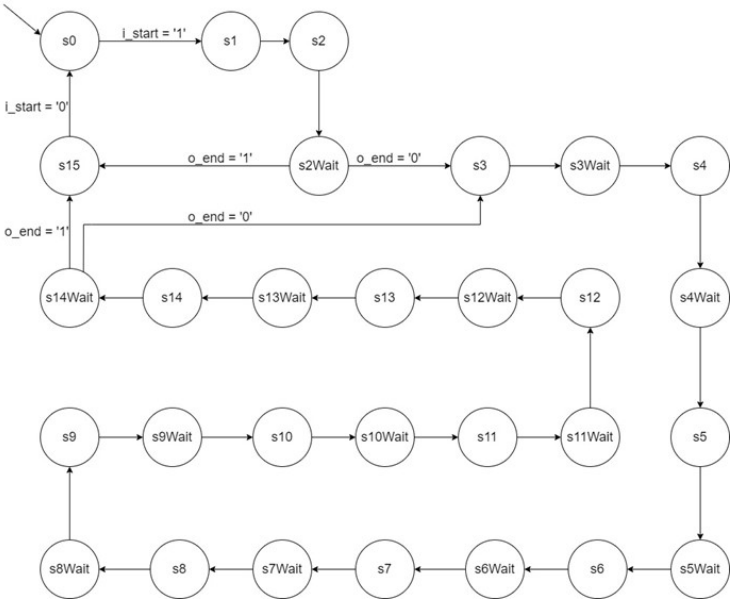


Figure 3: Finite State Machine Diagram

4.1 Default Values

$r1\_load = r2\_load = r3\_load = r4\_load = r5\_load = r6\_load = r7\_load = r8\_load = 0$

$s\_sel = 000$

$m1\_sel = m2\_sel = 10$

$m3\_sel = m4\_sel = 1$

$m5\_sel = 00$

$o\_en = o\_we = o\_done = 0$

4.2 States

- S0: $r7\_load = r8\_load = 1$, $m1\_sel = m2\_sel = 00$, $m3\_sel = m4\_sel = 00$
- S1: $o\_en = 1$, $m5\_sel = 00$
- S2: $r6\_load = 1$
- S3: $o\_en = 1$, $m5\_sel = 01$
- S4: $r1\_load = 1$

- S5: $s\_sel = 111$, $m1\_sel = m2\_sel = 01$, $r2\_load = 1$

- S6: $s\_sel = 110$, $m1\_sel = m2\_sel = 01$, $r3\_load = 1$

- S7: $s\_sel = 101$, $m1\_sel = m2\_sel = 01$, $r4\_load = 1$

- S8: $s\_sel = 100$, $m1\_sel = m2\_sel = 01$, $r5\_load = 1$

- S9: $o\_en = 1$, $o\_we = 1$, $m5\_sel = 10$

- S10: $s\_sel = 011$, $m1\_sel = m2\_sel = 01$, $r2\_load = 1$

- S11: $s\_sel = 010$, $m1\_sel = m2\_sel = 01$, $r3\_load = 1$

- S12: $s\_sel = 001$, $m1\_sel = m2\_sel = 01$, $r4\_load = 1$

- S13: $s\_sel = 000$, $m1\_sel = m2\_sel = 01$, $r5\_load = 1$

- S14: $o\_en = 1$, $o\_we = 1$, $m5\_sel = 11$, $r7\_load = 1$, $r8\_load = 1$

- S15: $o\_done = 1$

4.3 Status Description

- **S0**: Initial state which is also used as a reset state. In this state, all the datapath components are initialized. Registers 7 and 8 are initialized to 0 and 1000 respectively, and the flip-flops to 0. If the start signal is at 1, the state goes to S1; otherwise, it remains in S0.

- **S1**: The number of words to be encoded is read from memory at address 0 (asynchronous memory, so the number will be provided by memory in the next cycle).

- **S2**: The number requested from the memory in state S1 is written in register 1.

- **S2Wait**: The value of *o_end* provided by the comparator is evaluated. If $o\_end = 0$, it goes to state S3; if $o\_end = 1$, it goes to S15.

- **S3**: The word to be encoded is read from memory.

- **S4**: Save the word requested in the previous cycle in register 1.

- **S5-S8**: States for the encoding process of the first 4 bits of the word saved in register 1.

- **S9**: Write to memory the concatenation of registers 2, 3, 4, and 5 in the memory cell to the number saved in register 8.

- **S10-S13**: States for the encoding process of the second 4 bits of the word saved in register 1.

- **S14**: Write to memory the concatenation of registers 2, 3, 4, and 5 in the memory cell to the number saved in register 8 plus 1.

- **S14wait**: The value of *o_end* provided by the comparator is evaluated. If $o\_end = 0$, it goes to state S3; if $o\_end = 1$, it goes to S15.

- **S15**: Encoding ends, and the *o_done* signal is set to 1. It switches to the S0 state only when the *i_start* signal is set to 0. This is the only state in which *o_done* is set to 1; in all the others, it is set to 0.

- **SXwait**: These are the waiting states that ensure that the DATAPATH signals take on the correct values to allow the correct use of these by the other states.

5 EXPERIMENTAL RESULTS

5.1 REPORT

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	78	0	0	134600	0.06
LUT as Logic	78	0	0	134600	0.06
LUT as Memory	0	0	0	46200	0.00
Slice Registers	85	0	0	269200	0.03
Register as Flip Flop	85	0	0	269200	0.03
Register as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

Timing Report

Slack (MET) : 5.377ns (required time - arrival time)

Figure 4: Report

5.2 SIMULATIONS

The following tests were passed in both Behavioral Simulation and Post-Synthesis:

- Test 1: Check the example condition provided by the specification.
- Test 2: 6-word sequence and asynchronous RESET.
- Test 2bis: Consecutive processing with RESET on the first.
- Test 3: Max length sequence (RAM(0) = '11111111').
- Test 3bis: Zero-length sequence (RAM(0) = '00000000').
- Test 4: Double processing on the same RAM.
- Test 5: Encodes 3 streams one after the other.

6 CONCLUSIONS

The model created largely satisfies the requirements proposed by the specification: it does not have a latch, and the clock period is less than 100 ns. The specification was very clear, and it was not difficult to implement the module, following what was explained in class and also thanks to excellent teamwork. After a few initial difficulties related to the functioning of the program used, Vivado, we encountered few problems in writing the code due to precise and effective design. Testing also went smoothly. As a result, we were able to spend the last hours dedicated to the project optimizing some states and ensuring that we had not left anything pending, and we were ultimately satisfied with the result obtained.

The realization of this project was very interesting and allowed us to refine what we had learned in the course from a practical point of view. Since it was done in pairs, it was also useful to improve our ability to work in a group and communicate, which was essential for the correct and effective performance of the test.