2018

Hardware LZ4 Decompressor Project Report

COE405 PROJECT REPORT

FOR DR AIMANE

FARIS HIJAZI S201578750

MOHAMMED BEJADI S201582510

Contents

Introduction

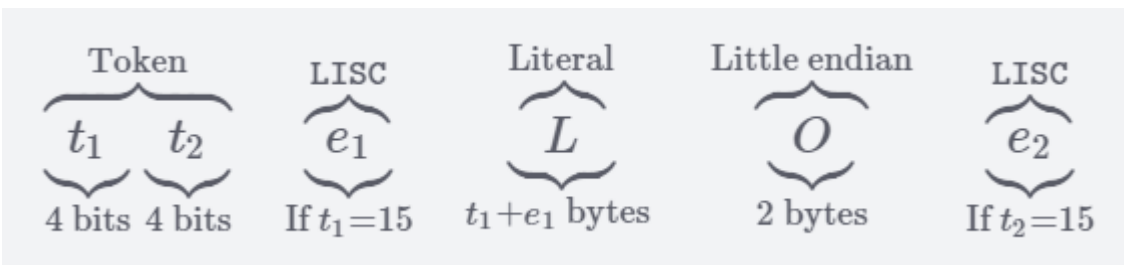
This report discusses designing and implementing LZ4 decompression algorithm in hardware using Verilog hardware description language.

LZ4 is lossless compression algorithm, providing compression speed at 400 MB/s. It features an extremely fast decoder, with speed in multiple GB/s.

LZ4 Block Format Description

LZ4 is an LZ77-type compressor with a fixed, byte-oriented encoding. An LZ4 compressed block is composed of sequences. A sequence is a suite of literals (not-compressed bytes), followed by a match copy. Each sequence starts with a token. The token is a one-byte value, separated into two 4-bits fields. Therefore, each field ranges from 0 to 15. [1]

A block looks like this:



LZ4 Sequence

Token : ==> 4-high-bits : literal length / 4-low-bits : match length

Token	*Literal length+ (optional)*	Literals	Offset	*Match length+ (optional)*
1-byte	0-n bytes	0-L bytes	2-bytes (little endian)	0-n bytes

1 Design

We've focused on designing the Datapath and control unit. We first drew schematics and then focused on implementing it using Verilog. For the Datapath, we first looked at the requirements and mapped them to specifications, this allowed us to choose what components are needed.

In the algorithm, we need to read, store and keep track of some values, those are depicted below:

- **Offset**: Indicates how far behind the MatchPointer should be relative to the WritePointer
- **MatchLength**: Keeps track of how many literals in the match were written to the output buffer. It is decremented every write cycle
- **LiteralLength**: Keeps track of the length of the literal
- **WritePointer**: Indicates which address to write to in the output buffer
- **MatchPointer**: A pointer to the output buffer, used to indicate which literal to copy

For each of the values (*Offset, MatchLength, LiteralLength, WritePointer, MatchPointer*) there is a special register added with functionality (such as *add*, *load, increment*) depending on what is needed.

We have gone over and reviewed the LZ4 algorithm and how it works. We used software tools to help, by reviewing some of the code, testing many examples, and consulting [professor Aiman H. El-Maleh](#).

1.1 Modified LZ4 Convention

The LZ4 algorithm is made to be used by high-level software, in the basic implementation, the compressed data would include a length field in the beginning, this helps the decompressor by informing it with the size of the data. In hardware however, this is not synthesizable, so we have added our own convention to LZ4 so the decompressor could identify when each block is over.
The convention is as follows: having *Offset=0, and MatchLength=0*, indicates that the block has ended.

1.2 Datapath

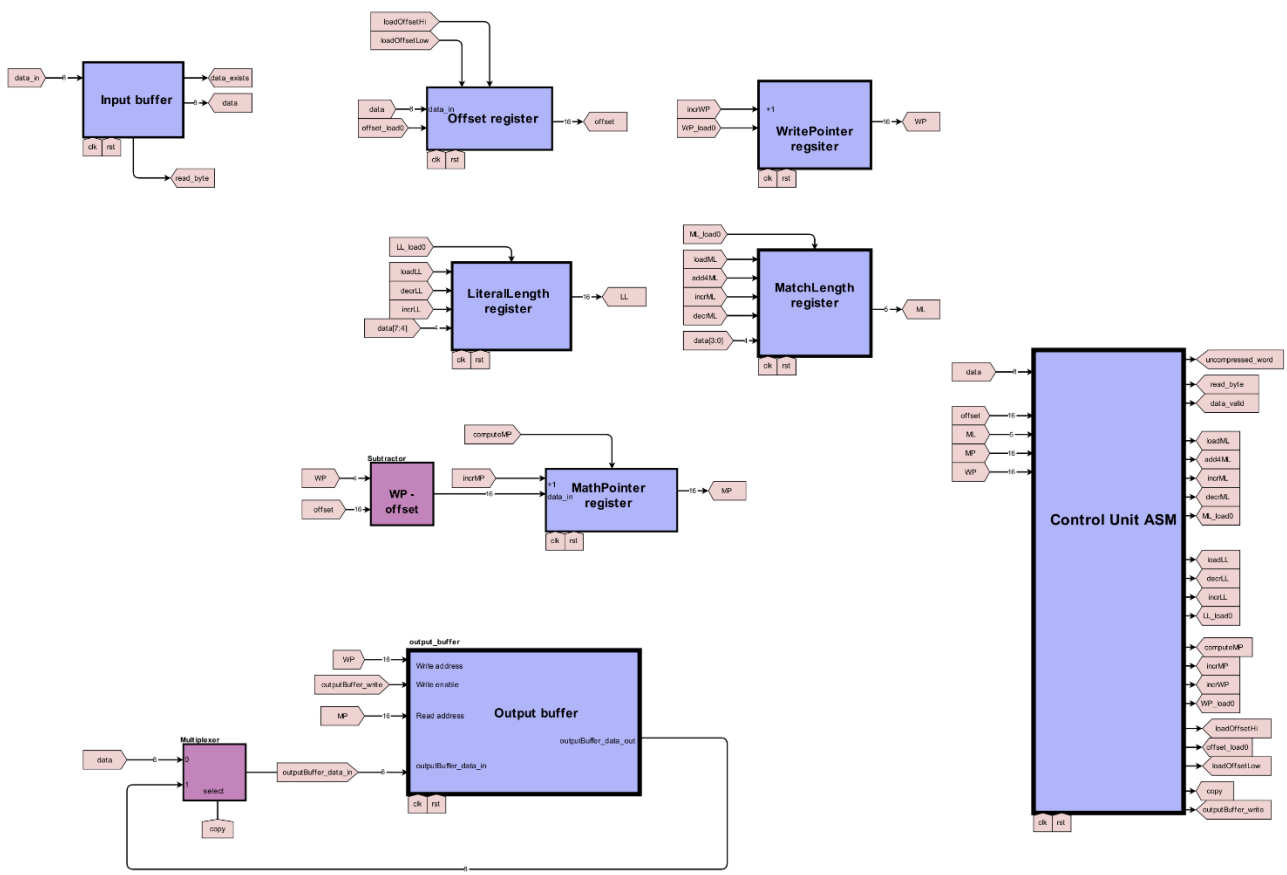


Figure 1 Decompressor Datapath schematic

This schematic was created using an online tool: digikey.com/schemeit.

The following subsections will explain and breakdown the Datapath components.

1.2.1 Input Buffer

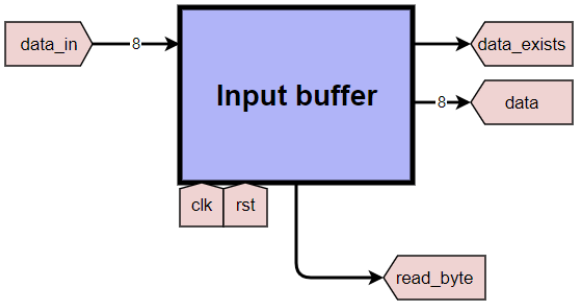


Figure 2 Input buffer

The input buffer mimics the queue data structure, it is used to buffer the incoming bytes until the decompressor decodes them. This is needed because the decompressor doesn't read the data as fast the data is being provided. The signal ***data_in*** is an external signal, the data is fed serially as bytes over clock cycles from the external user. The input buffer module contains pointers inside that keep track of the data that has been read and the data that has been written, from these pointers, it provides the ***data_exists*** signal, indicating that a full block is ready to be decompressed. The data can be requested by setting ***read_byte***=1.

1.2.2 Output Buffer

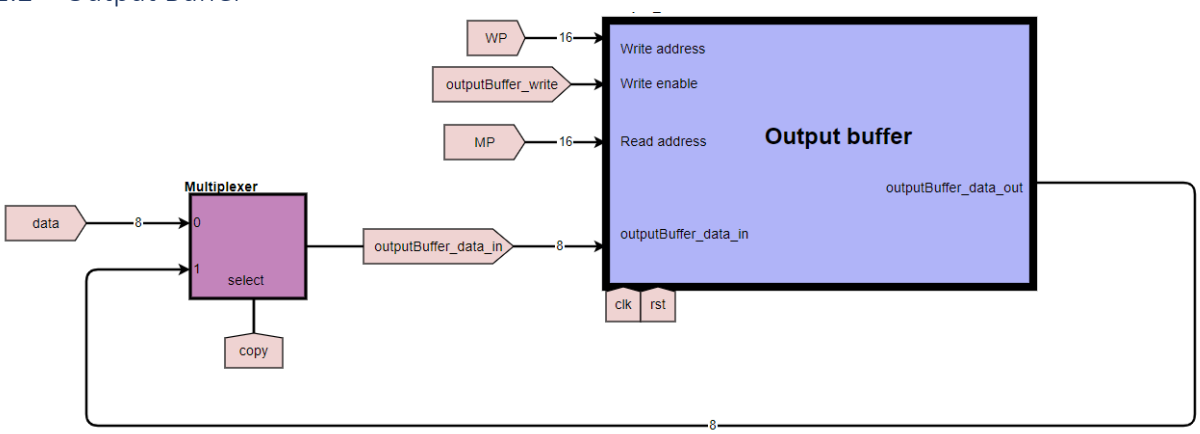


Figure 3 Output buffer

The output buffer is used to temperarely store the literals so that they can be copied. A multiplexer is used to choose between new writing literals from data, or to copy existing literals. Either way, literals are always written in the address of the write pionter (**WP**);

The match pointer (**MP**) is used read the literal to copy and written to the address of the write pointer (**WP**).

1.2.3 Offset Register

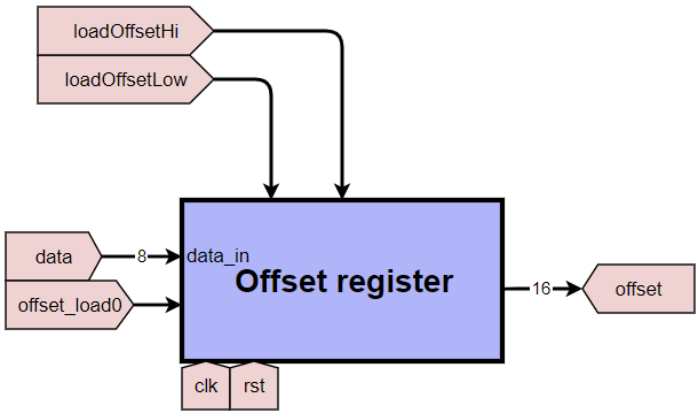


Figure 4 Offset register

The offset value indicates how far back the target match is behind the write pointer in the output buffer.

Signals and their affects:

- *incrWP:* *WP = WP + 1*
- *loadOffsetHi:* *offset[15:8] = data*
- *loadOffsetLow:* *offset[7:0] = data*
- *offset_load0:* *offset = 0*

1.2.4 Match Length Register

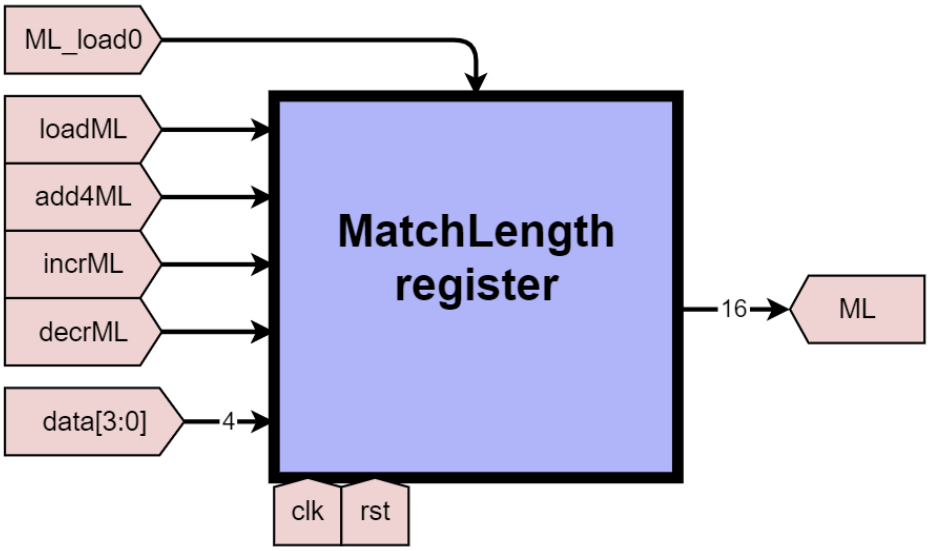


Figure 5 Match Length register

Signals and their affects

- *ML_load0:* *ML = 0*
- *loadML:* *ML = data[3:0]*
- *add4ML:* *ML = ML + 4*

- *incrML:* *ML = ML + 1*
- *decrML:* *ML = ML -1*

1.2.5 Literal Length Register

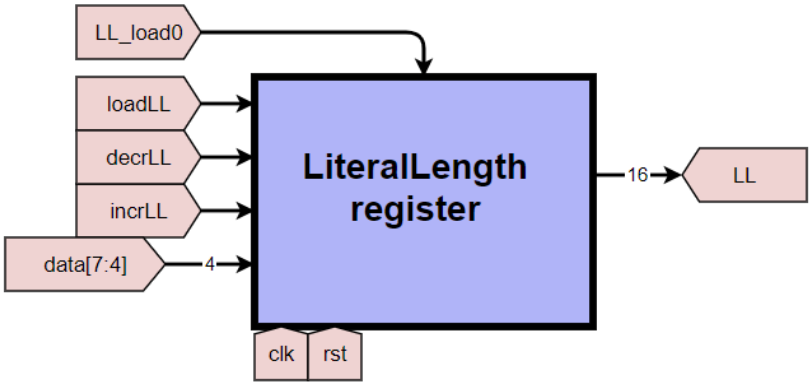


Figure 6 Literal Length register

Signals and their affects:

- *LL_load0:* *LL = 0*
- *loadLL:* *LL = data[7:4]*
- *decrLL:* *LL = LL - 1*
- *incrLL:* *LL = LL + 1*

1.2.6 Match Pointer Register

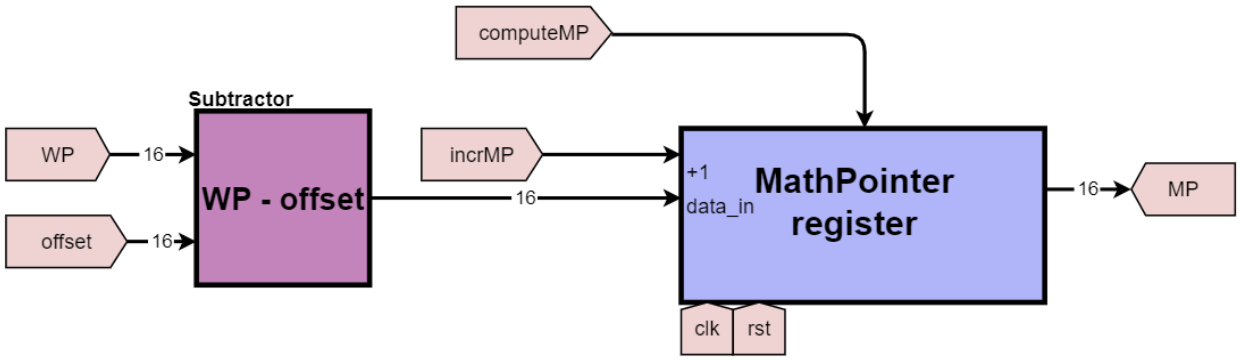


Figure 7 Match Pointer register

The match pointer (**MP**) is used to address the literals to read from the output buffer, this is done when copying literals. The value should be the write pointer (**WP**) subtracted by the offset.

In the phase of Copying matches, MP is computed, then WP and MP are incremented to go to copy the next literal.

$$MP = WP - offset$$

Signals and their affects:

- *incrMP:* *MP = MP + 1*

- *computeMP:* *MP = WP - offset*

1.2.7 Write Pointer

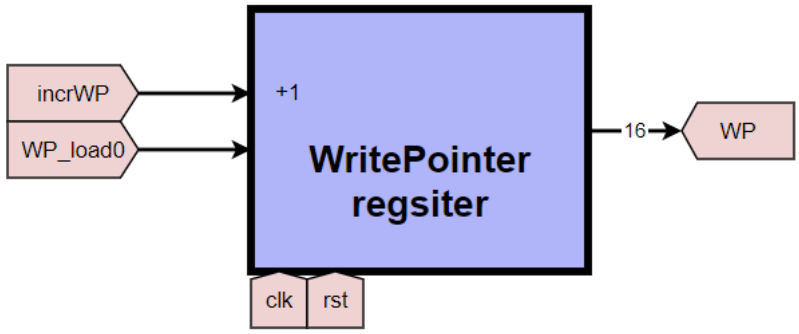


Figure 8 Write Pointer register

The write pointer (**WP**) is used to address the byte that will be written in the output buffer. It is first initialized by zero from the control unit (at the start of every block) and is incremented when writing or copying literals.

Signals and their affects:

- *incrWP:* *WP = WP + 1*
- *WP_load0:* *WP = 0*

1.2.8 Algorithmic State Machine (ASM)

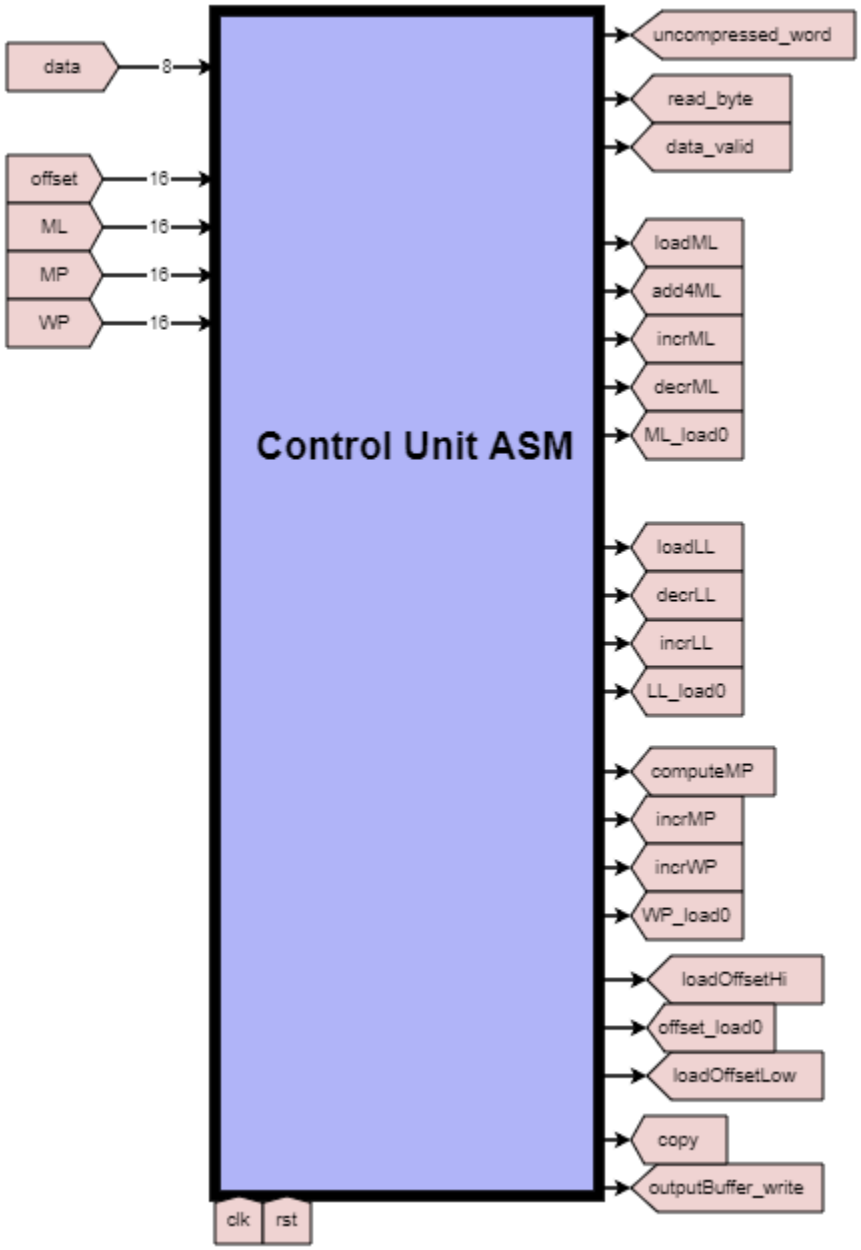


Figure 9 Datapath Algorithmic State Machine

The ASM synchronizes and manages all the other elements in the Datapath. It contains all the control signals and has the register values as inputs, those are needed to make decisions. Refer to figure// for a full diagram of the ASM.

1.3 Control Unit

Figure10 shows the Algorithmic State Machine Diagram (ASMD) of the Control Unit of the decompressor.

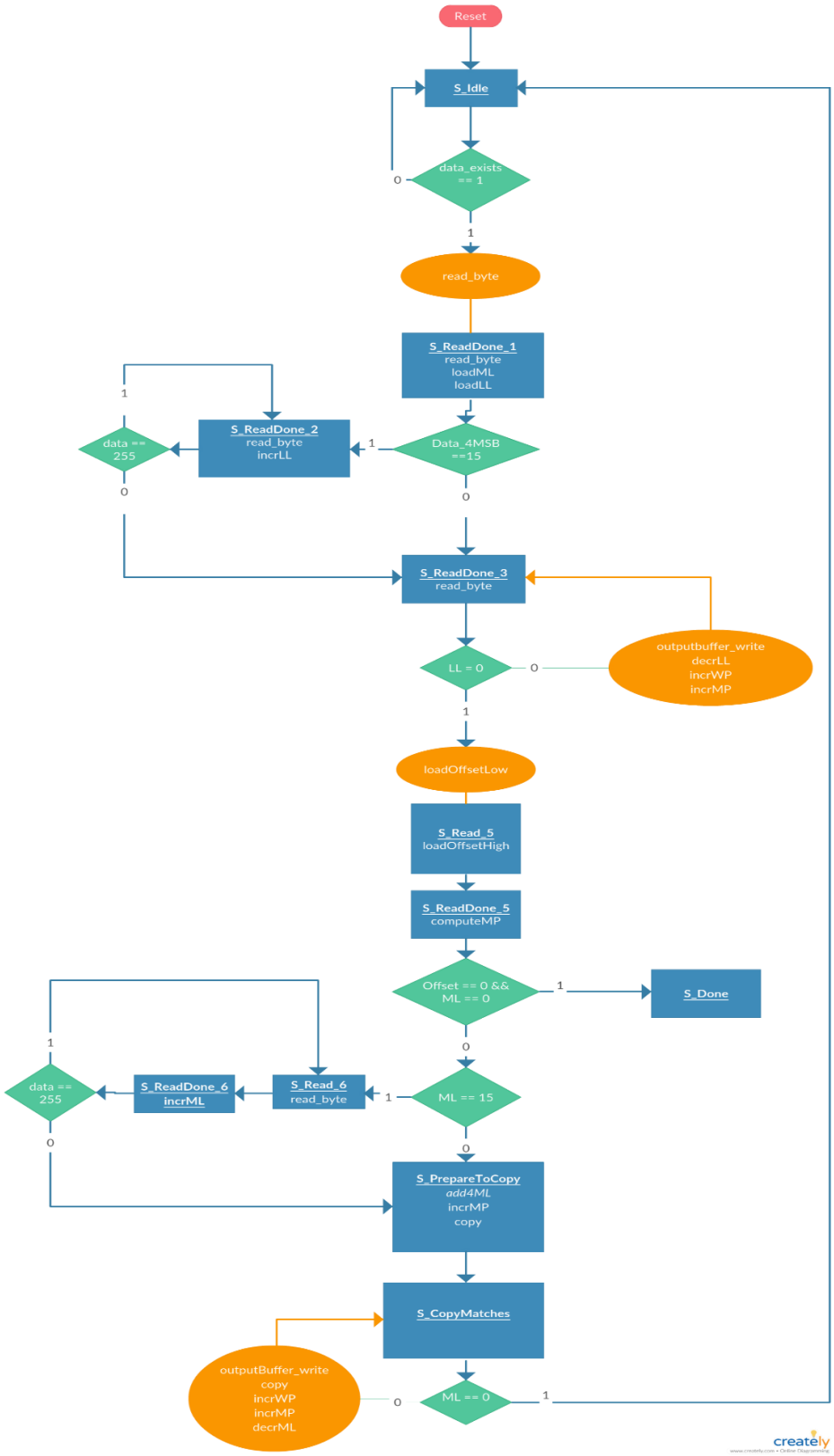


Figure 10 Control Unit ASMD

1.3.1 Waiting for data

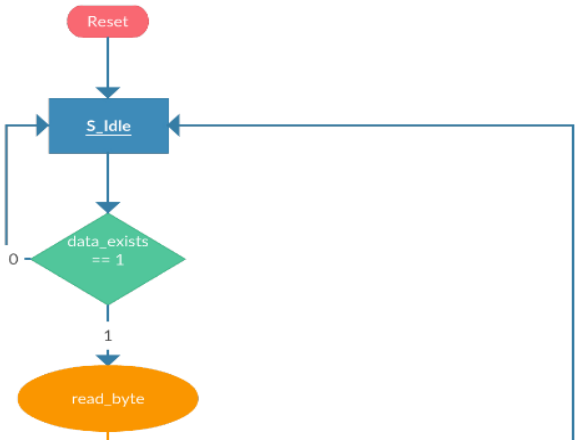


Figure 11 Waiting-for-data stage

Figure 11 shows the beginning of our ASM, as long as there's no data, stay in S_idle, when data exist, the decompression process starts with reading the first byte.

1.3.2 Accumulating literal length

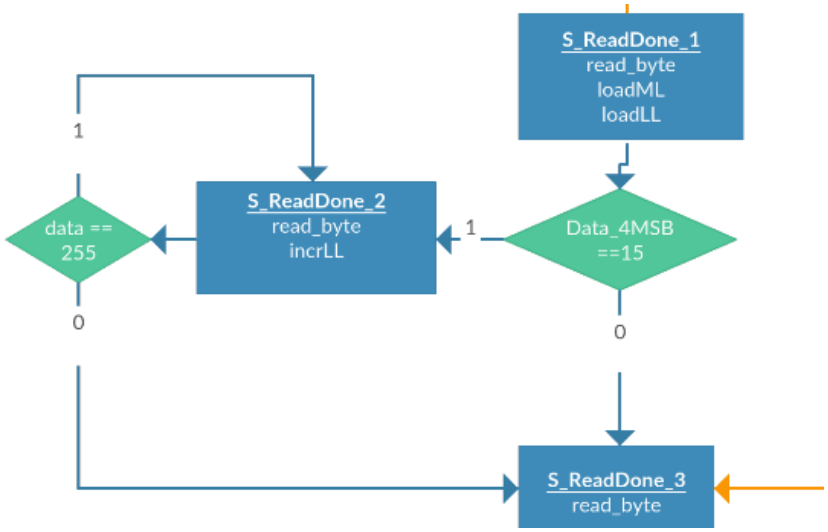


Figure 12 Accumulating-Literal-Length stage

After reading the first byte. Load signals are sent to load the data into its corresponding registers, at the same cycle, we check the 4 significant bits of the data, if it is equal to 15, we will have more bytes for the literal length. Therefore, we will accumulate it in a loop while checking the extra bytes if equal to 255 or not. When accumulating all the literal length bytes, we go to the next stage.

1.3.3 Writing literals

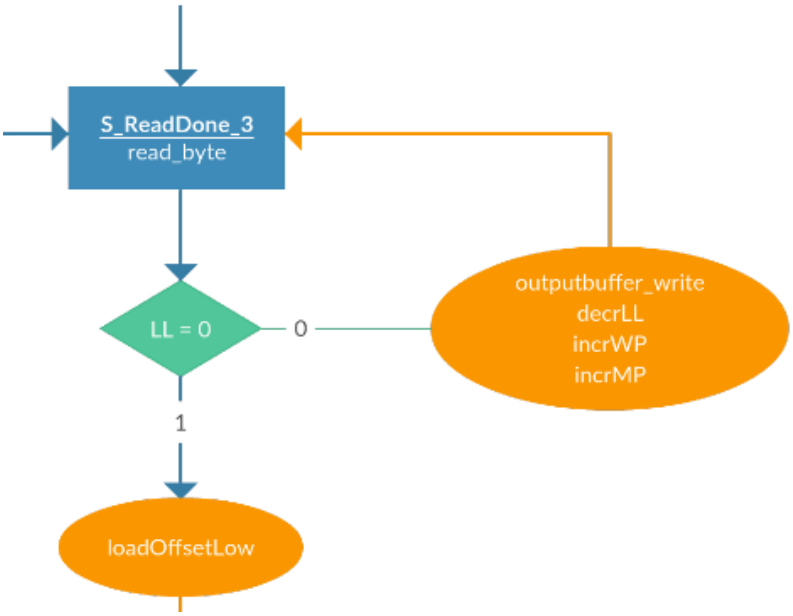


Figure 13 Writing-literals stage

The next stage is writing the literals, we read the byte and check whether the literal length is 0 or not, as long as it is not 0, we will write the byte as it is one of the literals and decrement the literal length by 1 and increment both the write and match pointers by 1. When the literal length is 0, we send a signal to load the low 8 bits of the offset with the data.

1.3.4 Checking for end of block

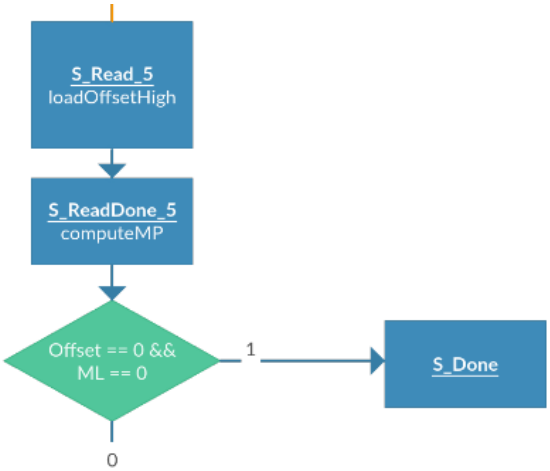


Figure 14 Check-for-end-of-block stage

Next, we must load the high part of the offset with the data. After going to the next stage, we will have the value of the offset, so we can send a signal to compute the match pointer.

However, we must check whether both the offset and match length are 0, which is our indication for the end of the block.

1.3.5 Accumulating match length

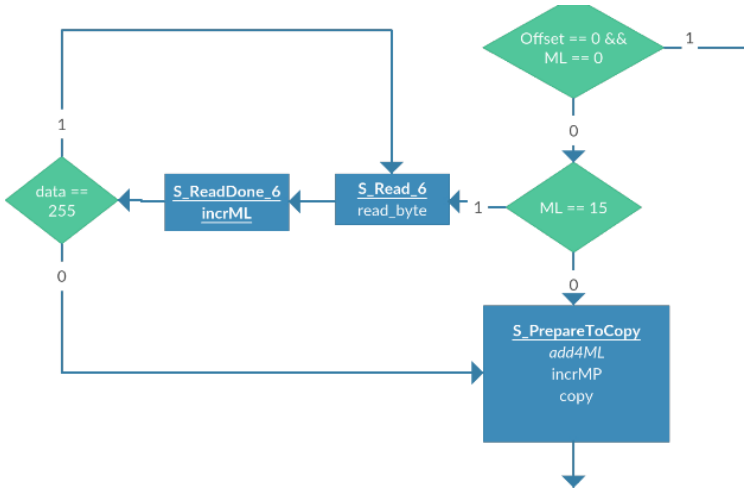


Figure 15 Accumulating-match-length stage

In case the match length is equal to 15, this indicates extra bytes exist for the length, so we will loop again to accumulate the extra bytes as long as the read byte (The extra one) is 255, when it is not. We go to the next stage to write the matched literals found.

1.3.6 Copying matches

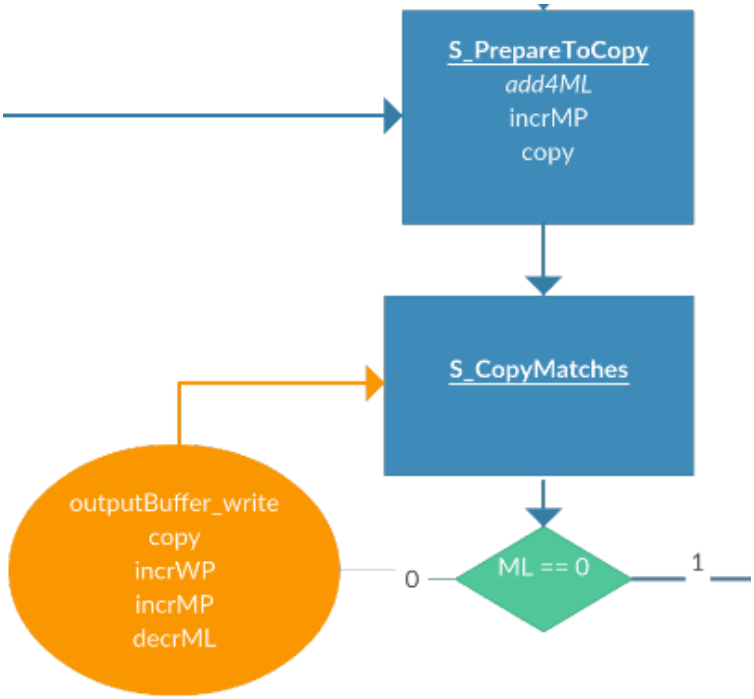


Figure 16 Copying-matches stage

The last stage is to copy the literals. We add the extra 4 to the match length and start copying and decrementing the match length as long as the match length is not 0. When it reaches 0, we go back to the idle state.

1.3.7 Signals

Signal	Effect
incrWP	*WP = WP + 1*
incrMP	*MP = MP + 1*
loadOffsetHigh	*offset[15:8] = data*
loadOffsetLow	*offset[7:0] = data*
decrLL	*LL = LL – 1*
loadLL	*LL = data[word_size-1:4]*
loadML	*ML = data[3:0]*
incrML	*ML = ML + data*
add4ML	*ML = ML + 4*
computeMP	*MP = WP - offset*
decrML	*ML = ML - 1*
incrLL	*LL = LL + data*

1.4 Testing process

For testing, we tried many test cases and simulated the output to ensure the correctness of the design. It is key to make sure that your test cases are sufficient and will pass through all states of the state machine. There are test cases where the literals are only copied, some where there are no matches (this helped fix the design by exposing a fault), cases where there are extra literal length bytes and match length bytes.

The following test cases follow the form:

Test blocks:

Input (HEX)	10 31 0100 10 32 0100 10 33 0100 00 0e00 10 31 0e00 10 32 0e00 02 0f00 03 0200 50 31 3131 3131 00 00
Input (DEC)	16 49 1 0 16 50 1 0 16 51 1 0 0 14 0 16 49 14 0 16 50 14 0 2 15 0 3 2 0 80 49 49 49 49 49 0 0
Ouput(ASCII)	11111 22222 33333 1111 12222 23333 311111 1111111 11111

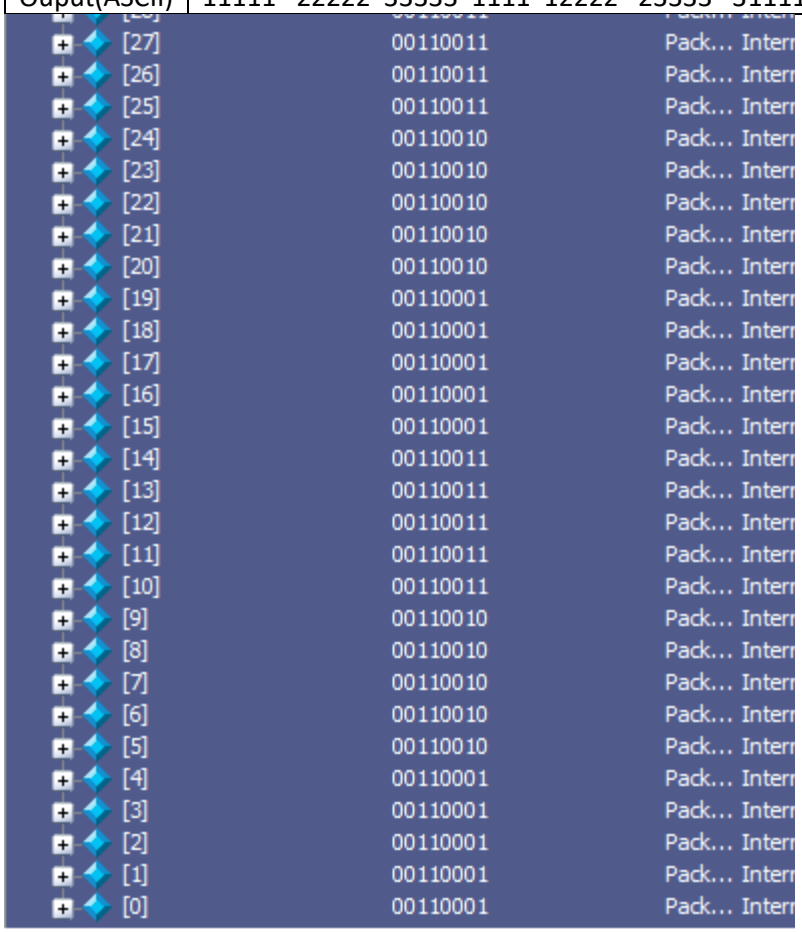
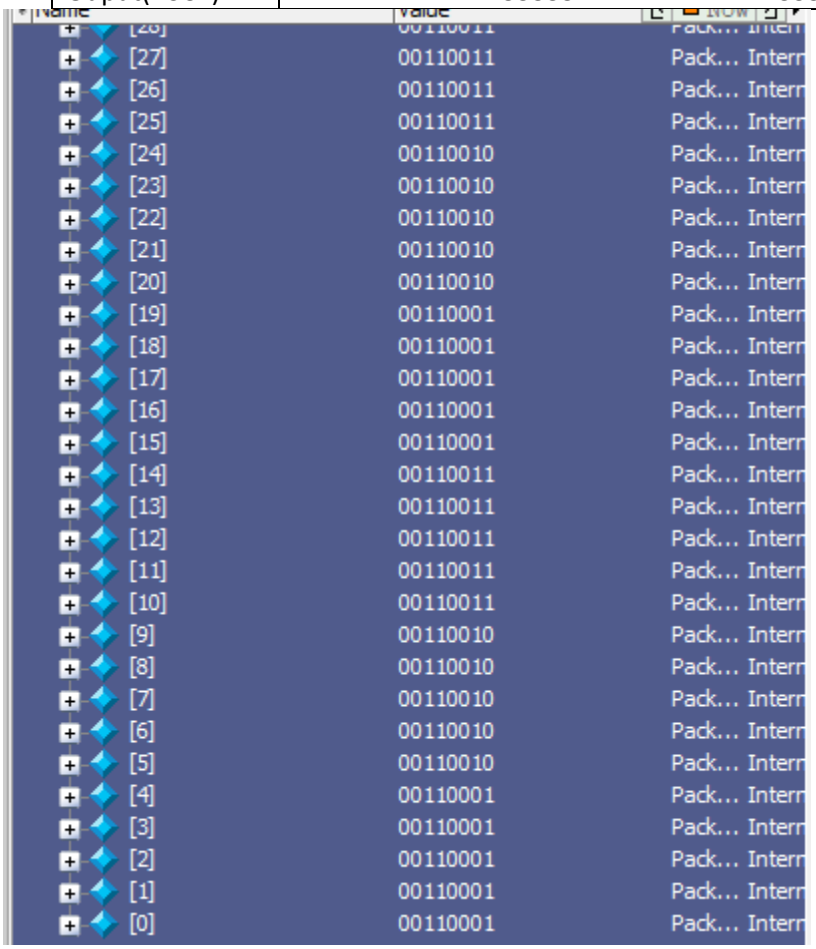


Figure 17 Output buffer content for test case 1

Input (HEX)	10 31 0100 10 32 0100 10 33 0100 00 0e00 b0 31 3232 3232 3233 3333 3333 0000
Input (DEC)	16 49 1 0 16 50 1 0 16 51 1 0 0 14 0 176 49 50 50 50 50 50 51 51 51 51 51 0 0

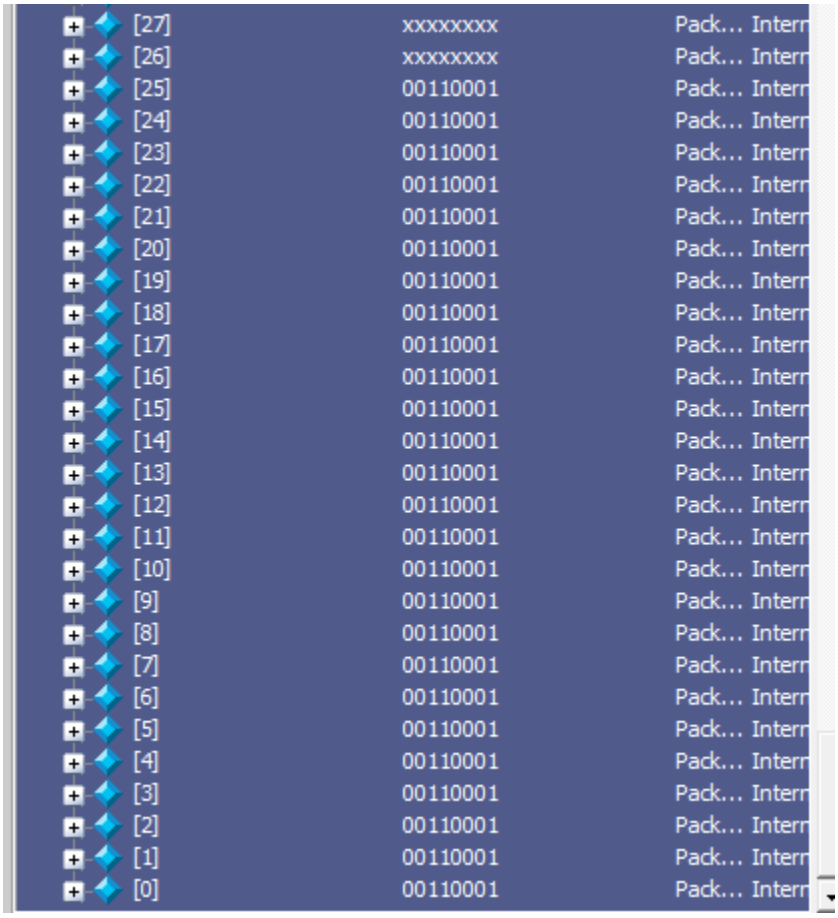
Ouput(ASCII)	11111 22222 33333 1111 12222233333



[27]	00110011	Pack... Intern
[26]	00110011	Pack... Intern
[25]	00110011	Pack... Intern
[24]	00110010	Pack... Intern
[23]	00110010	Pack... Intern
[22]	00110010	Pack... Intern
[21]	00110010	Pack... Intern
[20]	00110010	Pack... Intern
[19]	00110001	Pack... Intern
[18]	00110001	Pack... Intern
[17]	00110001	Pack... Intern
[16]	00110001	Pack... Intern
[15]	00110001	Pack... Intern
[14]	00110011	Pack... Intern
[13]	00110011	Pack... Intern
[12]	00110011	Pack... Intern
[11]	00110011	Pack... Intern
[10]	00110011	Pack... Intern
[9]	00110010	Pack... Intern
[8]	00110010	Pack... Intern
[7]	00110010	Pack... Intern
[6]	00110010	Pack... Intern
[5]	00110010	Pack... Intern
[4]	00110001	Pack... Intern
[3]	00110001	Pack... Intern
[2]	00110001	Pack... Intern
[1]	00110001	Pack... Intern
[0]	00110001	Pack... Intern

Figure 18 Output buffer content for test case 2

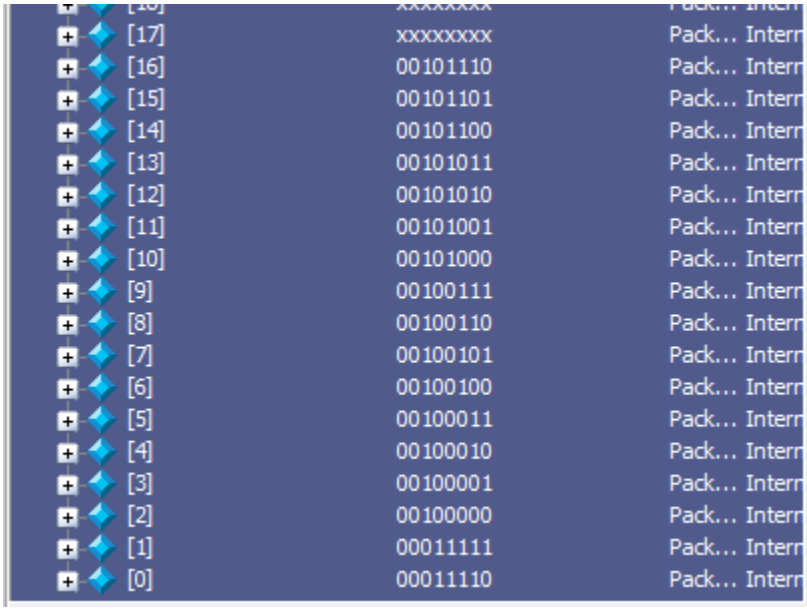
Input (HEX)	1f 31 0100 01 50 3131 3131 31
Input (DEC)	31 49 1 0 1 80 49 49 49 49 49 0 0
Ouput(ASCII)	1 11111 11111 11111 11111 11111



[27]	xxxxxxxx	Pack... Interr
[26]	xxxxxxxx	Pack... Interr
[25]	00110001	Pack... Interr
[24]	00110001	Pack... Interr
[23]	00110001	Pack... Interr
[22]	00110001	Pack... Interr
[21]	00110001	Pack... Interr
[20]	00110001	Pack... Interr
[19]	00110001	Pack... Interr
[18]	00110001	Pack... Interr
[17]	00110001	Pack... Interr
[16]	00110001	Pack... Interr
[15]	00110001	Pack... Interr
[14]	00110001	Pack... Interr
[13]	00110001	Pack... Interr
[12]	00110001	Pack... Interr
[11]	00110001	Pack... Interr
[10]	00110001	Pack... Interr
[9]	00110001	Pack... Interr
[8]	00110001	Pack... Interr
[7]	00110001	Pack... Interr
[6]	00110001	Pack... Interr
[5]	00110001	Pack... Interr
[4]	00110001	Pack... Interr
[3]	00110001	Pack... Interr
[2]	00110001	Pack... Interr
[1]	00110001	Pack... Interr
[0]	00110001	Pack... Interr

Figure 19 Output buffer content for test case 3

Input (HEX)	f0 02 3031 3233 3435 3637 3839 6162 6364 6566 67 00
Input (DEC)	240 2 4849 5051 5253 5455 5657 9798 99100 101102 103 00
Ouput(ASCII)	0123456789abcdefg



[17]	xxxxxxxx	Pack... Intern
[16]	00101110	Pack... Intern
[15]	00101101	Pack... Intern
[14]	00101100	Pack... Intern
[13]	00101011	Pack... Intern
[12]	00101010	Pack... Intern
[11]	00101001	Pack... Intern
[10]	00101000	Pack... Intern
[9]	00100111	Pack... Intern
[8]	00100110	Pack... Intern
[7]	00100101	Pack... Intern
[6]	00100100	Pack... Intern
[5]	00100011	Pack... Intern
[4]	00100010	Pack... Intern
[3]	00100001	Pack... Intern
[2]	00100000	Pack... Intern
[1]	00011111	Pack... Intern
[0]	00011110	Pack... Intern

Figure 20 Output buffer content for test case 4

2 Issues and Design Decisions

2.1 Design Decisions

2.1.1 Synchronous/asynchronous reset

Making the reset synchronous or asynchronous didn't impact the design much, so it was decided that synchronous reset would be simpler to synthesize and takes less resources.

2.1.2 Synchronous/asynchronous memory read

Having asynchronous memory read would have made the design much easier, however due to the limitations of the FPGA, we had to use synchronous memory reading for both the input and output buffers. The reason is the it would be much cheaper to synthesize synchronous reading, as there are already block RAMs in the FPGA, however using asynchronous memory reading would consume many of the FPGA lookup-tables, the resources simply aren't available to use such an approach.

2.1.3 Bonus functionality

Some concepts that didn't make it to the design, or would make it in the next version

- Having the input buffer be a circular buffer, this would allow for the system to continuously work despite the size of the buffer (assuming that on average, the *input data rate* <= *decompression data rate*)

2.2 Issues faced

Most of the issues faced were due to incorrect timing between reading and writing the data to the input and output buffers. It was tricky since requesting the data from the input buffer on one clock cycle

means that the data will come in the next cycle. We had to carefully time the signals so that the data would arrive at the correct time.

We had an issue with implementing the control unit, as we had mixed both synchronous and asynchronous components (combinational and sequential) in the *always* block. We then went over it and distinguished between the two types of signals and separated them.

3 Conclusion

We have designed and implemented the control unit and Datapath of a LZ4 decompressor. We've also simulated and verified the results to ensure correctness. Issues, design ideas, and ideas that didn't make it to the design were also addressed.

4 References

1. https://ieeexplore.ieee.org/document/7440278
2. Datapath online schematic: https://www.digikey.com/schemeit/project/lz4-decompressor-LIJ5K984006G/
3. Control Unit diagram: https://creately.com/diagram/jp2n42602/vD8r2aKbcIJxpNeomUlPHe6p1U%3D

5 Appendix A

Team contribution

	Member contribution percentage (%)	
Activity	**Faris Hijazi**	**Mohammed Bejadi**
Control Unit design	30	70
Datapath design	70	30
Report writing	60	40
Writing Verilog code	60	40
Writing buffers	100	0
Writing Control Unit	20	80
Debugging	40	60

6 Appendix B

This appendix is dedicated to the Verilog code used to implement and test the design.

6.1 Input buffer

```
`timescale 1ns / 1ps

module input_buffer #(parameter word_size=8, address_size=16)(
    output reg [word_size-1:0] data_out,
```

```
    output data_exists, // true when there is data that hasn't been read yet
    input [word_size-1:0] data_in,
    input read_byte, // outputs the next byte (if valid) and increments the read pointer
    input clk, write // write enable
    );

    parameter memory_size=2**address_size; // memory size is computed from the address size

    reg [address_size-1:0] read_pointer; // the address of the next word to be read
    reg [address_size-1:0] write_pointer; //
    reg [word_size-1:0] memory [memory_size-1:0]; // we have as many as memory_size bytes

    reg carry; // signal is never used, this is to avoid compiler warnings

    initial begin
        write_pointer = 0;
        read_pointer = 0;
    end

    assign data_exists = (read_pointer < write_pointer);

    always @ (posedge clk) begin
        if (write) begin
            memory[write_pointer] = data_in;
            {carry, write_pointer} = write_pointer + 1; // carry is never used
        end
        if (read_byte && data_exists) begin
            data_out = memory[read_pointer];
            read_pointer = read_pointer + 1;
        end
    end

endmodule
```

6.2 Output buffer

```
`timescale 1ns / 1ps

module output_buffer #(parameter word_size=8, address_size=4)(
    output reg [word_size-1:0] data_out,
    input [word_size-1:0] data_in,
    input [address_size-1:0] address_r, // for reading
    input [address_size-1:0] address_w, // for writing
    input clk, write);

    parameter memory_size=2**address_size;

    reg [ word_size-1:0] memory[memory_size-1:0];


    always @ (posedge clk) begin
        if (write) memory[address_w] = data_in;
        data_out = memory[address_r];
    end

endmodule
```

6.3 Control Unit

```
`timescale 1ns / 1ps
module ControlUnit #(parameter word_size = 8) (
        output  [word_size-1:0] uncompressed_word,
        output reg read_byte, // for the input_buffer
        output data_valid, // indicates if uncompressed_word is valid
        input [word_size-1:0] data, // a compressed word coming from the input buffer
        input clk,
        input data_exists, // signal from the input buffer indicating if the data is valid
        input reset // active high
    );
```

```
11
12      parameter S_idle =          4'b0000; // 0
13      parameter S_ReadDone_1 =    4'b0001; // 2
14      parameter S_ReadDone_2 =    4'b0010; // 3
15      parameter S_ReadDone_3 =    4'b0011; // 4
16      parameter S_Read_5 =        4'b0100; // 5
17      parameter S_ReadDone_5 =    4'b0101; // 6
18      parameter S_Done =          4'b0110; // 7
19      parameter S_Read_6 =        4'b0111; // 8
20      parameter S_ReadDone_6 =    4'b1000; // 9
21      parameter S_PrepareToCopy = 4'b1001; // a
22      parameter S_CopyMatches =   4'b1010; // c
23
24      parameter address_size = 16;
25
26
27      reg [3:0] next_state, current_state;
28
29
30      reg [address_size-1:0] MP, WP;
31      reg [address_size-1:0] LL, ML;
32      reg outputBuffer_write;
33      reg copy;
34
35      // offset reg
36      reg [15:0] offset;
37
38      wire [word_size-1:0] outputBuffer_data_in;
39      wire [word_size-1:0] outputBuffer_data_out;
40
41
42      reg incrWP;
43      reg incrMP;
44      reg loadOffsetHigh;
```

```
45      reg loadOffsetLow;
46      reg decrLL;
47      reg loadLL, loadML;
48      reg incrML;
49      reg add4ML;
50      reg computeMP;
51      reg decrML;
52      reg incrLL;
53      assign uncompressed_word = outputBuffer_data_in;
54      assign data_valid = outputBuffer_write;
55
56      // output_buffer
57      assign outputBuffer_data_in = copy? outputBuffer_data_out: data;
58
59      output_buffer #(.word_size(word_size), .address_size(address_size)) output_buffer(
60          .data_out(outputBuffer_data_out),
61          .data_in(outputBuffer_data_in),
62          .address_r(MP),
63          .address_w(WP),
64          .clk(clk),
65          .write(outputBuffer_write)
66      );
67
68      always @(posedge clk, posedge reset) begin
69          if (reset) begin
70              current_state = S_idle;
71              ML = 0;
72              LL = 0;
73              WP = 0;
74              MP = 0;
75              incrWP = 0;
76              incrMP = 0;
77              offset = 0;
78
```

```
            copy = 0;
            decrLL = 0;
            loadOffsetLow= 0;
            loadOffsetHigh= 0;
            loadLL= 0;
            loadML= 0;
            incrLL= 0;
            incrML= 0;
            add4ML= 0;
            computeMP= 0;
            decrML= 0;
        end
        else begin
            current_state = next_state;
        end

        if(incrMP) begin
            MP = MP + 1;
        end
        if(incrWP) begin
            WP = WP + 1;
        end
        if (decrLL) begin
            LL = LL - 1;
        end
        if (loadOffsetLow) begin
            offset[7:0] = data;
        end
        if (loadOffsetHigh) begin
            offset[15:8] = data;
        end
        if (loadLL) begin
            LL = data[word_size-1:4];
        end
```

```
        if (loadML) begin
            ML = data[3:0];
        end
        if (incrLL) begin
            LL = LL + data;
        end
        if (incrML) begin
            ML = ML + data;
        end
        if (add4ML) begin
            ML = ML + 4;
        end
        if (computeMP) begin
            MP = WP - offset;
        end
        if (decrML) begin
            ML = ML - 1;
        end

    end


    always @(*) begin
        read_byte = 0;
        outputBuffer_write = 0;
        copy = 0;
        decrLL = 0;
        loadOffsetLow= 0;
        loadOffsetHigh= 0;
        loadLL= 0;
        loadML= 0;
        incrLL= 0;
        incrML= 0;
        add4ML= 0;
```

```
        computeMP= 0;
        decrML= 0;
        incrWP = 0;
        incrMP = 0;

        case (current_state)
            S_idle:
                if (data_exists)begin
                    read_byte = 1;
                    next_state = S_ReadDone_1;
                end
                else
                    next_state = S_idle;

            S_ReadDone_1: begin

                loadML = 1;
                loadLL = 1;
                read_byte = 1;
                if (data[word_size-1:4] == 15) begin// if the LL is 15
                    next_state = S_ReadDone_2;
                end
                else begin
                    next_state = S_ReadDone_3;
                end
            end
            S_ReadDone_2: begin
                read_byte = 1;
                incrLL = 1;
                if (data == 255) begin
                    next_state = S_ReadDone_2; // loop (go to same state)
                end
                else begin
                    next_state= S_ReadDone_3;
```

```
            end
        end

        S_ReadDone_3: begin// Read Byte
            read_byte = 1;

            if (LL == 0) begin
                next_state = S_Read_5;
                loadOffsetLow = 1;
            end
            else begin// Loop and Write in buffer
                outputBuffer_write = 1;
                decrLL = 1;
                incrWP = 1;
                incrMP = 1;

                next_state = S_ReadDone_3;
            end
        end

        S_Read_5: begin// Read Byte
            loadOffsetHigh = 1;
            next_state = S_ReadDone_5;
        end

        S_ReadDone_5: begin
            computeMP = 1;
            if (offset == 0 && ML == 0)
                next_state = S_Done;
            else if (ML == 15)
                next_state = S_Read_6;
            else
                next_state = S_PrepareToCopy;
        end
```

```
215
216             S_Done: begin
217             end
218
219             S_Read_6: begin
220                 read_byte = 1;
221                 next_state = S_ReadDone_6;
222             end
223
224             S_ReadDone_6: begin
225                 incrML = 1;
226
227                 if (data == 255) begin
228                     next_state = S_Read_6;
229                 end
230                 else begin
231                     next_state = S_PrepareToCopy;
232                 end
233             end
234
235             S_PrepareToCopy: begin
236                 add4ML = 1;
237                 incrMP = 1;
238                 copy = 1;
239                 next_state = S_CopyMatches;
240             end
241
242             S_CopyMatches:
243                 if (ML == 0) begin
244                     next_state = S_idle;
245                 end
246                 else begin
247                     outputBuffer_write = 1;
248                     copy = 1;
```

```
                    incrWP = 1;
                    incrMP = 1;
                    decrML = 1;
                    next_state = S_CopyMatches;
                end
            default:
                next_state = S_idle;
        endcase
    end
endmodule
```

6.4 LZ4Decompressor (main module)

```
`timescale 1ns / 1ps
module LZ4Decompressor #(parameter word_size=8)(
        output [word_size-1:0] uncompressed_word,   //
        output data_valid,
        input [word_size-1:0] compressed_word,
        input write, clk, reset
    );

    wire [word_size-1:0] data;
    wire read_word;

    input_buffer #(.word_size(word_size), .address_size(16)) input_buffer (
        .data_out(data),
        .data_exists(data_exists),// true when there is data that hasn't been read yet
        .data_in(compressed_word),
        .read_byte(read_word), // outputs the next byte (if valid) and increments the read pointer
        .clk(clk),
        .write(write)
    );

    ControlUnit #(.word_size(word_size)) ControlUnit(
        .uncompressed_word(uncompressed_word),
        .read_byte(read_word),
        .data_valid(data_valid),
        .data(data),
        .clk(clk),
        .data_exists(data_exists),
        .reset(reset)
    );
endmodule
```

6.5 Test bench 1

```
`timescale 1ns / 1ps
module LZ4DecompressorTB ();
    parameter word_size = 8;
    // parameter address_size = 16;



    // Compressed data =    10 31 0100  10 32 0100  10 33 0100  00 0e00     10 31 0e00      10 32 0e00      02 0f00     03 0200         50 31 3131 3131 00 00
    // Compressed input:    16 49 1 0   16 50 1 0   16 51 1 0   0 14 0      16 49 14 0      16 50 14 0      2 15 0      3 2 0           80 49 49 49 49 49 0 0
    // Compressed Decimal = 11111       22222       33333       1111        12222           23333           311111      1111111         11111


    // 11111 22222 33333 1111 12222 23333 311111 1111111 11111


    reg clk, reset, data_exists;
    reg [word_size-1:0] compressed_word;
    reg write_en;
    wire [word_size-1:0] uncompressed_word;
    wire data_valid;

    LZ4Decompressor #(.word_size(word_size))  decompressor(
        .uncompressed_word(uncompressed_word),  //
        .data_valid(data_valid),
        .compressed_word(compressed_word),
        .write(write_en),
        .clk(clk),
        .reset(reset)
    );

    initial begin
        clk = 0 ; forever #10 clk = ~clk ;
    end

    initial begin
```

```
            reset = 1;
            @(negedge clk)
            reset = 0;

            @(negedge clk) // readbyte s1
            write_en = 1;
            compressed_word = 16;
            @(negedge clk) // Read_Done_3
            compressed_word = 49;
            @(negedge clk)
            compressed_word = 1;
            @(negedge clk)
            compressed_word = 0;
            @(negedge clk)

            compressed_word = 16;
            @(negedge clk)
            compressed_word = 50;
            @(negedge clk)
            compressed_word = 1;
            @(negedge clk)
            compressed_word = 0;
            @(negedge clk)


            compressed_word = 16;
            @(negedge clk)
            compressed_word = 51;
            @(negedge clk)
            compressed_word = 1;
            @(negedge clk)
            compressed_word = 0;
            @(negedge clk)

```

```
            compressed_word = 0;
            @(negedge clk)
            compressed_word = 14;
            @(negedge clk)
            compressed_word = 0;
            @(negedge clk)


            compressed_word = 16;
            @(negedge clk)
            compressed_word = 49;
            @(negedge clk)
            compressed_word = 14;
            @(negedge clk)
            compressed_word = 0;
            @(negedge clk)


            // 10 32 0e00
            // 23333
            // 16 50 14 0
            compressed_word = 16;
            @(negedge clk)
            compressed_word = 50;
            @(negedge clk)
            compressed_word = 14;
            @(negedge clk)
            compressed_word = 0;
            @(negedge clk)

            // 02 0f00
```

```
            // 311111
            // 2 15 0
            compressed_word = 02;
            @(negedge clk)
            compressed_word = 15;
            @(negedge clk)
            compressed_word = 0;
            @(negedge clk)



            // 03 0200
            // 1111111
            // 3 2 0

            compressed_word = 03;
            @(negedge clk)
            compressed_word = 02;
            @(negedge clk)
            compressed_word = 00;
            @(negedge clk)

            compressed_word = 80;
            @(negedge clk)
            compressed_word = 49;
            @(negedge clk)
            compressed_word = 49;
            @(negedge clk)
            compressed_word = 49;
            @(negedge clk)
            compressed_word = 49;
            @(negedge clk)
            compressed_word = 49;
            @(negedge clk)
```

```
            compressed_word = 00;
            @(negedge clk)

            compressed_word = 00;

        end
    endmodule
```

6.6 Test bench 2

```
`timescale 1ns / 1ps



module LZ4DecompressorTB2 ();
    parameter word_size = 8;
    // parameter address_size = 16;


    // Compressed data =    10 31 0100          10  32 0100     10 33 0100      00 0e00         b0 31 3232 3232 3233 3333 3333 0000
    // Decompressed data = 11111               22222           33333           1111            12222233333
    // decimal input:       16  49  1   0       16  50  1   0   16  51  1   0   0   14  0       176 49  50  50  50  50  50  51  51  51  51  51  0   0

    // 11111 22222 33333 1111 12222 23333 311111 1111111 11111

    reg clk, reset, data_exists;
    reg [word_size-1:0] compressed_word;
    reg write_en;
    wire [word_size-1:0] uncompressed_word;
    wire data_valid;

```

```
22          LZ4Decompressor #(.word_size(word_size))  decompressor(
23              .uncompressed_word(uncompressed_word),  //
24              .data_valid(data_valid),
25              .compressed_word(compressed_word),
26              .write(write_en),
27              .clk(clk),
28              .reset(reset)
29          );
30
31          initial begin
32              clk = 0 ; forever #10 clk = ~clk ;
33          end
34
35          initial begin
36              reset = 1;
37              @(negedge clk)
38              reset = 0;
39
40              @(negedge clk) // readbyte s1
41              write_en = 1;
42              compressed_word = 16;
43              @(negedge clk) // Read_Done_3
44              compressed_word = 49;
45              @(negedge clk)
46              compressed_word = 1;
47              @(negedge clk)
48              compressed_word = 0;
49              @(negedge clk)
50
51              compressed_word = 16;
52              @(negedge clk)
53              compressed_word = 50;
54              @(negedge clk)
```

```
55              compressed_word = 1;
56              @(negedge clk)
57              compressed_word = 0;
58              @(negedge clk)
59
60
61              compressed_word = 16;
62              @(negedge clk)
63              compressed_word = 51;
64              @(negedge clk)
65              compressed_word = 1;
66              @(negedge clk)
67              compressed_word = 0;
68              @(negedge clk)
69
70
71
72              compressed_word = 0;
73              @(negedge clk)
74              compressed_word = 14;
75              @(negedge clk)
76              compressed_word = 0;
77              @(negedge clk)
78
79
80              compressed_word = 176;
81              @(negedge clk)
82              compressed_word = 49;
83              @(negedge clk)
84              compressed_word = 50;
85              @(negedge clk)
86              compressed_word = 50;
87              @(negedge clk)
```

```
88
89
90              compressed_word = 50;
91              @(negedge clk)
92              compressed_word = 50;
93              @(negedge clk)
94              compressed_word = 50;
95              @(negedge clk)
96              compressed_word = 51;
97              @(negedge clk)
98              compressed_word = 51;
99              @(negedge clk)
100             compressed_word = 51;
101             @(negedge clk)
102             compressed_word = 51;
103             @(negedge clk)
104             compressed_word = 51;
105             @(negedge clk)
106
107             compressed_word = 00;
108             @(negedge clk)
109
110             compressed_word = 00;
111
112         end
113     endmodule
```

6.7 Test bench 3

```
1 `timescale 1ns / 1ps
2
3
4
5 module LZ4DecompressorTB3 ();
```

```
    parameter word_size = 8;
    // parameter address_size = 16;


    // Compressed data =    1f 31 0100 01                                50  3131 3131 31
    // Decompressed data =  1   11111   11111   11111   11111           11111
    // decimal
input:       31  49  1   0   1                           80  49  49  49  49  49  0   0

    // 11111 22222 33333 1111 12222 23333 311111 1111111 11111

    reg clk, reset, data_exists;
    reg [word_size-1:0] compressed_word;
    reg write_en;
    wire [word_size-1:0] uncompressed_word;
    wire data_valid;

    LZ4Decompressor #(.word_size(word_size))  decompressor(
        .uncompressed_word(uncompressed_word),  //
        .data_valid(data_valid),
        .compressed_word(compressed_word),
        .write(write_en),
        .clk(clk),
        .reset(reset)
    );

    initial begin
        clk = 0 ; forever #10 clk = ~clk ;
    end

    initial begin
        reset = 1;
        @(negedge clk)
```

```
        reset = 0;
        @(negedge clk) // readbyte s1
        write_en = 1;
        compressed_word = 31;
        @(negedge clk) // Read_Done_3
        compressed_word = 49;
        @(negedge clk)
        compressed_word = 1;
        @(negedge clk)
        compressed_word = 0;
        @(negedge clk)

        compressed_word = 1;
        @(negedge clk)
        compressed_word = 80;
        @(negedge clk)
        compressed_word = 49;
        @(negedge clk)
        compressed_word = 49;
        @(negedge clk)


        compressed_word = 49;
        @(negedge clk)
        compressed_word = 49;
        @(negedge clk)
        compressed_word = 49;
        @(negedge clk)
        compressed_word = 0;
        @(negedge clk)


```

```
        compressed_word = 0;

    end
endmodule
```

6.8 Test bench 4

```
`timescale 1ns / 1ps



module LZ4DecompressorTB4 ();

    parameter word_size = 8;

    // parameter address_size = 16;

    // 0123456789abcdefg 00

    // Compressed data =    f0  02  3031    3233    3435    3637    3839    6162    6364    6566    67 00

    // Decompressed data =  0123456789abcdefg

    // decimal input:       240  2  4849    5051    5253    5455    5657    9798    99100   1011 02  103 00

    // 11111 22222 33333 1111 12222 23333 311111 1111111 11111

    reg clk, reset, data_exists;
```

```
17    reg [word_size-1:0] compressed_word;
18    reg write_en;
19    wire [word_size-1:0] uncompressed_word;
20    wire data_valid;
21
22    LZ4Decompressor #(.word_size(word_size))  decompressor(
23        .uncompressed_word(uncompressed_word),  //
24        .data_valid(data_valid),
25        .compressed_word(compressed_word),
26        .write(write_en),
27        .clk(clk),
28        .reset(reset)
29    );
30
31    initial begin
32        clk = 0 ; forever #10 clk = ~clk ;
33    end
34
35    initial begin
36        reset = 1;
```

```
37          @(negedge clk)
38          reset = 0;
39          @(negedge clk) // readbyte s1
40          write_en = 1;
41
42          // f0   02  0123456789abcdefg 00
43          compressed_word = 240;
44          @(negedge clk) // Read_Done_3
45          compressed_word = 2;
46          @(negedge clk)
47          compressed_word = 48;
48          @(negedge clk)
49          compressed_word = 49;
50          @(negedge clk)
51
52          compressed_word = 50;
53          @(negedge clk)
54          compressed_word = 51;
55          @(negedge clk)
56          compressed_word = 52;
```

```
57          @(negedge clk)
58          compressed_word = 53;
59          @(negedge clk)
60          compressed_word = 54;
61          @(negedge clk)
62          compressed_word = 55;
63          @(negedge clk)
64          compressed_word = 56;
65          @(negedge clk)
66          compressed_word = 57;
67          @(negedge clk)
68
69
70
71          compressed_word = 97; //a
72          @(negedge clk)
73          compressed_word = 98;
74          @(negedge clk)
75          compressed_word = 99;
76          @(negedge clk)
```

```
        compressed_word = 100;
        @(negedge clk)
        compressed_word = 101;
        @(negedge clk)
        compressed_word = 102;
        @(negedge clk)
        compressed_word = 103;
        @(negedge clk)
        compressed_word = 104;
        @(negedge clk)
        compressed_word = 105;
    end
endmodule
```