National Tsing Hua University
Department of Electrical Engineering
EE4292 IC Design Laboratory (積體電路設計實驗)
Fall 2018

Homework Assignment #4 **(15%)**

QR Code Encoder

Assigned on Nov 1, 2018
Due by Nov 15, 2018

I. Introduction

Fig. 1 Example of QR code (15-byte "www.nthu.edu.tw", test pattern no. 00)

QR code (Quick Response Code) [1] is a type of matrix barcode used for storing text information in a robust way. It is widely used for URL instant access. Its specification defines lots of variations including version 1-40 (21x21-177x177 pixel square), four character sets, and different levels for error correcting capability. In this assignment, you will implement the simplest QR code encoder: version 1 (21x21), 8-bit Byte mode (JIS8 code in Appendix A), and error correction level L (up to 17-byte text available) for error-free input images (no error correction is required). The number of total data codewords in this spec is 19.

II. GOAL of this assignment

In this assignment, a given QR code without data codewords will be embedded in an image with logo as its background. At first, you need to locate the QR code and its orientation. Then you will encode the data codewords and put it into QR code to form a complete one *i.e.* it can be decoded with any QR code software. Scan with your smart phone to prove it!

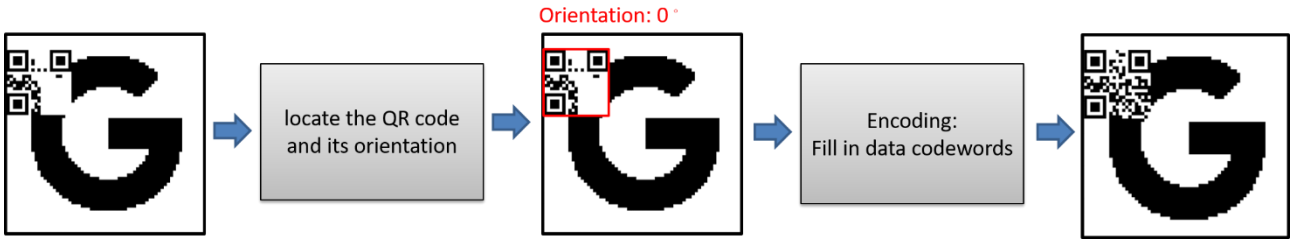


Fig. 2 Flow-chart of this assignment

III. Detailed flow for this assignment

A. Input structure in SRAM

The general structure for a QR code symbol of version 1 is shown in Fig. 3. It has four major components: position detection pattern, format/version information, data codewords, and error correction codewords. In this assignment, we will give you the QR code with position detection pattern, format/version information, and error correction codewords. Therefore, you will only need to fill in the correct data codewords.

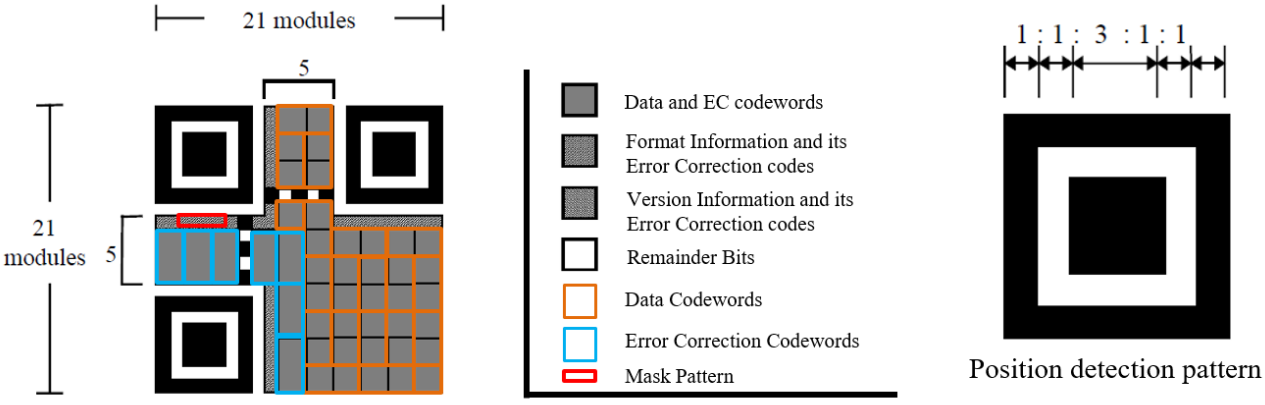


Fig. 3 Structure of a QR code version 1 symbol [1].

In this assignment, 21x21 QR code patterns are embedded into 64x64 images which have different logos as their background. The whole 64x64 image are stored in SRAM (simulated in testbench). The SRAM has a capacity of 1024 words and 4-bits word size. Within the SRAM, each word contains 4 pixels (as shown in Fig 4). Each pixel use 1'b1 for black and 1'b0 for white.(the gray pixel in the background is actually white in our test patterns).

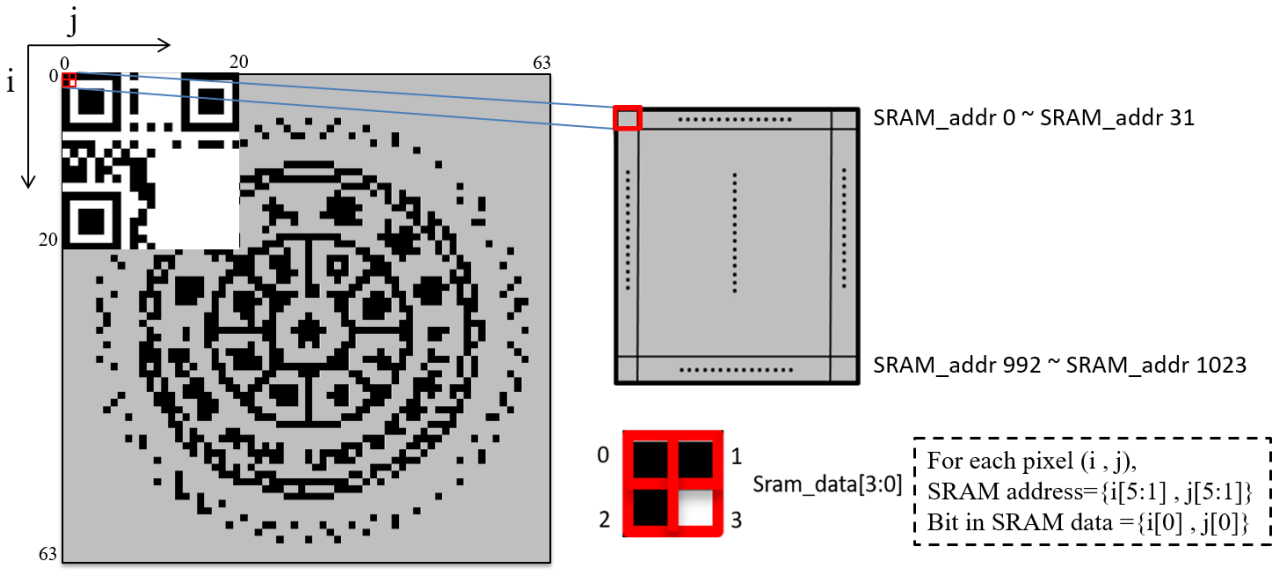


Fig. 4 Location of QR code in SRAM (1024x4b).

B. Locate the QR code and its orientation

The test patterns will put QR code into 64x64 image in arbitrary position, so you need to locate the target position for the QR code first. The target position and the orientation can be recognized with patterns on the three corners of the QR code. In this assignment, the QR code has only four possible orientations $\{0^\circ, 90^\circ, 180^\circ, 270^\circ\}$. The corresponding orientation of the QR code is shown in Fig. 5.

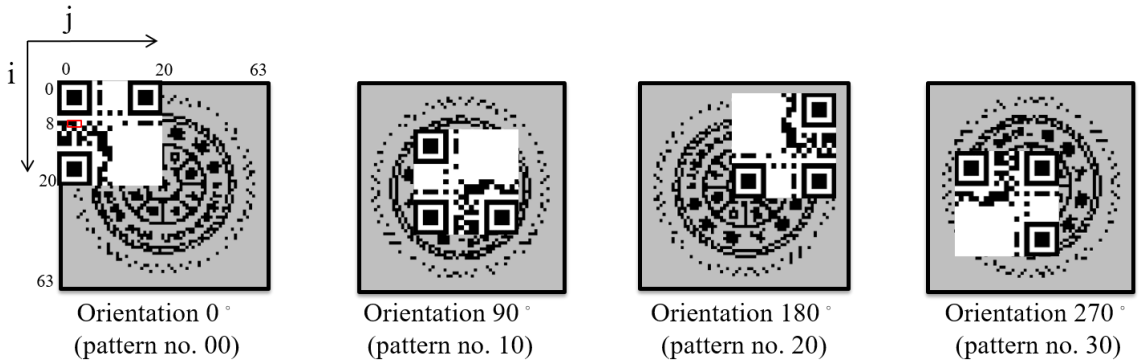


Fig. 5 Four QR code orientations for "www.nthu.edu.tw"

C. Encode data codewords

The data codewords can be divided into three parts: Mode indicator, Character count indicator, and Encoded data. The mode indicator indicates encoding mode. In this assignment, we only use Byte Mode, which means the mode indicator will always be a 4-bit code 4'b0100. The character count indicator represents the encoded character count (*e.g.* 15) using 8-bit unsigned integer. The encoded data is composed of the code of characters. Each character will be encoded with an 8-bit JIS8 code which can be found in JIS8 code table (Appendix A). The following explanation will use a 15-character text "www.nthu.edu.tw" as an example. Fig. 6 is part of components of data codewords.

Mode Indicator (Fixed)	Character Count Indicator	Encoded Data
0100	00001111	01110111 01110111 01110111 00101110.....01110100 01110111
		w w w . t w
		8 x 15 = 120 bits

Fig. 6 Part of components of data codewords

Because the number of data codewords is limited and can only be 19 and each of them is 8-bit. The total bit for the data codewords would be 19*8=152. If the data codewords (mode indicator, character count indicator, and encoded data) is less than 152, we will fill the remaining data codewords with the following rules:

1. Add **Terminator** 4'b0000 first
2. Add **Pad Bytes** if total bits are still less than 152 after adding Terminator:
   Method : repeating **11101100 00010001** until total bits are equal 152

After compensating enough bits with the rules above, the full components of data codewords are shown in Fig. 7.

Mode Indicator (Fixed)	Character Count Indicator	Encoded Data	Terminator	Pad Bytes
0100	00001111	01110111....01110111	0000	11101100 00010001

8 x 15 = 120 bits

Fig. 7 Full components of data codewords

The data codeword is generated by grouping each 8-bit from most-significant-bit (MSB) to least-significant-bit (LSB). And the final data codewords is shown in Fig. 8.

codeword0	codeword1	codeword2	codeword3	...	codeword15	codeword16	codeword17	codeword18
01000000	11110111	01110111	01110111	...	01000111	0111 0000	11101100	00010001

Terminator Pad Bytes

Fig. 8 Total 19 data codewords

Although the number of data codewords is 19 (8-bit for each), we repartition them into 20 blocks in order to make each block more meaningful and easier to implement. The first block and last block are 4-bit, and the others are 8-bit. The component of each block is shown in Fig. 9. And the position of each block on QR code is shown in Fig. 10.

block0	block1	block2	block3	...	block16	block17	block18	block19
0100	00001111	01110111	01110111	...	01110111	0000 1110	11000001	0001
Mode	Character Count	Encode word1	Encode word2	...	Encode word15	Encode word16	Encode word17	End

w.nthu.edu.t Terminator Pad Bytes

Fig. 9 All 20 blocks and its meaning

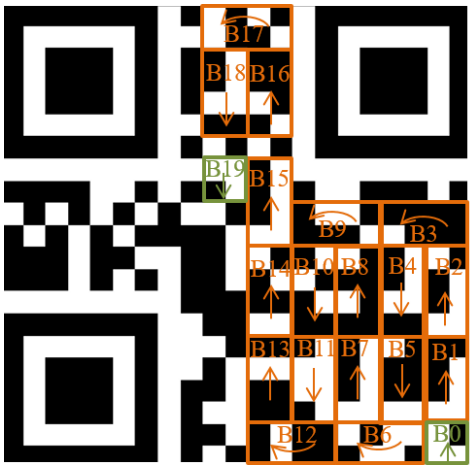


Fig. 10 position of all 20 blocks on QR code

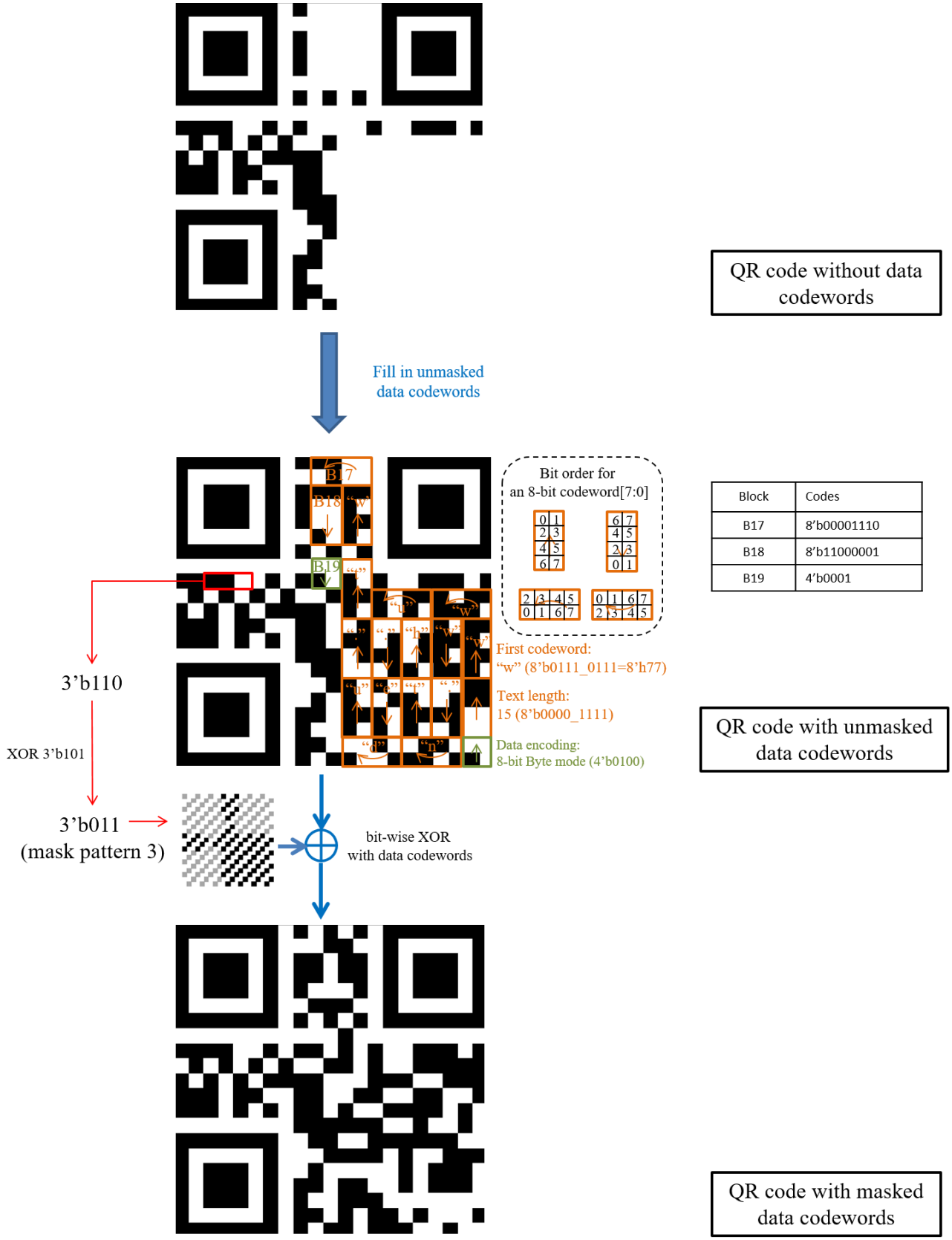


Fig. 11 Example of QR code encoding. Note that a black pixel stands for "1" and white for "0".

An encoding flow for this assignment is shown in Fig. 11. In this example, we encode 15-byte text "www.nthu.edu.tw". At first, we need to put the 19 data codewords (=20 blocks) into QR code in the order shown in Fig. 10. And then data codewords of a QR code symbol are masked (XORed) with one pre-defined pattern (*e.g.* 3'b011) for balancing the numbers of black and white pixels.

The 3-bit data in the red rectangle, positions (i,j)=(8,2-4) for orientation 0° (as shown in Fig. 11) should be masked(XOR) with a fixed 3'b101 to derive the mask pattern reference ID. There are eight mask patterns as listed in Fig. 12. The bit-mask for data codewords at position (i,j) is one if the corresponding condition is met (refer to Appendix C for details).

Mask Pattern Reference	Condition
000	$(i + j) \bmod 2 = 0$
001	$i \bmod 2 = 0$
010	$j \bmod 3 = 0$
011	$(i + j) \bmod 3 = 0$
100	$((i \div 2) + (j \div 3)) \bmod 2 = 0$
101	$(ij) \bmod 2 + (ij) \bmod 3 = 0$
110	$((ij) \bmod 2 + (ij) \bmod 3) \bmod 2 = 0$
111	$((ij) \bmod 3 + (i+j) \bmod 2) \bmod 2 = 0$

Fig. 12 Mask pattern generation conditions [1].

The sequence text to encode will be transmitted through *jis8_code* port after a one cycle pulse *get_jis8_code_start* is acknowledged. The *jis8_code* is accompanied by *jis8_code_end* to signify the last data being transmitted. Each character will only last one cycle on *jis8_code*. And each sequence text can be read for just one time.

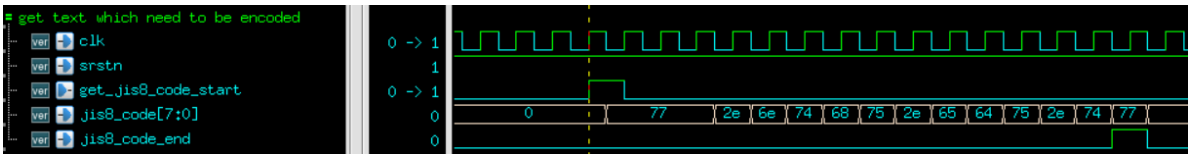


Fig. 13 waveform of get text signal (e.g. 15-byte text: www.nthu.edu.tw)

The result image(embedded with encoded QR code) should be written back to the same SRAM. In some circumstances, we may face the circumstances that only a few bits in a word should be modified, with the others remained. In such case, we use **write mask** to meet the requirement. The write mask indicates which bits of write_data would be written back to the word of the SRAM. The write mask function is shown in Table 1. "0" stands for write back, and "1" stands for not write back for each bit in write mask. Fig. 14 shows a simple example of write mask.

Write_Mask[i]	rw_enable	Function
X	1	Read only addressed memory location
1	0	Memory location unchanged
0	0	Write to addressed memory location

p.s. i is bit number (i = 0,1,2,3)

Table 1. Single-port SRAM write mask function

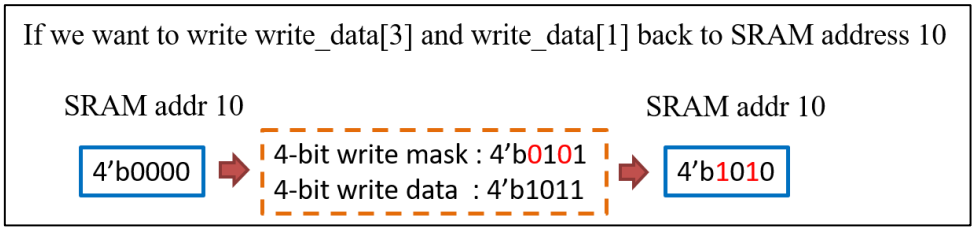


Fig.14 Example for Write mask

D. Summary

In this assignment, 21x21 QR code patterns are embedded in 64x64 images which have different logos as its background. You need to:

1. **Locate the QR code and its orientation**
2. **Read the text to encode, encode and mask them with correct mask pattern**
3. **Put the encoded data codewords back to SRAM**

For more details about the QR code, you can refer to [1].

Notes: The origin SRAM contains the image and the QR code framework without data codewords. You should only add the data codewords to the QR code as shown in Fig. 15. DO NOT modify the other pixels outside the region of data codewords including the background image and the other part of QR code. Otherwise, you will get the wrong result.

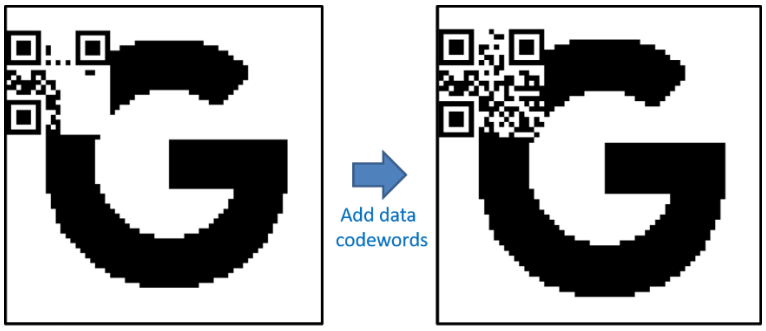
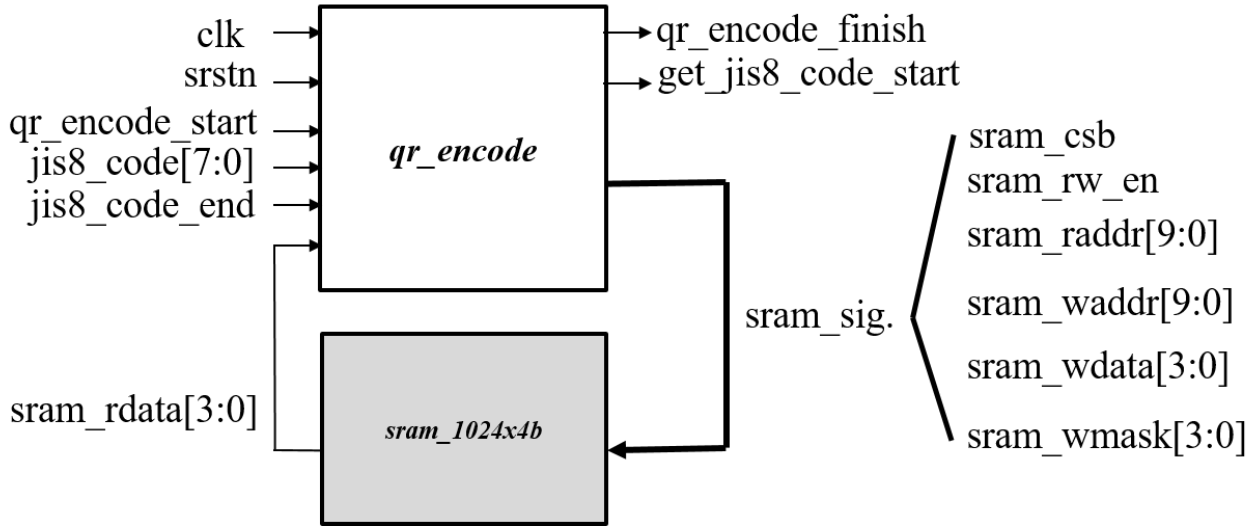


Fig. 15 The image in SRAM before and after processing your qr_encode engine (pat 02)

IV. Design specification

The RTL module name and I/O definition **must** be as follows:




```
module qr_encode(
    input           clk,            //clock input
    input           srstn,          //synchronous reset (active low)
    input           qr_encode_start,//start encoding for one QR code
                                    //1: start (one-cycle pulse)
    output          qr_encode_finish,//1: encoding one QR code is finished
    input   [3:0]   sram_rdata,     //read data from SRAM
    output  [3:0]   sram_wdata,     //write data to SRAM
    output  [9:0]   sram_raddr,     //read address from SRAM
    output  [9:0]   sram_waddr,     //write address to SRAM
    output          sram_rw_en,     //read-write enable,0:write,1:read
    output          sram_csb,       //sram chip enable, active low
    output  [3:0]   sram_wmask,     //write mask: determine which bit
                                    //should be writed into sram when
                                    //operate write
    input           jis8_code_end,  //1: represent the end of JIS8 code
    input   [7:0]   jis8_code,      //JIS8 code which need to be encoded
    output          get_jis8_code_start //1: start to get JIS8 code
);
```

V. Grading rules

For this assignment, we have three grading ranks:

- Rank A – Locations of QR code are randomly located in images with logos and the orientation may be 0°, 90°, 180°, 270°. Pass all simulation patterns in "*sim/bmp/RANK_A*" (no. 0-39)
- Rank B – Locations of QR code are randomly located in images with white background and the orientation may be 0°, 90°, 180°, 270°. Pass all simulation patterns in "*sim/bmp/RANK_B*" (no. 0-39)
- Rank C – Locations of QR code are randomly located in images with white background and the orientation may be 0°. Pass all simulation patterns in "*sim/bmp/RANK_C*" (no. 0-9)

You need to complete the following missions:

1. RTL coding and simulation (A: 6%; B: 4%; C:2%).
2. Use nLint to show the synthesizability (A: 1%; B: 1%; C:1%).
3. Logic synthesis with the provided scripts. Only two script files can be modified: ***0_readfile.tcl*** for adding your self-defined RTL files, and ***synthesis.tcl*** for changing the clock timing constraint TEST_CYCLE (e.g. 2.1 for 2.1 ns). Your timing slack should not be negative. The grading will be based on your performance index (defined later):

$$\text{For Rank A: } \begin{cases} 5\% & if\ \ \boldsymbol{PI\_A} > 3.5\times10^9 \\ 6\% & if\ \ 3.5\times10^9 \geq \boldsymbol{PI\_A} > 5.5\times10^8, \\ 8\% & if\ \ \boldsymbol{PI\_A} \leq 5.5\times10^8 \end{cases}$$

$$\text{For Rank B: } \begin{cases} 3\% & if\ \ \boldsymbol{PI\_B} > 3\times10^9 \\ 4\% & if\ \ 3\times10^9 \geq \boldsymbol{PI\_B} > 3\times10^8. \\ 5\% & if\ \ \boldsymbol{PI\_B} \leq 3\times10^8 \end{cases}$$

$$\text{For Rank C: } \begin{cases} 1\% & if\ \ \boldsymbol{PI\_C} > 13000 \\ 2\% & if\ \ 13000 \geq \boldsymbol{PI\_C} > 11000 \\ 3\% & if\ \ \boldsymbol{PI\_C} \leq 11000 \end{cases}$$

4. **Bonus**. For Rank A works, 3% will be given to the best work in terms of the performance index PI_A, and 1% given to the second best.

Definition of performance index:

A: Total area (shown in report/report_area_qr_encode.out, e.g. Total area: 7642)
T: TEST_CYCLE (your timing constraint, e.g. 2.1 for a nice implementation)
C: The cycle count you need to complete the simulation of *test_qr_encode.v*, which is shown on the monitor when the simulation is finished, for example:

Congratulations! Simulation from pattern no 00 to 39 is successfully passed!
Total cycle count = 33480!

Performance index for Rank A (PI_A)= A*T*C (e.g. $5.5*10^8$ for the above example)
Performance index for Rank B (PI_B)= A*T*C (e.g. $2.7*10^8$ for the above example)
Performance index for Rank C (PI_C)= A*T (e.g. 10844 for the above example)

VI. Deliverable

A、 Synthesizable Verilog functional module *qr_encode.v* and all other RTL codes for your own defined modules (if any).

B、 Create a file named *hdl.f*, which contains all the RTL files (*.v) you use in this assignment.

C、 nLint report file (don't forget to use ***nLint*** rule set : iclab_HW4.rc).

D、 The two modified synthesis script files: *0_readfile.tcl* and *synthesis.tcl*. The report files for timing and area: *report_area_qr_encode.out* and *report_time_ qr_encode.out*. The synthesis log file *da.log*.

E、 A text file ***misc.txt*** summarizing your Rank, performance index and your A, T, and C as well.

Submit File Structure (If you don't follow the structure below, you will lose up to 1%)

- HW4/
  - hdl/
    - ✓ hdl.f
    - ✓ qr_encode.v
    - ✓ …(other Verilog files)
  - syn/
    - ✓ 0_readfile.tcl
    - ✓ synthesis.tcl
    - ✓ report_time_qr_encode.out
    - ✓ report_area_qr_encode.out
    - ✓ da.log
  - nLint/
    - ✓ nLint.rep
  - ✓ misc.txt

* Remember to compress files as HW4_10XXXXXXX.zip ! Don't use .rar or other format.
Or you will lose some points.

VII. File Organization

Directory	Filename	Description
qr_encode/hdl/	qr_encode.v	RTL code for QR code encoder (modify it directly for this assignment)
qr_encode/sim/	test_qr_encode.v	Testbench for testing QR code decoder

	test_qr_encode.f	File list for simulation
	sram_model/sram_1024x4b.v	SRAM behavior model
	bmp/RANK_A/*.bmp	QR code image: 00.bmp-39.bmp
	bmp/RANK_B/*.bmp	QR code image: 00.bmp-39.bmp
	bmp/RANK_C/*.bmp	QR code image: fixed orientation 0°
	golden/*.dat	Test pattern information
	golden/bmp/RANK_*/*.bmp	Golden SRAM results (decodable for any QR code software)
qr_encode/syn/	*.tcl	.tcl file for DC scripts
	run_dc.bat	Batch file for running synthesis
qr_encode/nLint/	iclab_HW4.rs	nLint Rule set for this HW

Note:

For debugging, you can run simulation for selected test patterns. For example, you can test the pattern no. 2, Rank A by modify in test_qr_encode.f:

ncverilog –f test_qr_encode.f + define+PAT_START_NO=2 + define+PAT_END_NO=2 +define+RANK_A

or the patterns of Rank A only by

ncverilog –f test_qr_encode.f + define+PAT_START_NO=0 + define+PAT_END_NO=39 +define+RANK_A

VIII. Reference

[1] ISO/IEC 18004:2000(E), *Information technology — Automatic identification and data capture techniques — Bar code symbology — QR Code*.

[2] Wikipedia, https://en.wikipedia.org/wiki/QR_code.

Appendix A – JIS8 code table [1]

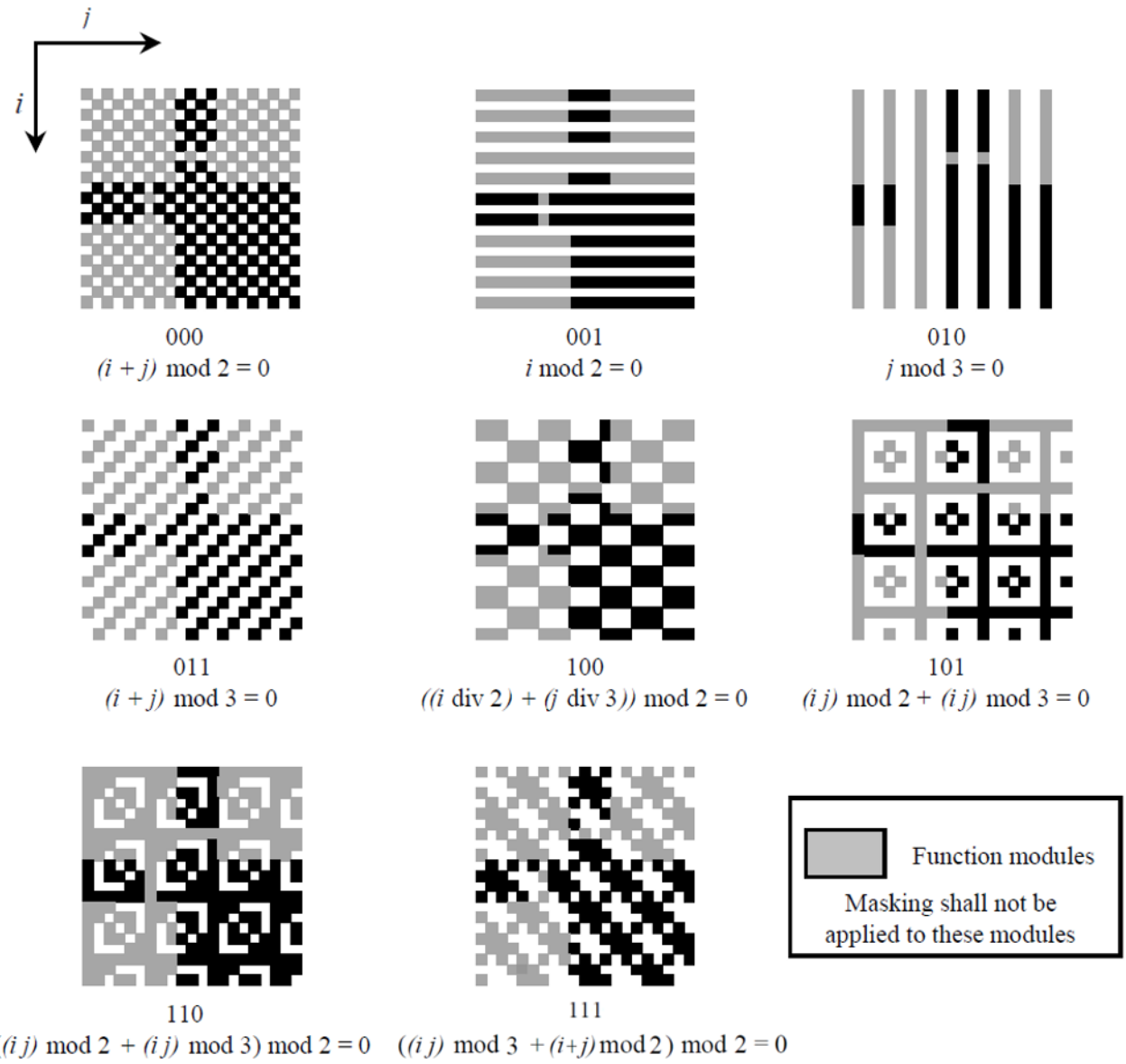
Char.	Hex	Char.	Hex	Char.	Hex	Char.	Hex	Char.	Hex	Char.	Hex	Char.	Hex	Char.	Hex
NUL	00	SP	20	@	40	`	60		80		A0	タ	C0		E0
SOH	01	!	21	A	41	a	61		81	。	A1	チ	C1		E1
STX	02	"	22	B	42	b	62		82	「	A2	ツ	C2		E2
ETX	03	#	23	C	43	c	63		83	」	A3	テ	C3		E3
EOT	04	$	24	D	44	d	64		84	、	A4	ト	C4		E4
ENQ	05	%	25	E	45	e	65		85	・	A5	ナ	C5		E5
ACK	06	&	26	F	46	f	66		86	ヲ	A6	ニ	C6		E6
BEL	07	'	27	G	47	g	67		87	ァ	A7	ヌ	C7		E7
BS	08	(	28	H	48	h	68		88	ィ	A8	ネ	C8		E8
HT	09	)	29	I	49	I	69		89	ゥ	A9	ノ	C9		E9
LF	0A	*	2A	J	4A	j	6A		8A	ェ	AA	ハ	CA		EA
VT	0B	+	2B	K	4B	k	6B		8B	ォ	AB	ヒ	CB		EB
FF	0C	,	2C	L	4C	l	6C		8C	ャ	AC	フ	CC		EC
CR	0D	-	2D	M	4D	m	6D		8D	ュ	AD	ヘ	CD		ED
SO	0E	.	2E	N	4E	n	6E		8E	ョ	AE	ホ	CE		EE
SI	0F	/	2F	O	4F	o	6F		8F	ッ	AF	マ	CF		EF
DLE	10	0	30	P	50	p	70		90	ー	B0	ミ	D0		F0
DC1	11	1	31	Q	51	q	71		91	ア	B1	ム	D1		F1
DC2	12	2	32	R	52	r	72		92	イ	B2	メ	D2		F2
DC3	13	3	33	S	53	s	73		93	ウ	B3	モ	D3		F3
DC4	14	4	34	T	54	t	74		94	エ	B4	ヤ	D4		F4
NAK	15	5	35	U	55	u	75		95	オ	B5	ユ	D5		F5
SYN	16	6	36	V	56	v	76		96	カ	B6	ヨ	D6		F6
ETB	17	7	37	W	57	w	77		97	キ	B7	ラ	D7		F7
CAN	18	8	38	X	58	x	78		98	ク	B8	リ	D8		F8
EM	19	9	39	Y	59	y	79		99	ケ	B9	ル	D9		F9
SUB	1A	:	3A	Z	5A	z	7A		9A	コ	BA	レ	DA		FA
ESC	1B	;	3B	[	5B	{	7B		9B	サ	BB	ロ	DB		FB
FS	1C	<	3C	¥	5C	|	7C		9C	シ	BC	ワ	DC		FC
GS	1D	=	3D	]	5D	}	7D		9D	ス	BD	ン	DD		FD
RS	1E	>	3E	^	5E	‾	7E		9E	セ	BE	゛	DE		FE
US	1F	?	3F	_	5F	DEL	7F		9F	ソ	BF	゜	DF		FF

Note that the codes 8’h00-8’h7f are the same as those in the ASCII code.

Appendix B – Test pattern list

pat no image orientation mask text length text
pat 00: 00.bmp, 000 degree, mask pattern 3, 15-byte, "www.nthu.edu.tw"
pat 01: 01.bmp, 000 degree, mask pattern 1, 17-byte, "www.wikipedia.org"
pat 02: 02.bmp, 000 degree, mask pattern 5, 17-byte, "www.google.com.tw"
pat 03: 03.bmp, 000 degree, mask pattern 4, 13-byte, "goo.gl/rYWrb4"
pat 04: 04.bmp, 000 degree, mask pattern 1, 13-byte, "goo.gl/w8z8rb"
pat 05: 05.bmp, 000 degree, mask pattern 0, 05-byte, "F=ma?"
pat 06: 06.bmp, 000 degree, mask pattern 6, 17-byte, "(i + j) mod 3 = 0"
pat 07: 07.bmp, 000 degree, mask pattern 2, 17-byte, "wire a=(Z>B)?1:0;"
pat 08: 08.bmp, 000 degree, mask pattern 7, 14-byte, "exp(-i*pi)+1=0"
pat 09: 09.bmp, 000 degree, mask pattern 2, 16-byte, "Cogito, ergo sum"
pat 10: 10.bmp, 090 degree, mask pattern 3, 15-byte, "www.nthu.edu.tw"
pat 11: 11.bmp, 090 degree, mask pattern 1, 17-byte, "www.wikipedia.org"
pat 12: 12.bmp, 090 degree, mask pattern 5, 17-byte, "www.google.com.tw"
pat 13: 13.bmp, 090 degree, mask pattern 4, 13-byte, "goo.gl/rYWrb4"
pat 14: 14.bmp, 090 degree, mask pattern 1, 13-byte, "goo.gl/w8z8rb"
pat 15: 15.bmp, 090 degree, mask pattern 0, 05-byte, "F=ma?"
pat 16: 16.bmp, 090 degree, mask pattern 6, 17-byte, "(i + j) mod 3 = 0"
pat 17: 17.bmp, 090 degree, mask pattern 2, 17-byte, "wire a=(Z>B)?1:0;"
pat 18: 18.bmp, 090 degree, mask pattern 7, 14-byte, "exp(-i*pi)+1=0"
pat 19: 19.bmp, 090 degree, mask pattern 2, 16-byte, "Cogito, ergo sum"
pat 20: 20.bmp, 180 degree, mask pattern 3, 15-byte, "www.nthu.edu.tw"
pat 21: 21.bmp, 180 degree, mask pattern 1, 17-byte, "www.wikipedia.org"
pat 22: 22.bmp, 180 degree, mask pattern 5, 17-byte, "www.google.com.tw"
pat 23: 23.bmp, 180 degree, mask pattern 4, 13-byte, "goo.gl/rYWrb4"
pat 24: 24.bmp, 180 degree, mask pattern 1, 13-byte, "goo.gl/w8z8rb"
pat 25: 25.bmp, 180 degree, mask pattern 0, 05-byte, "F=ma?"
pat 26: 26.bmp, 180 degree, mask pattern 6, 17-byte, "(i + j) mod 3 = 0"
pat 27: 27.bmp, 180 degree, mask pattern 2, 17-byte, "wire a=(Z>B)?1:0;"
pat 28: 28.bmp, 180 degree, mask pattern 7, 14-byte, "exp(-i*pi)+1=0"
pat 29: 29.bmp, 180 degree, mask pattern 2, 16-byte, "Cogito, ergo sum"
pat 30: 30.bmp, 270 degree, mask pattern 3, 15-byte, "www.nthu.edu.tw"
pat 31: 31.bmp, 270 degree, mask pattern 1, 17-byte, "www.wikipedia.org"
pat 32: 32.bmp, 270 degree, mask pattern 5, 17-byte, "www.google.com.tw"
pat 33: 33.bmp, 270 degree, mask pattern 4, 13-byte, "goo.gl/rYWrb4"
pat 34: 34.bmp, 270 degree, mask pattern 1, 13-byte, "goo.gl/w8z8rb"
pat 35: 35.bmp, 270 degree, mask pattern 0, 05-byte, "F=ma?"
pat 36: 36.bmp, 270 degree, mask pattern 6, 17-byte, "(i + j) mod 3 = 0"
pat 37: 37.bmp, 270 degree, mask pattern 2, 17-byte, "wire a=(Z>B)?1:0;"
pat 38: 38.bmp, 270 degree, mask pattern 7, 14-byte, "exp(-i*pi)+1=0"
pat 39: 39.bmp, 270 degree, mask pattern 2, 16-byte, "Cogito, ergo sum"

Appendix C – Mask patterns [1]



Note: *div* means integer division. (Hint: You may first use the "/" operator to test correctness and then try another implementation to reduce logic gates.)