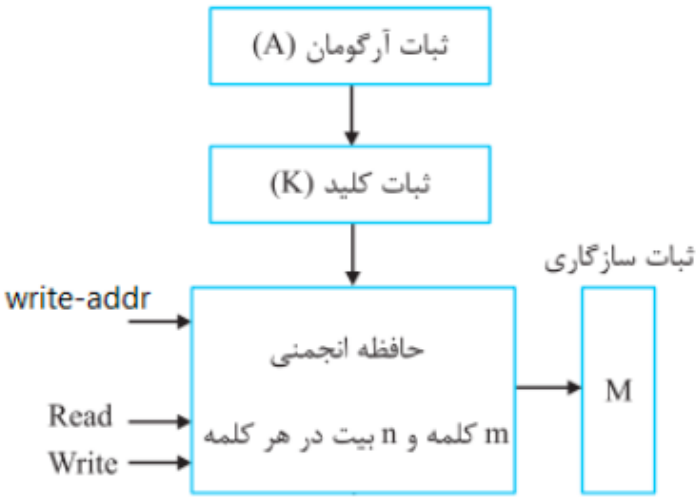
📝 گزارش آزمایش ۹

آزمایشگاه طراحی سیستم‌های دیجیتال

> *Alireza Habibzadeh* `99109393`
> *June 21, 2022*

مقدمه

در این آزمایش یک حافظه‌ی محتوا دسترس‌پذیر از نوع سه‌گانه (Ternary Content Addressable Memory) را پیاده‌سازی می‌کنیم.



شکل شماتیک حافظه

پیاده‌سازی

کد ما دو قسمت اصلی read و write دارد که به خواسته‌ی سوال وقتی `read_enable` و `write_enable` یک هستند وارد بلاک if متناظرشان می‌شویم و اولویت با read است.

بخش write

در این بخش به سادگی قسمت متناظر با `write_address` در حافظه را به شکل زیر مقداردهی می‌کنیم:

```
data[write_address] <= ~(key | 16'bxxxxxxxxxxxxxxxx) ^ arg;
```

عبارت سمت راست ابتدا از روی `key` یک مقدار جدید می‌سازد که جای بیت‌های 1 همان 0 و جای بیت‌های 0، X قرار گرفته است. سپس این عبارت را با `arg` XOR می‌کنیم. نتیجه این است که بیت‌های

با اهمیت بی‌تغییر می‌مانند و بیت‌های بی‌اهمیت (don't care) برابر با X می‌شوند که مطابق خواسته‌ی سوال است. برای مثال:

```
key = 1001
arg = 1010

key | xxxx = 1xx1
~(key | xxxx) = 0xx0
~(key | xxxx) ^ arg = 1xx0 // done
```

بخش read

در این بخش از روشی مشابه بخش قبل استفاده شده است. از یک حلقه‌ی for برای چک کردن تک‌تک بیت‌های `match_result` استفاده می‌کنیم. البته نکته اینجا است که چون در assignmentها از `<=` استفاده شده است، این حلقه باعث تاخیر و کندی حافظه‌ی ما که باید تاخیر $\mathcal{O}(1)$ داشته باشد نمی‌شود و حتما می‌توان این حلقه را به عملیات‌های موازی سنتز کرد.

```
for (i = 0; i < 16; i = i + 1)
    match_result[i] <= |{1'bx, (arg ^ data[i]) & key} === 1'bx;
```

در این خط ابتدا `arg` با حافظه XOR شده است که حاصل بیت 1 برای بیت‌های نامساوی، 0 برای برابری‌ها و X برای don't care حافظه است. سپس این مقدار با `key` AND شده که بیت‌هایی که در این مقایسه مهم نبودند حتما 0 شوند. حال باید ببینیم حاصل نهایی بیت 1 دارد یا خیر. برای این کار یک reduction OR کار را انجام می‌دهد و اگر یک بیت هم 1 باشد حاصل را یک می‌کند. اما در غیر این صورت حاصل ممکن است 0 یا X باشد. برای حل این مشکل الان که فکر می‌کنم می‌شد از `!=` استفاده کرد :)) ولی راهی که در نهایت من استفاده کردم concat کردن یک X و مطمئن شدن از این که حالت 0 به X تبدیل می‌شود. همچنین از `===` استفاده شده تا با X مثل یک مقدار رفتار کند.

کد نهایی ماژول `tcam` برابر بلاک زیر است:

```
// Ternary Content Addressable Memory
module tcam(
    input clock,
    input reset,
    input [15:0] arg,
    input [15:0] key,
    input [3:0] write_address,
    input read_enable,
```

```
    input write_enable,
    output reg [15:0] match_result
    );

    reg [15:0] data [0:15];
    reg [4:0] i;

    always @(posedge clock or negedge reset) begin
        if (!reset) begin
            match_result <= 0;
            for (i = 0; i < 16; i = i + 1) data[i] <= 0;
        end else begin
            if (read_enable)
                for (i = 0; i < 16; i = i + 1)
                    match_result[i] <= |{1'bx, (arg ^ data[i]) & key} === 1'bx;
            else if (write_enable)
                data[write_address] <= ~(key | 16'bxxxxxxxxxxxxxxxx) ^ arg;
        end
    end
endmodule
```

تست

از آنجایی که این نوع حافظه کمی پیشرفته‌تر از یک «حافظه» است، برای تست آن مقدارهای جالبی می‌توان ذخیره کرد که عملا نوعی pattern-match انجام می‌دهد. چند خانه‌ی اول حافظه اعداد بی‌معنی برای تست هستند و خانه‌های آخر آن سعی شده با مقادیر معنی‌دار پر شوند:

```
write_address = 9; // one check
arg = 16'b0000000000000001;
key = 16'b1111111111111111;
```

```
write_address = 10; // greater/equal 1000000000000000 check
arg = 16'b1000000000000000;
key = 16'b1000000000000000;
```

```
write_address = 11; // even check
arg = 16'b0000000000000000;
key = 16'b0000000000000001;
```

```
write_address = 12; // odd check
arg = 16'b0000000000000001;
key = 16'b0000000000000001;
```

```
write_address = 13; // tautology (always true)
arg = 16'b1111111111111111;
key = 16'b0000000000000000;
```

```
write_address = 14; // all one (-1) check
arg = 16'b1111111111111111;
key = 16'b1111111111111111;
```

```
write_address = 15; // zero check
arg = 16'b0000000000000000;
key = 16'b1111111111111111;
```

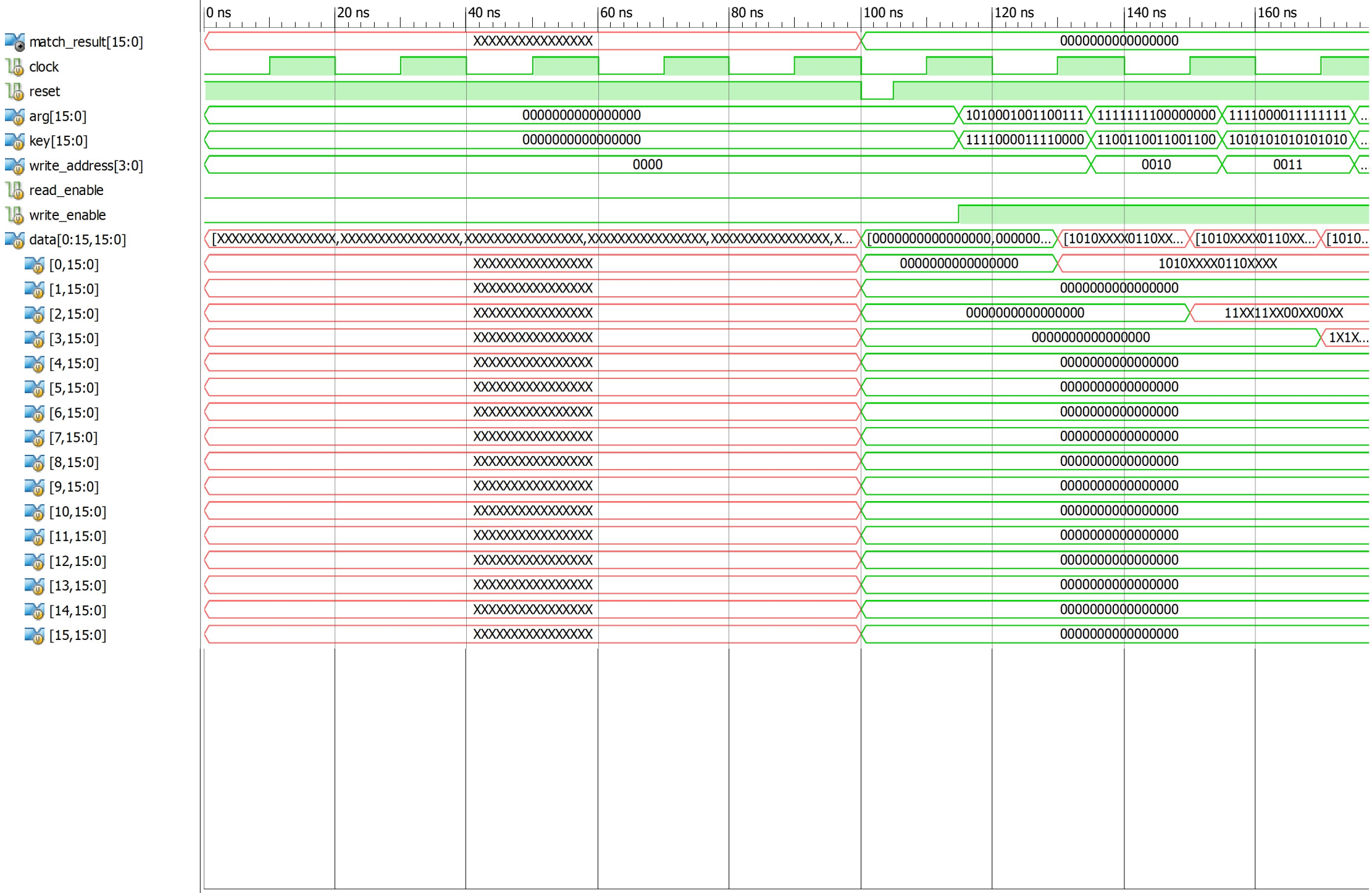
کد نهایی ماژول تست‌بنچ به دلیل طولانی بودن اینجا قرار نگرفته البته در فایل‌های ارسالی موجود می‌باشد.

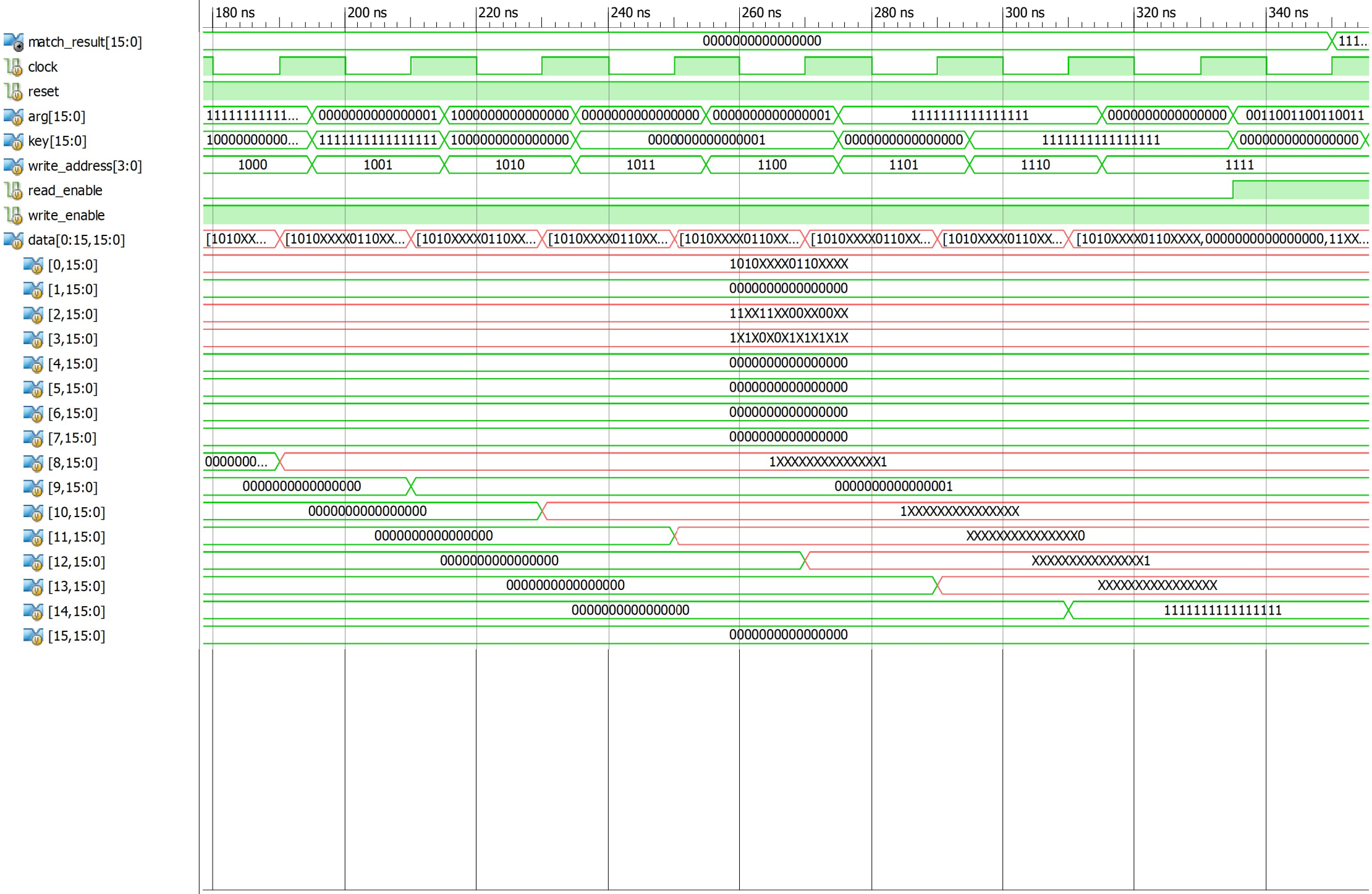
شبیه‌سازی

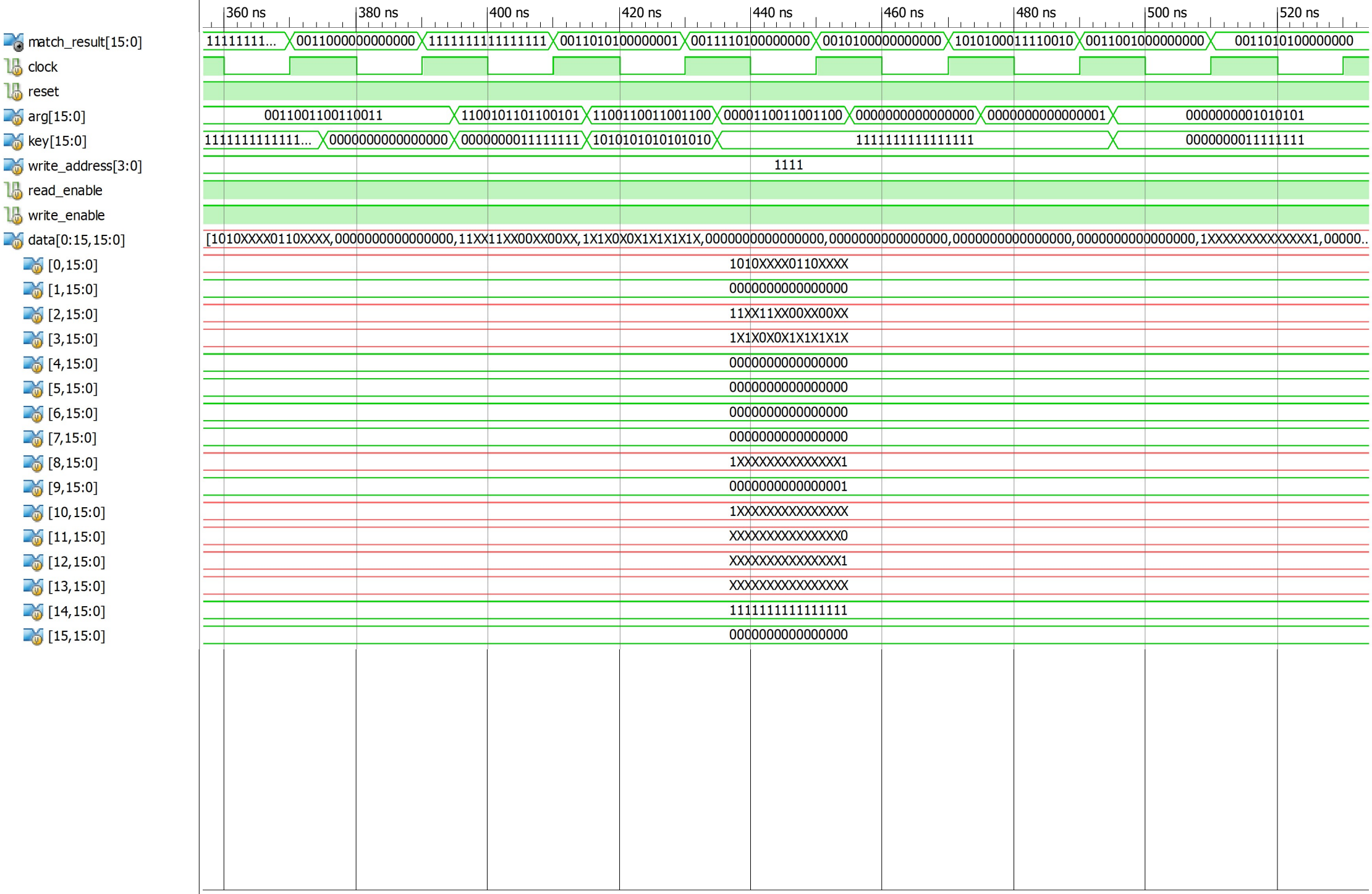
شکل wave نتیجه‌ی شبیه‌سازی در انتهای فایل آمده است. بیت‌های match

- *zero check*
- *all one (-1) check*
- *tautology (always true)*
- *odd check*
- *even check*
- *greater/equal 1000000000000000 check*
- *one check*

که در بخش قبلی تعریف شدند علاوه بر خانه‌هایی از حافظه که داده‌های الکی برای تست داشتند به ترتیب از چپ به راست در خروجی `match_result` قابل مشاهده‌اند. (چون رجیستر آن به صورت little-endian تعریف شده)


0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
match_result[15:0]
clock
reset
arg[15:0]
key[15:0]
write_address[3:0]
read_enable
write_enable
data[0:15,15:0]
[0,15:0]
[1,15:0]
[2,15:0]
[3,15:0]
[4,15:0]
[5,15:0]
[6,15:0]
[7,15:0]
[8,15:0]
[9,15:0]
[10,15:0]
[11,15:0]
[12,15:0]
[13,15:0]
[14,15:0]
[15,15:0]
XXXXXXXXXXXXXXX
0000000000000000
1010001001100111
1111111100000000
1111000011111111
1111000011110000
1100110011001100
1010101010101010
0000
0010
0011
1010XXXX0110XXXX
11XX11XX00XX00XX
1X1X...


180 ns
200 ns
220 ns
240 ns
260 ns
280 ns
300 ns
320 ns
340 ns
match_result[15:0]
clock
reset
arg[15:0]
key[15:0]
write_address[3:0]
read_enable
write_enable
data[0:15,15:0]
[0,15:0]
[1,15:0]
[2,15:0]
[3,15:0]
[4,15:0]
[5,15:0]
[6,15:0]
[7,15:0]
[8,15:0]
[9,15:0]
[10,15:0]
[11,15:0]
[12,15:0]
[13,15:0]
[14,15:0]
[15,15:0]
0000000000000000
111..
11111111111...
0000000000000001
1000000000000000
0000000000000000
0000000000000001
1111111111111111
0000000000000000
0011001100110011
10000000000...
1111111111111111
1000000000000000
0000000000000001
0000000000000000
1111111111111111
0000000000000000
1000
1001
1010
1011
1100
1101
1110
1111
[1010XX...
[1010XXXX0110XX...
[1010XXXX0110XX...
[1010XXXX0110XX...
[1010XXXX0110XX...
[1010XXXX0110XX...
[1010XXXX0110XX...
[1010XXXX0110XXXX,0000000000000000,11XX...
1010XXXX0110XXXX
0000000000000000
11XX11XX00XX00XX
1X1X0X0X1X1X1X1X
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000...
1XXXXXXXXXXXXXX1
0000000000000000
0000000000000001
0000000000000000
1XXXXXXXXXXXXXXX
0000000000000000
XXXXXXXXXXXXXXX0
0000000000000000
XXXXXXXXXXXXXXX1
0000000000000000
XXXXXXXXXXXXXXXX
0000000000000000
1111111111111111
0000000000000000


360 ns
380 ns
400 ns
420 ns
440 ns
460 ns
480 ns
500 ns
520 ns
match_result[15:0]
clock
reset
arg[15:0]
key[15:0]
write_address[3:0]
read_enable
write_enable
data[0:15,15:0]
[0,15:0]
[1,15:0]
[2,15:0]
[3,15:0]
[4,15:0]
[5,15:0]
[6,15:0]
[7,15:0]
[8,15:0]
[9,15:0]
[10,15:0]
[11,15:0]
[12,15:0]
[13,15:0]
[14,15:0]
[15,15:0]
11111111...
0011000000000000
1111111111111111
0011010100000001
0011110100000000
0010100000000000
1010100011110010
0011001000000000
0011010100000000
0011001100110011
1100101101100101
1100110011001100
0000110011001100
0000000000000000
0000000000000001
0000000001010101
1111111111111...
0000000000000000
0000000011111111
1010101010101010
1111111111111111
0000000011111111
1111
[1010XXXX0110XXXX,0000000000000000,11XX11XX00XX00XX,1X1X0X0X1X1X1X1X,0000000000000000,0000000000000000,0000000000000000,0000000000000000,1XXXXXXXXXXXXXX1,00000..
1010XXXX0110XXXX
0000000000000000
11XX11XX00XX00XX
1X1X0X0X1X1X1X1X
0000000000000000
0000000000000000
0000000000000000
0000000000000000
1XXXXXXXXXXXXXX1
0000000000000001
1XXXXXXXXXXXXXX
XXXXXXXXXXXXXX0
XXXXXXXXXXXXXX1
XXXXXXXXXXXXXX
1111111111111111
0000000000000000