Bomb Processing

Daniel Vlaardingerbroek, 4580834
Koen van Remundt, 4601793

December 7, 2018

Abstract

This document describes the contents a design report for the module design of the bomb processing of the game "bomberman" within the EPO-3 project.

too short

Contents

'1111000' (120 clock cycles). The FSM of Timer Main can be found in Fig. 2.

When Timer# goes high the FSM will turn the counter o nby making `Counter_R#` low., meaning that the timer is assigned to the bomb currently placed. Once `Count_Out` has reached '1111000' it will set `Explosion#` high for one clock cycle, letting Bomb Handling know that the bomb has exploded. Then it will go back to Timer Reset, where it will wait until the next bomb is placed.

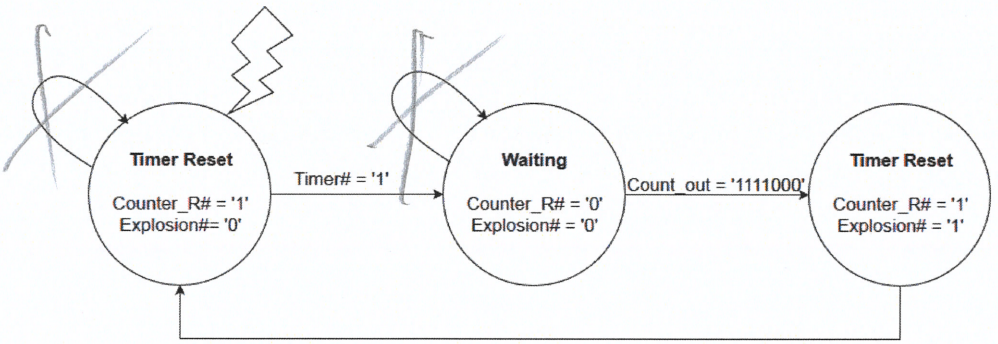

Figure 2: The FSM of the Timer Main entity.

3.1.3 Counter

The counter is controlled using a reset signal (`Counter_R#`) coming from Timer Main (3.1.2). When `Counter_R#` is low, the counter will add a one to a seven bit vector signal called `Count_Out`, this happens every clock cycle. The counter will keep adding one to `Count_Out` until `Counter_R#` goes high again, after which the counter will stop counting and the value of `count_out` will be reset to 0.

3.1.4 Bomb Handling

As said in Sec. 3.1, the amount of timers is equal to the maximum amount of bombs that can be on the field at a time. The task of the entity Bomb Handling is to assign a timer to a bomb once it has been placed. As explained in Sec. 3.1.2, `Counter_R#` dictates whether the counter is on or off, therefor it can a be used to determine whether a timer is in use or not. The FSM of Bomb Handling can be found in Fig. 5. There are three possible input scenario's for Bomb Handling: Player one places a bomb, player two places a bomb and both players place a bomb at the same time.

When player one places a bomb, the FSM goes to the *Timer1* state, here will be determined whether Timer1 is in use or not by checking whether the `Counter_R1` signal is high or low. If the timer is not in use it will be started. If the timer is in use then the FSM will go to the next timer, in this case that is the *Timer2* state, where it will once again check whether the timer is in use or not. This sequence will continue for the amount of bombs that a player can maximally place. When a player has reached their maximum amount of bombs placed at a time, the FSM will just cycle through all timers and then return to the reset state. The same sequence holds for player two, but then with different timers (Timer5-6 in the case of 2 bombs each).
A special case is when both players want to place a bomb at exactly the same time. If this happens, the bomb of player one will first be placed and then the bomb of player two. This is done by setting

a flag (dflag) to high in the case that both players place a bomb at the same time. When a timer of player one starts in the *Start Timer1-2* state, it is checked whether dflag is set to high or low. In the case that dflag is set to high, the FSM will then continue with checking whether the timers of player 2 are in use, and turning them on if that's not the case. Just like it has been described above.

3.1.5 Upgradability

The main upgrade that will probably happen (that affect Bomb Timer) is upping the amount of bombs a player can place at a time. This will be very easy to add to Bomb Timer, as it is a matter of copy-pasting the port maps for Timer Main # in the behavioural description of Bomb Handling. This will also then extra counters, as all Timer Mains need their own. Next to adding the port maps extra signals and states will have to be added too, but this is a matter of copying existing states and adding the right signals to them in accordance to the FSM as shown in Fig. 3.

3.2 Lethal state generator

As said in Sec. 2, the lethal state generator needs to generate a lethal state for a certain amount of tiles. This is why this section is split up in two different entities as seen in Fig. 4. The entity FF2 needs to keep track of the amount of tiles that have been set to the lethal state. The Lethal state FSM does the actual computing and tile coordinate generation.

3.2.1 Lethal state FSM

The Lethal state part of Fig. 4 is actually one large FSM as seen in Fig. 6. In the beginning, the FSM stores the coordinates given to him so that the inputs can change while the FSM is working. These values are stored as signed, since these values need to be able to be negative. This is all done in the *rest* state.
The FSM initiates after the `explosion` signal is high. The FSM checks if the last digit of the X-coordinate is a 1 or a 0, since the last digit decides if the coordinate is even or odd. If it is even (last digit being 0), the FSM checks the same for the vertical coordinate. If both are even, the FSM goes to the *Plusform* state. In this state the FF1 signal stores the value 1 in order to remember later on that the FSM needs to generate a plus form.

After that, the FSM checks for 5 values in a row(more on this in Sec. 3.2.2) if both a player and a `lethal_flag` signal are present, which shows that it has generated a new tile. This row of 5 tiles is managed by subtracting 2 at the beginning in the *horizontal* and *vertical* states, this is why the values need to be able to be negative. Because of this subtraction, the values were stored as signed and not as unsigned. The 5 individual tiles are calculated by switching between the *vert_wait* and the *vert_out* states for the vertical line check. The *hori_wait* and *hori_out* states are used for the horizontal line check.
If a player is never met after 5 tiles, the FSM starts over again. If a player is detected, the FSM will go to the *victory1* state when player 2 dies and will go to *victory2* state when player 1 dies. This will generate a 2-bits signal `victoryv` of which `victoryv(1)` becomes 1 when a player is hit. This can also be used as a permanent reset for all other subsystems except the lethal state FSM, since that system needs to stay in a victory state until the entire game has been reset. `victoryv(0)` becomes 1 when player 2 wins and becomes 0 when player 1 wins.

The signal `lethal_flag` was first designed to show to the "coordinates" group that they were allowed to read the newly calculated coordinates and change them to a lethal state and later on show that on the screen. Due to a space shortage, this feature had to be cut. This signal is now used by the FF2 part.

3.2.2 FF2

In this case, we chose to generate 2 tiles in each direction of the bomb, these directions being up, down, left and right, resulting in 5 horizontal tiles and 5 vertical tiles in one row in total(the 5th tile in both directions is the bomb location itself). We might want to add things such as "power ups" in the game that might allow this value to rise to a higher value, but this might take a lot of space, so it has been decided to go with 2 tiles in each direction for now. As mentioned in the FSM part, the signal *lethal_flag* is now used by the FF2 part to count how many tiles have been checked. Amount of times the signal *lethal_flag* goes to high will be counted until the counter reaches [100]. At first we tested with the counter stopping at [101], which is the actual value of 5. For some reason, this resulted in a line of 6 tiles, so we changed it to [100] and this seems to work fine. The possibility is that the system added 1 to the counter after the FSM checks if the value is 5 and thus adding one too many tiles to the row or column.

3.2.3 Upgradability

This part of the system is pretty straight forward. It checks all the tiles when bombs detonate. This is an essential part of the game and is difficult to do in a completely different way or with other values. Generating the coordinates is already an essential part of the subsystem, but can later on be used to generate a old lethal state in the game by displaying something like a smoke screen. However, since the coordinates are already generated for this system, the requirements are already met and need to be used by another subsystem to generate a smoke screen.
An upgrade that might be made to this subsystem is a longer lethal state. This subsystem checks on present players the instant the bomb explodes. In the original "Bomberman" this is not the case. An exploding bomb stays lethal for a while before disappearing. If the space on the chip allows this, it might be able to implement this.
Lastly, as already mentioned in Sec. 3.2.2 we might want to increase or decrease the radius of the bombs, if we chose to add power-ups to the game. If this is the case, the number stored in the FF2 needs to be a variable instead of a constant. Since this is all handled in the FF2 part, one extra input will not be a big problem.

4 Results

As explained above, this module has two very different entities who do not interact a lot with each other, for that reason the two entities have been simulated separately, as simulating them together would make the waveform unnecessary large. After simulating both of them separately, they have been put together in one entity to get an accurate result for the amount of sequential cells, combinational cells and the area of the module. These values can be found in table 1. In the upcoming two sections the results of the simulation of both entities will be covered extensively.

Parameters	Values
Sequential cells	78
Combinational cells	279
Total Area	6748.045 μm^2

Table 1: The amount of sequential cells, combinational cells and the total area of the Bomb Processing module.

4.1 Bomb Timer

To simulate the workings of Bomb Timer on the chip, a testbench has been made. This testbench has emulated all possible scenario's the entity can be in, while showing the important signals on the waveform. The waveform is shown in Fig. 11. To emulate all possible scenario's in a compact way the bomb will go off once `Count_Out` reaches '0001010', also only one entity Timer Main and one entity Counter have been simulated, specifically the Counter and Timer Main for Timer1.

First and foremost, one can see that no matter how long the player presses the button (`Bombp1` and `Bombp2`), the input into the entity Bomb Handling only lasts for two clock signals, showing that the buffer works as intended. When this input is high, `Timer1` goes high to make `Count_r1` low, which starts the counter. This is all done in accordance to the FSM of Sec. 3.1.2. When `Count_Out` reaches '0001010' the counter stops and makes the `Explosion1` signal high for one clock cycle. This whole sequence also works for player 2, as `Explosion5` goes high after player two presses the button. There is a slight deviation in the time it takes for the bomb to explode, but that's due to the path that the FSM has to go through in order to start the timer. This takes between 3 and 8 clock cycles, depending on who presses the button and how many timers are already in use. However, this delay will be negligible when the chip runs on a clock speed of 12.5MHz.

When both player 1 and player 2 press the button at the same time, one can see that roughly 15-20 clock cycles later bomb 1 and then bomb 2 explode. This shows that the FSM described in Sec. 3.1.4 works as intended, as the bomb of player 1 explodes first.

Throughout the simulation both player 1 and player 2 placed multiple bombs and sometimes two straight after each other. When this happens it can be seen that the FSM goes to the second timer for the bomb that is placed, both for player 1 and player 2. This means that the FSM can accurately state when a timer is in use and then move to the next timer.

All these results mean that the entity Bomb Timer works as specified in Sec. 2 and will be a good contribution to the game Bomberman.

4.2 Lethal State Generator

For the test of the Lethal State Generator, the worst case scenario would be that the FSM has to go trough both the vertical and the horizontal player check in one run. In order to do this, the bomb needs to be in an even X- and Y-coordinate. For the test bench, the bomb got the position x=6, y=6 and for the first run, the players would not be hit, since that would cause a premature stop in the FSM. As seen in Fig. 9 and Fig. 10 every tile in a plus form is being checked. A thing that also needed to be tested was when a player was present in the lethal tiles. Fig. 9 and Fig. 10 (and Fig. 7 and Fig. 8) were

actually one figure, but were split in 2 pieces, because the screenshots were very long and difficult to read if put on a single piece of paper.
After the plus form, player 2 was moved to tile x=6, y=7, as seen in Fig. 7, which is right next to the bomb. This was done to test the lethality in the system. As seen in Fig. 8, the fourth tile that is being checked is the x=6, y=7 tile, this causes the system to go to the victory screen and it will there indefinitely. With this, every state has been used in these two tests.

5 Conclusion

The module Bomb Processing has been fully completed for the initial design. It can time all bombs and give off a signal when it explodes, and it can calculate whether players are hit by bombs or not, also sending off a signal with all tiles that are affected by the explosion. This will not be used in the current design, but it can come in very handy for future designs (i.e. smoke animation). The chance that there will be upgraded designs is very high, due to the relative small amount of space that is currently used by all modules combined. These upgrades can be added with relative ease, due to the modular design of this module. This all means that the design has met the specifications with a lot of potential for future designs.

6 Appendices

looks nice

6.1 Code

6.1.1 Counter entity

```
library IEEE;
use IEEE.std_logic_1164.ALL;

entity bomb_timer is
   port(clk         : in  std_logic;
        count_reset : in  std_logic;
        count_out   : out std_logic_vector(6 downto 0)
        );
end bomb_timer;
```

6.1.2 Counter behavioural description

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
ARCHITECTURE behaviour_bomb_timer OF bomb_timer IS
        SIGNAL count, new_count : unsigned (6 DOWNTO 0);
BEGIN
        PROCESS (clk)
        BEGIN
            -- Check whether counter should be on or off
                IF (rising_edge (clk)) THEN
                        IF (count_reset = '1') THEN
                                count <= (OTHERS => '0');
                        ELSE
                                count <= new_count;
                        END IF;
                END IF;
        END PROCESS;
        PROCESS (count) -- Add one to new_count if timer is on
        BEGIN
                new_count <= count + 1;
        END PROCESS;
        count_out <= std_logic_vector (count);
END behaviour_bomb_timer;
```

6.1.3 Timer Main entity

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
ENTITY bomb_timer_main IS
        PORT (
```

```
6                   clk          : IN std_logic;
7                   timer1       : IN std_logic;
8                   reset        : IN std_logic;
9                   count_reset  : OUT std_logic;
10                  explosion    : OUT std_logic);
11  END bomb_timer_main;
```

6.1.4 Timer Main behavioural description

```
1   LIBRARY IEEE;
2   USE IEEE.std_logic_1164.ALL;
3   USE IEEE.numeric_std.ALL;
4   ARCHITECTURE behaviour_bomb_main OF bomb_timer_main IS
5           TYPE bomb_state IS (TIMER_RESET, WAITING, BOMB_EXPLODED);
6           COMPONENT bomb_timer IS
7                   PORT (
8                           clk          : IN std_logic;
9                           count_reset  : IN std_logic;
10                          count_out    : OUT std_logic_vector(6 DOWNTO 0)
11                  );
12          END COMPONENT;
13          SIGNAL state, new_state    : bomb_state;
14          SIGNAL count_out           : std_logic_vector(6 DOWNTO 0);
15          SIGNAL count_reset_signal  : std_logic;
16  BEGIN
17          lbl1 : PROCESS (clk)
18          BEGIN
19                  IF (clk'event AND clk = '1') THEN
20                          IF reset = '1' THEN
21                                  state <= TIMER_RESET;
22                          ELSE
23                                  state <= new_state;
24                          END IF;
25                  END IF;
26          END PROCESS;
27          lbl2 : PROCESS (state, timer1, count_out)
28          BEGIN
29                  CASE state IS
30                          WHEN TIMER_RESET =>
31                                  count_reset_signal <= '1';
32                                  explosion          <= '0';
33                                  IF timer1 = '1' THEN
34                                          new_state <= WAITING;
35                                  ELSE
36                                          new_state <= TIMER_RESET;
37                                  END IF;
38                          WHEN WAITING =>
39                                  count_reset_signal <= '0';
40                                  explosion          <= '0';
41                                  IF count_out = "0001010" THEN
42                                          new_state <= BOMB_EXPLOded;
43                                  ELSE
44                                          new_state <= WAITING;
45                                  END IF;
```

```
                        WHEN BOMB_EXPLODED =>
                                count_reset_signal <= '1';
                                explosion          <= '1';
                                new_state          <= TIMER_RESET;
                END CASE;
        END PROCESS;
        count_reset <= count_reset_signal;
        T1 : bomb_timer PORT MAP(clk, count_reset_signal, count_out);
END ARCHITECTURE behaviour_bomb_main;
```

6.1.5 Bomb Handling entity

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY bomb_handling IS
        PORT (
                clk        : IN std_logic;
                reset      : IN std_logic;
                bombp1     : IN std_logic;
                bombp2     : IN std_logic;
                explosion1 : OUT std_logic;
                explosion2 : OUT std_logic;
                explosion5 : OUT std_logic;
                explosion6 : OUT std_logic);
END bomb_handling;
```

6.1.6 Bomb Handling behavioural description

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
ARCHITECTURE behaviour_bomb_handling OF bomb_handling IS
        TYPE handling_state IS (BOMB_RESET, BOMB_WAIT, TIMER1, TIME_START1,
                                TIMER2, TIME_START2, TIMER5, TIME_START5,
                                TIMER6, TIME_START6, DOUBLE);
        COMPONENT bomb_timer_main IS
                PORT (
                        clk         : IN std_logic;
                        timer1      : IN std_logic;
                        reset       : IN std_logic;
                        count_reset : OUT std_logic;
                        explosion   : OUT std_logic);
        END COMPONENT;
        SIGNAL state, new_state
        SIGNAL count_r1, count_r2, count_r5, count_r6, timer_start1,
               timer_start2, timer_start5, timer_start6, dflag : std_logic;
BEGIN
        lbl1 : PROCESS (clk)
        BEGIN
                IF (clk'event AND clk = '1') THEN
```

```
23                      IF reset = '1' THEN
24                              state <= BOMB_RESET;
25                      ELSE
26                              state <= new_state;
27                      END IF;
28              END IF;
29      END PROCESS;
30      lbl2 : PROCESS (state, bombp1, bombp2, count_r1)
31      BEGIN
32              CASE state IS
33                      WHEN BOMB_RESET =>
34                              timer_start1 <= '0';
35                              timer_start2 <= '0';
36                              timer_start5 <= '0';
37                              timer_start6 <= '0';
38                              dflag        <= '0';
39                              new_state    <= BOMB_WAIT;
40                      WHEN BOMB_WAIT =>
41                              timer_start1 <= '0';
42                              timer_start2 <= '0';
43                              timer_start5 <= '0';
44                              timer_start6 <= '0';
45                              dflag        <= '0';
46                              IF bombp1 = '1' AND bombp2 = '0' THEN
47                                      new_state <= TIMER1;
48                              ELSIF bombp1 = '0' AND bombp2 = '1' THEN
49                                      new_state <= TIMER5;
50                              ELSIF bombp1 = '1' AND bombp2 = '1' THEN
51                                      new_state <= DOUBLE;
52                              ELSE
53                                      new_state <= BOMB_WAIT;
54                              END IF;
55                      WHEN DOUBLE =>
56                              timer_start1 <= '0';
57                              timer_start2 <= '0';
58                              timer_start5 <= '0';
59                              timer_start6 <= '0';
60                              dflag        <= '1';
61                              new_state    <= TIMER1;
62                      WHEN TIMER1 =>
63                              timer_start1 <= '0';
64                              timer_start2 <= '0';
65                              timer_start5 <= '0';
66                              timer_start6 <= '0';
67                              dflag        <= dflag;
68                              IF count_r1 = '1' THEN
69                                      new_state <= TIME_START1;
70                              ELSE
71                                      new_state <= TIMER2;
72                              END IF;
73                      WHEN TIME_START1 =>
74                              timer_start1 <= '1';
75                              timer_start2 <= '0';
76                              timer_start5 <= '0';
77                              timer_start6 <= '0';
78                              IF dflag = '1' THEN
79                                      new_state <= TIMER5;
```

inputs are missing! E.g dflag

```
80              ELSE
81                      new_state <= BOMB_WAIT;
82              END IF;
83      WHEN TIMER2 =>
84              timer_start1 <= '0';
85              timer_start2 <= '0';
86              timer_start5 <= '0';
87              timer_start6 <= '0';
88              dflag        <= dflag;
89              IF count_r2 = '1' THEN
90                      new_state <= TIME_START2;
91              ELSE
92                      new_state <= BOMB_WAIT;
93              END IF;
94      WHEN TIME_START2 =>
95              timer_start1 <= '0';
96              timer_start2 <= '1';
97              timer_start5 <= '0';
98              timer_start6 <= '0';
99              IF dflag = '1' THEN
100                     new_state <= TIMER5;
101             ELSE
102                     new_state <= BOMB_WAIT;
103             END IF;
104     WHEN TIMER5 =>
105             timer_start1 <= '0';
106             timer_start2 <= '0';
107             timer_start5 <= '0';
108             timer_start6 <= '0';
109             dflag        <= '0';
110             IF count_r5 = '1' THEN
111                     new_state <= TIME_START5;
112             ELSE
113                     new_state <= TIMER6;
114             END IF;
115     WHEN TIME_START5 =>
116             timer_start1 <= '0';
117             timer_start2 <= '0';
118             timer_start5 <= '1';
119             timer_start6 <= '0';
120             dflag        <= '0';
121             new_state    <= BOMB_WAIT;
122     WHEN TIMER6 =>
123             timer_start1 <= '0';
124             timer_start2 <= '0';
125             timer_start5 <= '0';
126             timer_start6 <= '0';
127             dflag        <= '0';
128             IF count_r6 = '1' THEN
129                     new_state <= TIME_START6;
130             ELSE
131                     new_state <= BOMB_WAIT;
132             END IF;
133     WHEN TIME_START6 =>
134             timer_start1 <= '0';
135             timer_start2 <= '0';
136             timer_start5 <= '0';
```

problem!

```
                                timer_start6 <= '1';
                                dflag        <= '0';
                                new_state    <= BOMB_WAIT;
                END CASE;
        END PROCESS;
        TM1 : bomb_timer_main PORT MAP(clk, timer_start1, reset, count_r1,
        explosion1); -- Timer main for timer 1
        TM2 : bomb_timer_main PORT MAP(clk, timer_start2, reset, count_r2,
        explosion2); -- Timer main for timer 2
        TM5 : bomb_timer_main PORT MAP(clk, timer_start5, reset, count_r5,
        explosion5); -- Timer main for timer 5
        TM6 : bomb_timer_main PORT MAP(clk, timer_start6, reset, count_r6,
        explosion6); -- Timer main for timer 6
END ARCHITECTURE behaviour_bomb_handling;
```

6.1.7 Bomb Timer entity

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ENTITY bomb_overall IS
        PORT (
                clk        : IN std_logic;
                reset      : IN std_logic;
                bombp1     : IN std_logic;
                bombp2     : IN std_logic;
                explosion1 : OUT std_logic;
                explosion2 : OUT std_logic;
                explosion5 : OUT std_logic;
                explosion6 : OUT std_logic;
                explosion  : OUT std_logic);
END bomb_overall;
```

6.1.8 Bomb Timer behavioural description

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
ARCHITECTURE behaviour OF bomb_overall IS
        COMPONENT buffer_bomb
                PORT (
                        clk      : IN std_logic;
                        reset    : IN std_logic;
                        b_input  : IN std_logic;
                        b_output : OUT std_logic);
        END COMPONENT;
        COMPONENT bomb_handling
                PORT (
                        clk        : IN std_logic;
                        reset      : IN std_logic;
                        bombp1     : IN std_logic;
                        bombp2     : IN std_logic;
```