

BPTI: Bomberman in VHDL

7. Februar 2018

Teil I. Beschreibung der Entities

1. bomberman_ent

bomberman_ent ist die Top-Level-Architektur. Die Eingänge sind die tatsächlichen Eingaben des Nutzers und auf den Ausgängen liegen die Informationen die für den VGA-Anschluss benötigt werden. in bomberman_ent werden drei Entities miteinander verbunden. Zu erst die sync_gen_ent, die hsync und vsync erzeugt. Außerdem die game_mechanic_ent, welche die eigentliche Spiellogik enthält. Zuletzt noch die graphics_ent, die für die Erzeugung der entsprechenden RGB-Farbwerte zuständig ist. In bomberman_ent werden außerdem die beiden Generics TILE_SIZE und PLAYER_SIZE definiert und mit 32 initialisiert.

2. game_mechanic_ent

game_mechanic_ent ist die Architektur, welche die eigentliche Spiellogik enthält. Hier werden zwei Spieler vom Typ player_ent erzeugt und mit der Logik in game_state_ent verbunden.

2.1. game_state_ent

game_state_ent ist einerseits für die Verwaltung des Spielfeldes, aber auch für die Kollisionen verantwortlich. Unser Spielfeld besteht aus insgesamt 15 Zeilen mit jeweils 15

Tabelle 1: Spielfeld-Kodierungen

TileID	Bezeichnung
0x0	Leeres Feld
0x1	Explosion
0xD	Bombe
0xE	Zerstörbarer Block
0xF	Unzerstörbarer Block

Spalten, wobei jedes Feld durch vier Bits kodiert wird. Die genauen Kodierungen der einzelnen Blöcke kann man Tabelle 1 entnehmen. In dieser Entity wird der Anfangswert für alle Zeilen des Spielfeldes bestimmt, die durch das Ablaufen des Spiels verändert werden können.

Sollte ein Spieler seine Bombe legen, bekommt `game_state_ent` die Position der Bombe, sowie die Information, dass die Bombe gelegt wurde. Daraufhin wird die entsprechende Zeile so geändert, dass die Bombe auch tatsächlich sichtbar ist. Nachdem der Zähler der Bombe abgelaufen ist und sie explodieren soll, wird das entsprechende `explode` Signal gesetzt. Dann werden die umgrenzenden Blöcke der Bombe überprüft und auf Explosionen gesetzt. Die Blöcke müssen zuerst überprüft werden, um zu verhindern, dass unzerstörbare Blöcke von den Explosionen zerstört werden. Wenn der entsprechende Explosions Zähler in der Bombe abläuft, wird das `explosion` Signal auf 0 gesetzt und die Felder mit Explosionen werden auf leere Felder geändert.

Desweiteren werden hier die Kollisionen der Spieler erkannt. Sollte sich ein Spieler auf einem Feld befinden, auf dem momentan eine Explosion ist, wird das `enable` Signal des Spielers auf 0 gesetzt und er "stirbt". Dabei verwenden wir jeweils den Mittelpunkt des Spielers als Vergleichswert, um zu entscheiden ob er stirbt oder nicht.

2.2. player_ent

2.2.1. movement_ent

2.2.2. clk_movement_ent

Die Entity `clk_movement_ent` ist ein Clock Divider, die Architecture wird funktional beschrieben. Die Entity wird genutzt, um die Geschwindigkeit der Bewegung der Spieler zu steuern. Es wird alle 78125 Originaltakte die Ausgabe Ausgabe negiert. Da in unserem Design die VGA-Clock genutzt wird, wird somit alle $\frac{78125 \cdot 2}{25.175 \cdot 10^6} s \approx 6.2ms$ ein Takt erzeugt. Dies hat zur Folge, dass bei ständiger Bewegung eines Spielers in die gleiche Richtung, die pro steigender Takt-Flanke des ausgegebenen Takts stattfindet, die Bewegung vom

Tabelle 2: Sprite-Kodierungen

SpriteID	Bezeichnung
0x0	Leeres Feld
0x1	Explosion
0x2	Spieler 1
0x3	Spieler 2
0x4	Kodierung für weißen Rand
0xD	Bombe
0xE	Zerstörbarer Block
0xF	Unzerstörbarer Block

linken bis zum rechten Rand des Spielfelds (480 Pixel) $480 * 6.2ms \approx 2.98s$ dauert. Diesen Wert haben wir bei BombermanGB über einen Emulator gemessen und „rückwärts“ den nötigen Zähler, also 78125, bestimmt.

2.2.3. bomb_ent

3. graphics_ent

graphics_ent kapselt die Farbenerzeugung. Als Eingabe bekommt es die Informationen über die Positionen der Spieler sowie über das Spielfeld. Aus diesen Werten, wird durch pixel_gen_ent entschieden welche Sprites gerade verwendet werden müssen. Anschließend werden die ausgelesenen Farbinformationen in rgb_assign_ent auf die einzelnen Farbausgänge gelegt.

3.1. pixel_gen_ent

pixel_gen_ent bekommt als Eingaben die Informationen über die Spieler, sowie über das Spielfeld. Abhängig von diesen Informationen gibt die Entity aus, auf welche Sprites zugegriffen werden soll und an welcher Stelle.

Die 160 Spalten am linken Rand des Monitors werden bei uns dauerhaft weiß gefärbt, da wir ein quadratisches Spielfeld wollten, was allerdings bei einer Auflösung von 640x480 nicht funktioniert. Unser tatsächliches Spielfeld ist somit nun 480x480 Pixel groß.

Der erste Spieler hat bei uns den Vorrang vor dem zweiten Spieler, also falls sich beide Spieler übereinander befinden, wird der erste Spieler gezeichnet.

Wir haben unterschiedliche Ausgänge für die Informationen über die Sprites für Spielfeld und Player, um im PlayerROM entscheiden zu können ob wirklich der Spieler oder doch

das Spielfeld gezeichnet wird, falls die Grafik des Spielers an der bestimmten Position transparent ist.

3.2. Sprites

In den beiden Entities PlayerROM und SpriteROM befinden sich Sprites, die in Abhängigkeit des aktuellen Zustands der Arena, der Spieler und der Bomben RGB-Werte an den VGA-Controller ausgeben. Dazu befinden sich in den beiden Entities PlayerROM und SpriteROM pro Sprite (es existieren Sprites für: Spieler 1, Spieler 2, leerer Block, zerstörbarer Block, unzerstörbarer Block, Bombe, Explosion) ein zweidimensionales konstantes Array (wird synthetisiert als ROM). Diese Arrays sind jeweils $32 * 384$ Bit groß. Die Größe ergibt sich folgendermaßen: Die Kacheln des Spielfeldes sowie die Quellsprites haben eine Größe von $32 * 32$ Pixel. Da für jeden Farbkanal des VGA-Ausgangs 4 Bit zur Verfügung stehen haben wir in dem Array den RGB-Wert jedes Pixels mit $3 * 4$ Bit (also 3 Hex-Werte) kodiert. So kommen 32 Zeilen mit jeweils $3 * 4 * 32 = 384$ Bit zustande. Die Sprites haben wir aus einem Sprite-Sheet von opengameart (https://opengameart.org/sites/default/files/DungeonCrawl_ProjectUtumnoTileset.png) ausgeschnitten und mithilfe eines Java-Programmes, welches im Abgabe-Verzeichnis zu finden ist und im folgenden kurz beschrieben wird, in VHDL-Arrays umgewandelt.

3.2.1. SpriteExtractor.java

Das Java-Programm bekommt als Kommandozeilenargumente die Pfade zu den $32 * 32$ Pixel großen Sprites. Über jedes der Sprites wird pixelweise iteriert und für jeden Pixel der RGB-Wert ausgelesen. Daraus werden die Werte der einzelnen Farbkanäle bestimmt, welche anschließend normiert werden, da auf dem FPGA nur 4 Bit pro Farbkanal nutzbar sind. Die normierten Werte werden in Hex-Character umgewandelt und auf der Konsole ausgegeben, sodass die RGB-Kanäle jedes Pixels mit 3 Hex-Charactern dargestellt werden. Die Ausgabe erfolgt in der Form von 32 komma-separierten std_logic_vectoren der Länge 384.

3.2.2. SpriteROM

Die Entity SpriteROM wird mit einer Architecture funktional beschrieben. Abhängig von der sprite_id wird mittels sprite_row und sprite_col asynchron auf die verschiedenen Arrays, die Sprites für den leeren/zerstörbaren/unzerstörbaren Block enthalten, zugegriffen. Die sprite_id orientiert sich hierbei an den Kodierungen des Spielfeldes (siehe game_state, z.B. leerer Block = x0). Ist keine gültige id gesetzt (also $id > x1$ und $id < xD$), wird RGB schwarz ausgegeben.

3.2.3. PlayerROM

Die Entity PlayerROM ist hinter SpriteROM geschaltet und überschreibt asynchron die RGB-Werte, die von SpriteROM ausgegeben werden, wenn sich im aktuellen Pixel des Bildes ein Spieler befindet. In PlayerROM befinden sich zwei konstante Arrays, die in der gleichen Form wie in SpriteROM die Sprites für die Spieler enthalten. PlayerROM bekommt zum Auslesen der Arrays eine id (x2 für Spieler 1, x3 für Spieler 2), sowie die x- und y-Position des Spielers im aktuell betrachteten Pixels. Ist die id gültig, wird der RGB-Wert des entsprechenden Sprites gelesen und ausgegeben. Überlappen sich beide Spieler, so wird stets Spieler 1 gezeichnet. Ist keine gültige id gesetzt (also $id > x3$ oder $id < x2$), wird der RGB-Wert von SpriteROM durchgeschaltet. Hat ein Pixel den (in unseren Sprites speziellen) RGB-Wert xEEE wird ebenfalls der RGB-Wert von SpriteROM durchgeschaltet. Dies sorgt dafür, dass die Spielfiguren erkennbare Strukturen haben (Spieler sind keine Kacheln) und sich sichtbar vor einem Hintergrund bewegen.

3.3. rgb_assign_ent

In der Entity rgb_assign_ent werden die Farbausgänge für den Monitor gesetzt. Dazu wird bitweise auf die RGB-Eingangsvektoren zugegriffen und die einzelnen Ausgangsbits gesetzt. Dieses Vorgehen am Ende der Bearbeitung erlaubt es uns während den Berechnungen mit drei Vektoren für die Farben zu arbeiten, anstatt immer 12 einzelne Bits zu benutzen.

4. sync_gen_ent

Die Entity sync_gen_ent wird mit einer Architecture strukturell beschrieben durch die Unterkomponenten hsync und vsync. In dieser Entity werden die Signale für den VGA-Controller generiert (also hsync und vsync). Desweiteren werden zum Setzen der Pixel die aktuelle row und column ausgegeben. An dieser Stelle sei auf das VGA-Übungsblatt und den Bericht darüber verwiesen. Wir haben unsere Lösung von dem Übungsblatt für das Projekt übernommen und im Laufe des Projekts nichts an der Entity inklusive Unterkomponenten geändert.

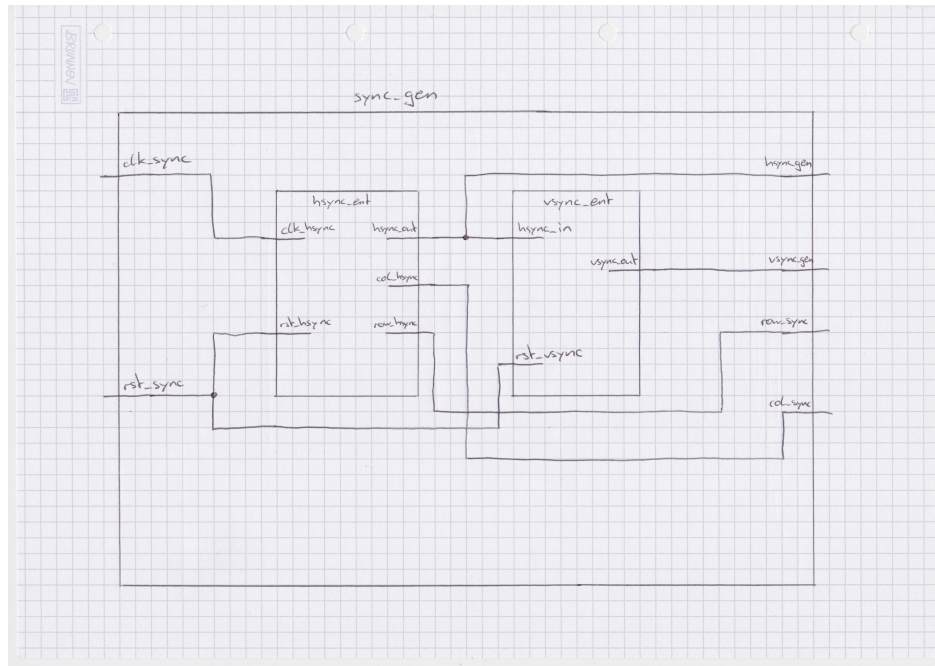


Abbildung 1: Blockdiagramm sync_gen_ent

Teil II.

Probleme

5. game_mechanic_ent

5.1. game_state_ent

5.2. player_ent

5.2.1. movement_ent

5.2.2. clk_movement_ent

5.2.3. bomb_ent

6. graphics_ent

6.1. pixel_gen_ent

6.2. Sprites

6.2.1. SpriteExtractor.java

6.2.2. SpriteROM

6.2.3. PlayerROM

7. sync_gen_ent

Ein Problem, das schon bei der Lösung des Übungsblatts aufgetaucht ist, hat sich auch durch das Projekt gezogen. Die VGA-Ausgabe funktioniert nicht an allen Monitoren

ordnungsgemäß (flackern). An unserem Test-Monitor hat die VGA-Ausgabe nach einigen Einstellungen jedoch funktioniert.