**EEL 5722 - FPGA DESIGN FINAL PROJECT:**

**Bomberman FPGA Video Game**

Contents

[Introduction 3](#_Toc522455437)

[Your Objective 4](#_Toc522455438)

[Your VGA Module 5](#_Toc522455439)

[Getting Started 6](#_Toc522455440)

[Graphics Memories: Pillar Distributed Memory ROM 7](#_Toc522455441)

[Arena & Pillars 10](#_Toc522455442)

[Bomberman Sprites 12](#_Toc522455443)

[Generating the Bomberman sprite Block RAM 13](#_Toc522455444)

[Bomberma+n Module 14](#_Toc522455445)

[Motion Timer Register 16](#_Toc522455446)

[Sprite X,Y Coordinate Register 16](#_Toc522455447)

[Pillar Collision Signals 17](#_Toc522455448)

[Sprite X,Y Coordinate Registers Next State Logic 19](#_Toc522455449)

[1. ) Animate Bomberman: Frame Timer & ROM Offset Registers 21](#_Toc522455450)

[Instantiate Sprite ROM and Indexing 22](#_Toc522455451)

[Outputs 23](#_Toc522455452)

[Block Module 24](#_Toc522455453)

[Creating the Dual Port RAM 25](#_Toc522455454)

[Block Module IO 26](#_Toc522455455)

[Collision Detection With Blocks 26](#_Toc522455456)

[Block Sprite ROM & block\_map RAM 27](#_Toc522455457)

[bm\_blocked 28](#_Toc522455458)

[Bomb Module 29](#_Toc522455459)

[IO](#_Toc522455460) 30

[Bomb Coordinates 3](#_Toc522455461)1

[Bomb & Explosion Counters 3](#_Toc522455462)1

[2.) Bomb and Explosion FSM 3](#_Toc522455463)2

[ROM indexing and Outputs 3](#_Toc522455464)5

[Enemy Module 3](#_Toc522455465)6

[IO 3](#_Toc522455466)6

[3.) Linear Feedback Shift Register 3](#_Toc522455467)6

[4.) Enemy Finite State Machine 3](#_Toc522455468)7

[Lives and Score: game\_lives Module 3](#_Toc522455469)9

[Score Display & binary2bcd Conversion](#_Toc522455470) 40

[5.) Binary to BCD 4](#_Toc522455471)1

[Numbers ROM 4](#_Toc522455472)2

[Top Module Description 4](#_Toc522455473)4

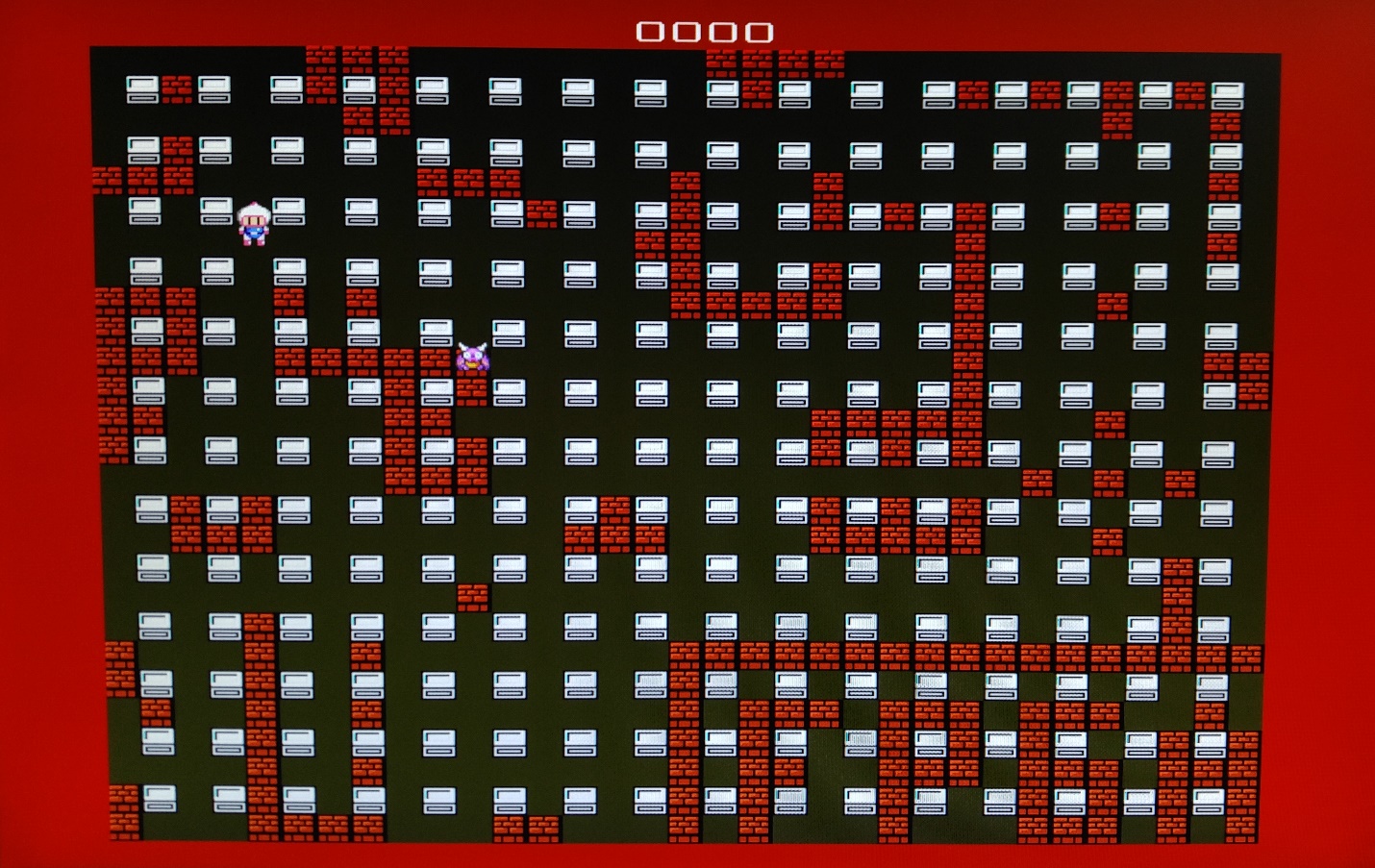
[6. ) Bomberman Life Bar 4](#_Toc522455474)6

7. ) Your Creative Expansion………………………………………………………………………47

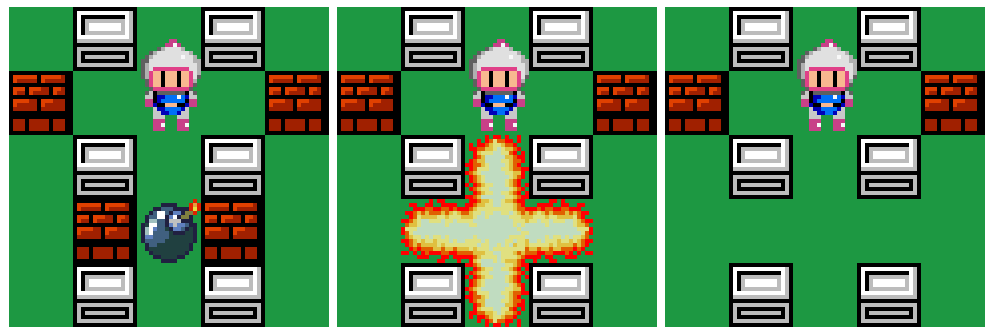
[Grading Rubric 4](#_Toc522455475)7

# Introduction

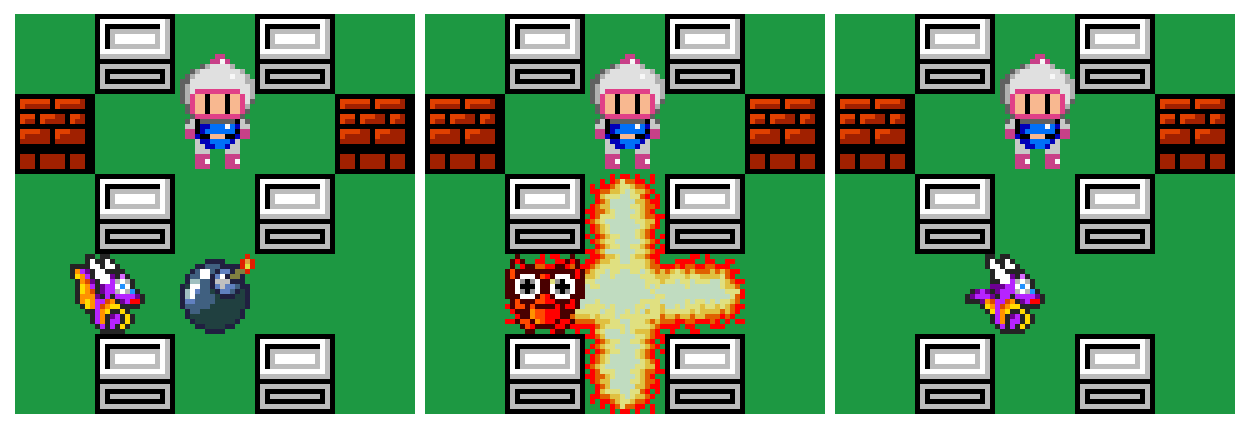
This project focuses on implementing a digital system on an FPGA that plays a simple game that is similar to Bomberman.



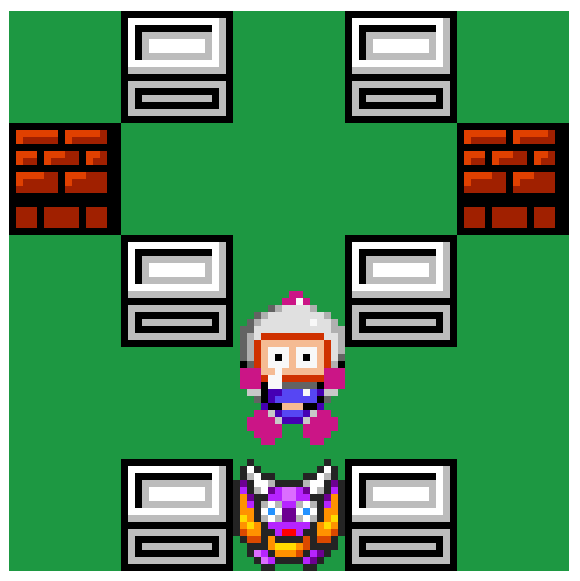
Bomberman walks around an arena filled by an array of indestructible white pillars, and blocks made of bricks that can be cleared by laying a bomb down and exploding them. Bomberman can lay one bomb at any time, which takes a moment to detonate, and then explodes into the surrounding four tiles on screen.



The enemy in the game is a flying dragon that moves around the arena with a mind of its own. If Bomberman lays a bomb skillfully, he can hit the dragon with its explosion, thereby earning points in the game, and making the dragon move faster and more erratically.



If the dragon intersects Bomberman or Bomberman is hit by the explosion of his bomb, he will lose a life. Bomberman starts with five lives, indicated by the shade of red surrounding the arena. As Bomberman loses lives the red will become darker, until he loses all lives and it becomes black. At this point the game is over.

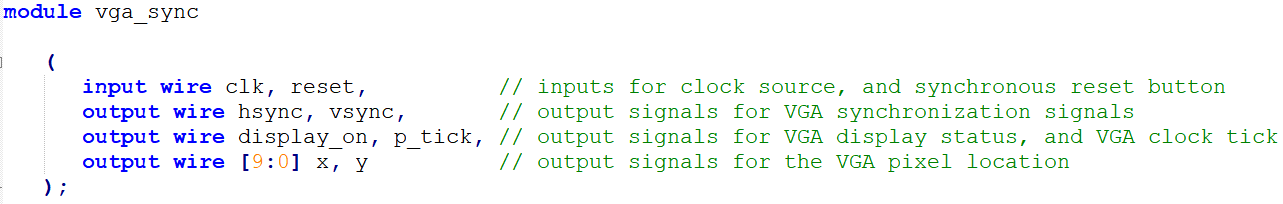


# Your Objective

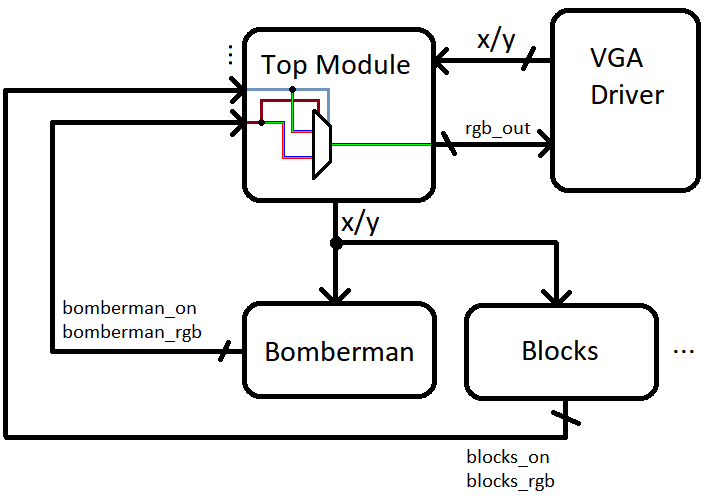
Your goal in this project is to complete certain aspects of the game’s various modules for it to function as intended. Along the way, the functionality of the provided HDL of each core will be outlined for you to learn from and build upon. **Tasks for you to complete will be coded red, with a matching number label found in the rubric at the end of this file.**

# Your VGA Module

The game is displayed on a VGA monitor using the VGA module developed in the class lab assignment. Ensure that your VGA module outputs a pixel clock, as well as the (x,y) coordinate of the pixel currently being drawn on the screen, following the IO format below:



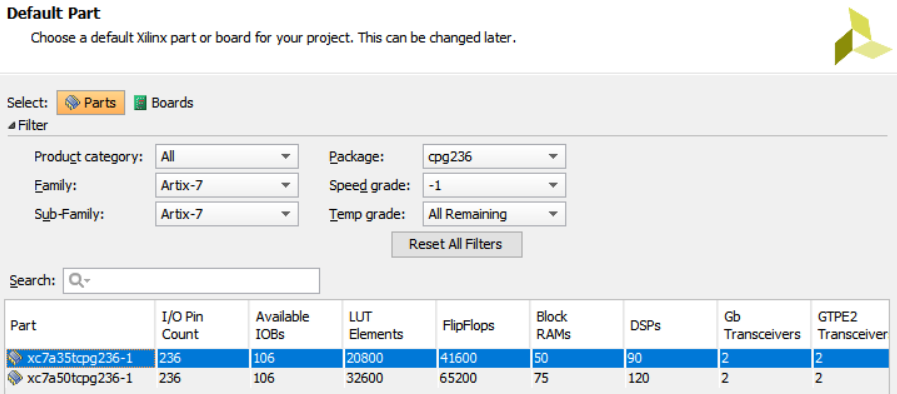
There is no frame buffer that stores all the RGB color data that is displayed on screen. Instead each element in the game (Bomberman, enemy, bomb, blocks, etc.) which has its own module receives an input of the current x/y coordinate of the pixel being drawn on screen. If the module knows that the pixel is within where its sprite should be drawn on screen it will assert an “on” signal stating that it is currently on. The top module receives this asserted signal and routes out that module’s RGB color data to the VGA driver. In this sense, the color data is dynamically drawn on screen, and the space needed to store a frame buffer is saved.



# Getting Started

The current hardware being targeted for this project is the Basys 3 FPGA board that is used in the lab.

In Vivado, create a new RTL project named bomberman\_basys3, ensuring that the following parameters are selected for the Part:



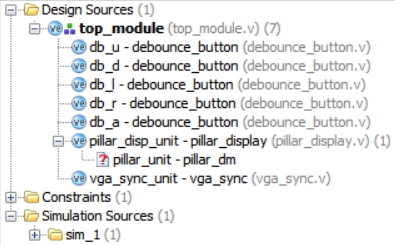
Add Sources from the HDL folder in the project files folder of:

1. top\_module.v
2. debounce\_button.v
3. pillar\_display.v
4. your VGA module
5. basys3.xdc

The **top\_module** handles IO to the FPGA peripherals, instantiates all the modules for the system and routes signals between them, passes color data from modules that generate it to the VGA driver when appropriate, and other small tasks that wrap the entire game up and make it functional.

For you, the **top\_module** is grown as this project progresses, and you will be uncommenting module instantiations as the project proceeds. The top module as given has almost everything ready to get started with the **pillar\_display** module.

The hierarchical view of the project so far should look as shown below:



You will see that there is one missing piece with a red question mark.

In this project we will use distributed and block memory to instantiate ROMs and a RAM.

# Graphics Memories: Pillar Distributed Memory ROM

In the graphics folder you will find images with associated coefficient files that are to be used to create prefilled memory elements.

The format of a .coe file is shown below:



with the final coefficient terminated with a semicolon.

The provided **image\_2\_coe.py** python script was used to read in each image and generate the coefficient file with 12-bit RGB color data for each pixel in the image. It is important to note that the image is read into the coefficient file in **Row-major order**.

In this section it will be demonstrated how to create the **distributed ROM** for the pillar ROM used in the **pillar\_display** module of the following section.

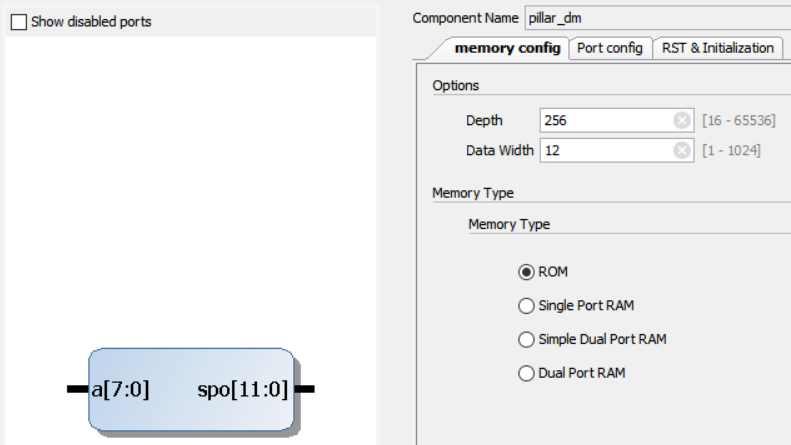
From the IP Catalog, select Distributed Memory Generator.

Name the component **pillar\_dm**.

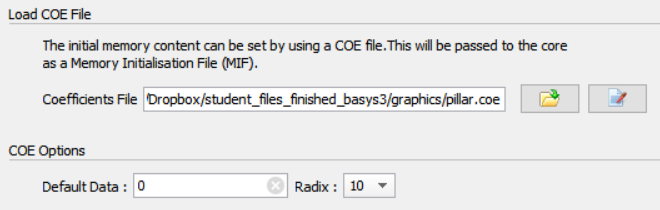
The pillar image is 16x16, so the depth of the ROM is 256.

The width of the data is 12 bits.

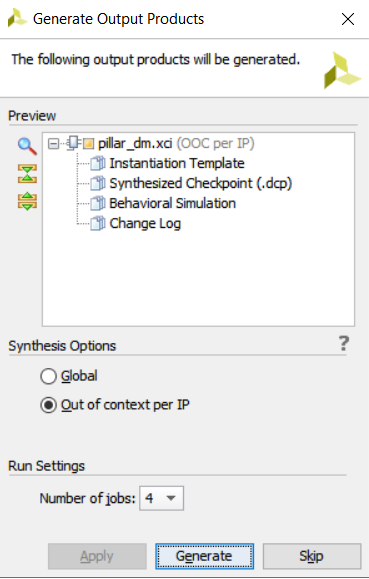
Select ROM for memory type.



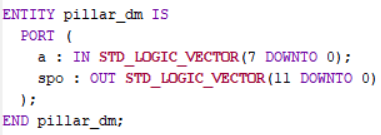
On the RST & Initialization tab, browse and load the **pillar.coe** file.



Press Ok and then Generate.



The generated interface to the memory is simple:



The address into the ROM is **a**, 8-bits for a depth of 256 entries, with an output **spo** of width 12 for the 12-bit color data stored within.

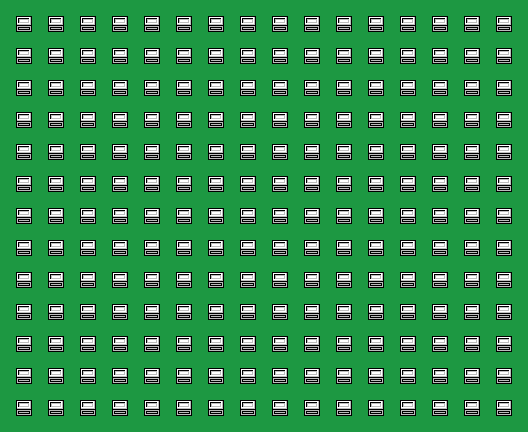
Now that the source hierarchy in Vivado is complete, take a moment to run Synthesis, Implement the design, generate the bitstream, and load it onto the FPGA.

You should see the gameplay arena centered on the monitor with an array of pillar tiles on screen.

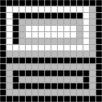
\*\*\***Note that the left most slide switch SW15 should be down to disable the reset.**

# Arena & Pillars

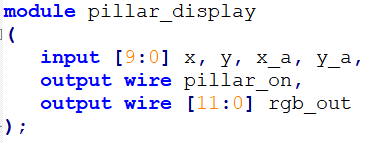
The gameplay **arena** is shown below. It has dimensions of 528x432 pixels. This will be the area that Bomberman and enemies will move within.



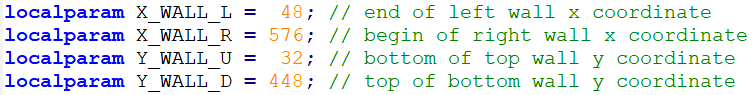
A regular grid of stationary and **indestructible** pillars is formed in the arena. Each pillar has dimensions of 16x16 pixels. The arena has length and width dimensions that are evenly divisible by 16, and the display and collision detection with the grid of pillars will take advantage of this.



The module **pillar\_display** contains combinational logic that determines where on the screen the pillars are drawn. It receives inputs for the (x, y) pixel location that is being set on screen by the VGA module. When the pixel location on screen is within where a pillar should be in the arena, the **pillar\_on** signal is asserted, and the appropriate RGB data indexed from the pillar ROM is routed out of the module.



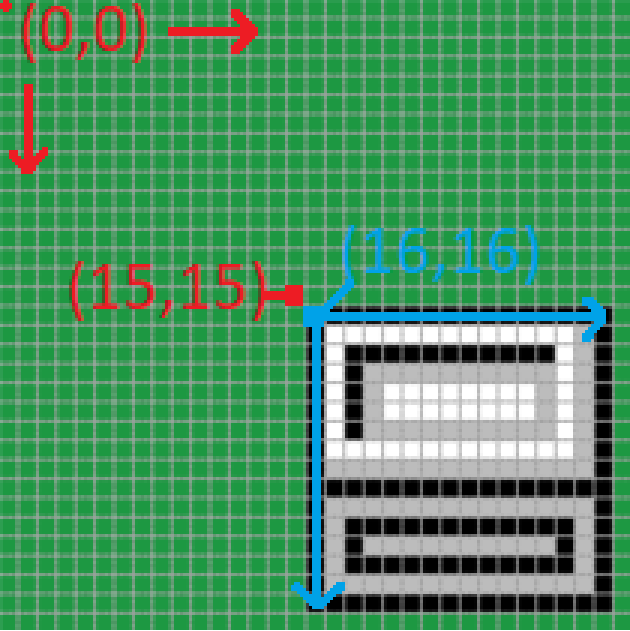
The VGA pixel (**x**, **y**) coordinates begin at the upper left corner of the entire screen. Since the arena is offset in the center of the entire screen with black borders, new coordinates (**x\_a**, **y\_a**) are generated which account for the offset and begin with (0, 0) at the top left corner of the arena. These are generated in the top module and routed in. When this offset calculation is performed, it will be described by saying that the coordinates were transformed to **arena coordinates**, or into the **arena coordinate frame**.



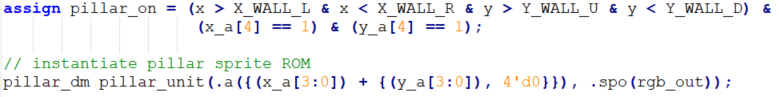
For a pillar to be placed in the arena in a regular grid (i.e. **pillar\_on** output asserted):

1. VGA pixel coordinates (x, y) must be in bounds of the arena.
2. Both (**x\_a**, **y\_a**) coordinates when divided by 16 must yield an odd number (i.e. 5th bit must be 1).

This second criteria is demonstrated in the diagram below from the upper left corner of the arena where (**x\_a**, **y\_a**) = (0, 0).



The x\_a and y\_a coordinate’s lower 4 bits are used to index into the instantiated pillar ROM as shown below, where “\*16” is replaced by “<< 4”.



# Bomberman Sprites

The Bombermans sprites used to animate the main character on screen are stored in block RAM on the FPGA to form a ROM. The coefficients are derived from a 16x250 image, where each of the 10 sprite frames is individually 16x25 pixels. A python script reads in this image and flattens the image coefficients to output a .coe file where the coefficients are in the row-major order format ({row\_0}, {row\_1}, … , {row\_299}) where each row contains the 16 pixel values across the row from left to right.

Indexing into the ROM to a specific sprite frame is done by using an offset multiple of 25 to get down to the start of the frame. Where the Bomberman sprite is on screen and current pixel coordinates of the VGA pixel being drawn on screen are related to fully index into the ROM:

**.addra( (x – x\_b) + (y – y\_b + offset) << 4)**

**= .addra( (x – x\_b) + {(y – y\_b + offset), 4’d0})**

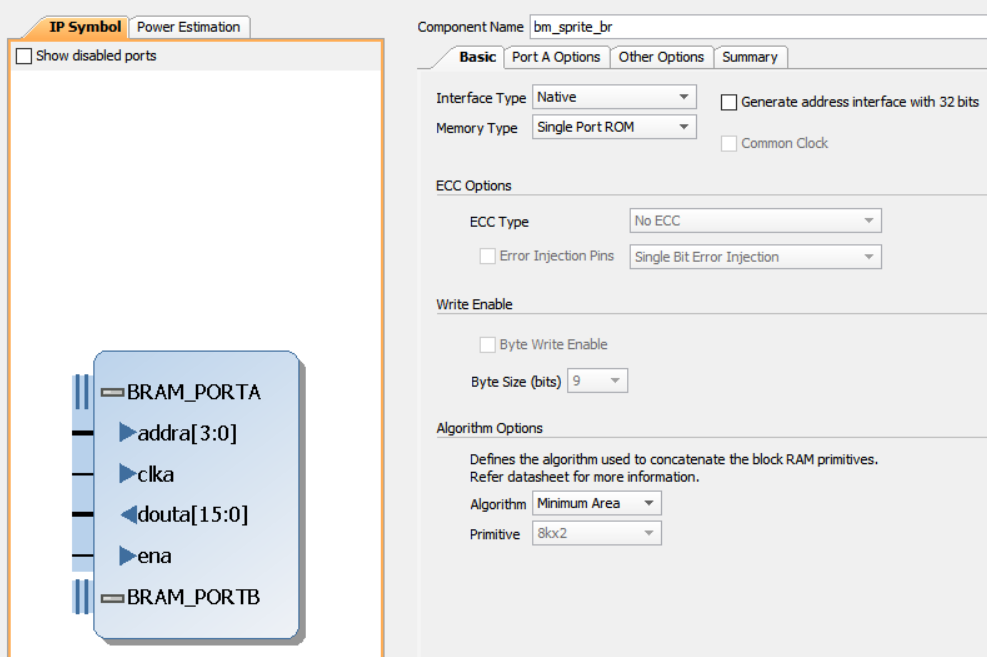
Where (**x**, **y**) are VGA pixel coordinates being drawn on the screen, (**x\_b**, **y\_b**) are Bomberman’s sprite coordinates denoting the upper left of the sprite (to be detailed later), and offset is the value that determines where to start indexing into the ROM or what sprite to draw on screen.

# Generating the Bomberman sprite Block RAM

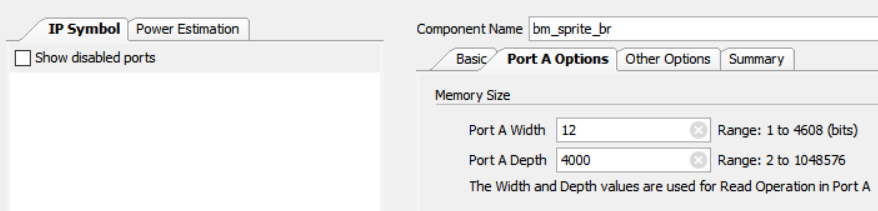
In the IP catalog open the Block Memory Generator

Under the basic tab yype in **bm\_sprite\_br** for the component name.

Select Single Port ROM for Memory Type, with native interface.



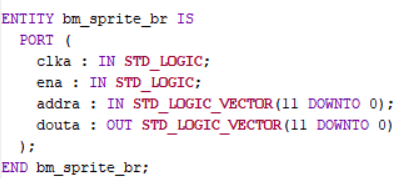
Under the Port A Options tab, use a Width of 12, and Depth of 16x250 = 4000.



Fron the Other Options tab, check “Load Init File” and browse for the **bomberman\_sprites\_10.coe** file. Check to fill remaining memory locations with 0.

Press OK and generate the ROM.

The interface has a clock input **clka** which should be connected to the master clock. The address input **addra** addresses into the 4000 entry ROM, and the output **douta** outputs the 12-bit color data.



Add the source **bomberman\_module.v** to the project.

In the **top\_module** uncomment the instantiation of the **bomberman\_module**.

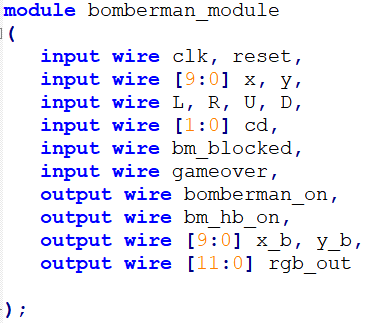
# Bomberman Module

The Bomberman module takes care of several tasks:

1. Display Bomberman on screen based on location coordinates of the sprite.
2. Update Bomberman’s location in response to user input and collision detection.
3. Animate Bomberman while he walks.
4. Index into Bomberman sprite ROM and output RGB color data appropriate for the current **x**/**y** VGA pixel location.
5. Output to other modules Bomberman’s location to be used for collision detection.

**This is the largest and most complex module in the whole game. It will be your job to implement the timing and indexing into the sprite ROM to properly animate the character on screen.**

**IO**



Inputs **L**, **R**, **U**, **D** come from the debounced user input of pushbuttons and tell this module which direction to move Bomberman on screen.

Input cd comes from the **top\_module** where the current direction is registered from the inputs **L**, **R**, **U**, **D**.

Input **bm\_blocked** is a signal from the **block\_module** that is asserted when Bomberman would collide with a block if he moved from his current location in his current direction and is used for collision detection. The **block\_module** and **bm\_blocked** will be discussed in detail later.

The input **gameover** is a signal asserted by the **game\_lives** module when the game is over and is used to shut off Bomberman’s motion in this module.

Output **bomberman\_on** is used in the top\_level module to coordinate when the VGA pixel drawing on screen is located within Bomberman’s sprite on screen, and therefore should display the Bomberman module’s RGB output signal, **rgb\_out**.

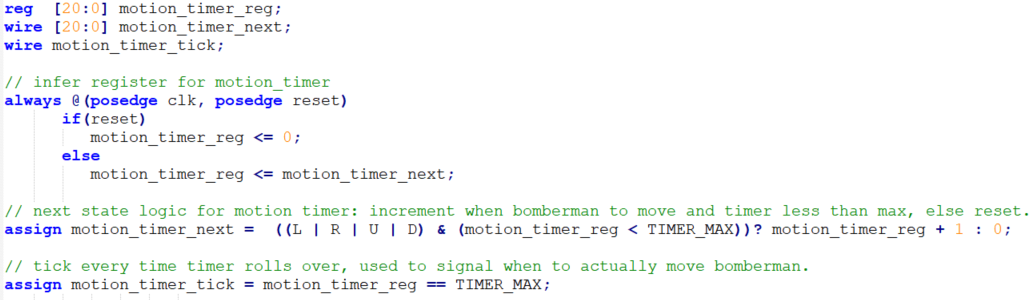
The output **bm\_hb\_on** is a signal that is asserted when the VGA pixel on screen is within Bomberman’s hitbox that is used for collision detection. This signal is used in the **game\_lives** module, as discussed later.

The output **x\_b**, **y\_b** is bomberman’s current position on screen.

## Motion Timer Register

To move the sprite of Bomberman across the screen in response to user input, its location coordinate registers (**x\_b**, **y\_b**) are incremented or decremented. For instance, if the user input **R** is asserted, Bomberman’s sprite location should move right, with **x\_b** incrementing. For this motion to be on human appreciable time scales, a timer is used to count for a specific period before incrementing the sprite location by one step.

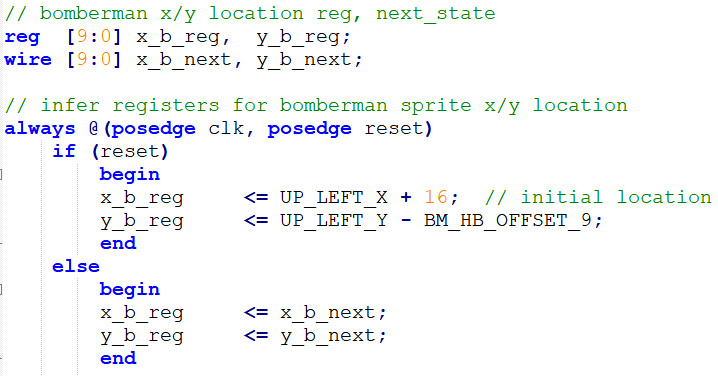
When a user direction input is asserted, the motion timer register counts from 0 to **TIMER\_MAX**, upon which a “tick” signal is asserted and the register resets to 0. When no user direction input is asserted the timer resets to 0.



The **motion\_timer\_tick** signal is then used to update the sprite location at a normal rate on screen. For instance, if the FPGA runs at 100 MHz, and TIMER\_MAX = 1,200,000, then the period of the counter is 0.012 seconds, which means in one second, the sprite would move 83 pixels. This is arbitrary and can be adjusted until Bomberman moves at a speed that is appropriate for gameplay.

## Sprite X,Y Coordinate Register

Two registers are inferred to hold the x and y coordinates of the Bomberman Sprite on the screen. The coordinate location is defined as the upper-left corner of the sprite.



## Pillar Collision Signals

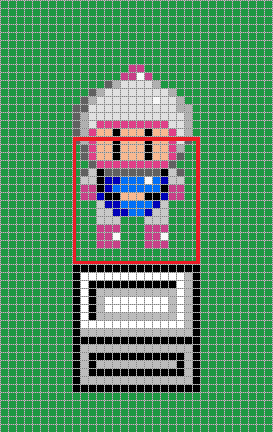
Bomberman’s sprite location should move when the user input signals a direction to move, the motion timer register has ticked, and the sprite can move in that direction. The sprite cannot move if Bomberman would run into a pillar, Bomberman would leave the arena, or Bomberman would run into a block.

When developing a model for constraining the sprite motion across the arena to stay between pillars, a 16x16 hitbox is placed at the bottom of the sprite and is constrained to not intersect with where a pillar is drawn on screen, known as the x and y pillar space.



There are four signals, **p\_c\_up**, **p\_c\_down**, **p\_c\_left**, and **p\_c\_right**, which are asserted when a pillar collision would happen if the sprite moved one pixel in the corresponding direction. These asserted signals tell the sprite **x\_b\_reg/y\_b\_reg** coordinate registers to not increment or decrement the location if it would mean colliding with a pillar.

For instance, suppose the Bomberman sprite is facing down, and the hit box is right against a pillar below:



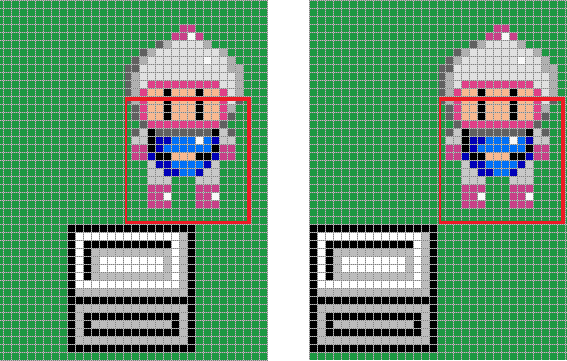
In this state, **p\_c\_down** would be asserted, and the sprite’s y location would not be allowed to increment and move down a pixel.

The x coordinates of the left/right edge of the box, and y coordinate below the bottom edge of the box can be tested for being within the space of a pillar (5th bit == 1, as explained in Arena & Pillars section).

If either left **or** right edge coordinates are within the pillars x space, **and** the y coordinate below the hitbox bottom are within the pillars y space, then the **p\_c\_down** signal is asserted.

The state shown below on the left would assert **p\_c\_down** signal, as the left edge x coordinate is within the pillar x space, and the y coordinate below the bottom edge is in the pillar y space.

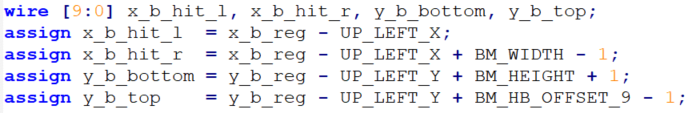
The state shown below on the right would not assert **p\_c\_down**. While the y coordinate below the bottom edge of the hit box is in the pillar y space, neither the left or right edge of the hit box are within the pillars x space.



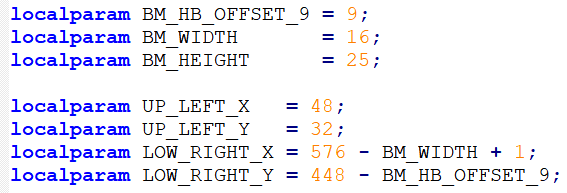
The **p\_c\_down** wire is assigned to the conditional statement:



Where the signals being compared in this conditional statement are defined as:



Where:

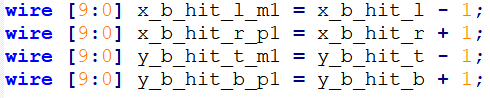


The **y\_b\_bottom** signal is the current sprite y coordinate, - 32 to translate into arena coordinates (see section Arena & Pillars), + 25 + 1 to bring the value down to the y coordinate below Bomberman’s hitbox.

The **x\_b\_hit\_l** and **x\_b\_hit\_r** are the x coordinates of the left and right sides of the hit box derived from the Bomberman sprite x coordinates, transformed into arena coordinates.

The pillar collision signals for up, left, and right are derived in a similar manner and are listed in the bomberman\_module source file in the Pillar Collision Signals section.

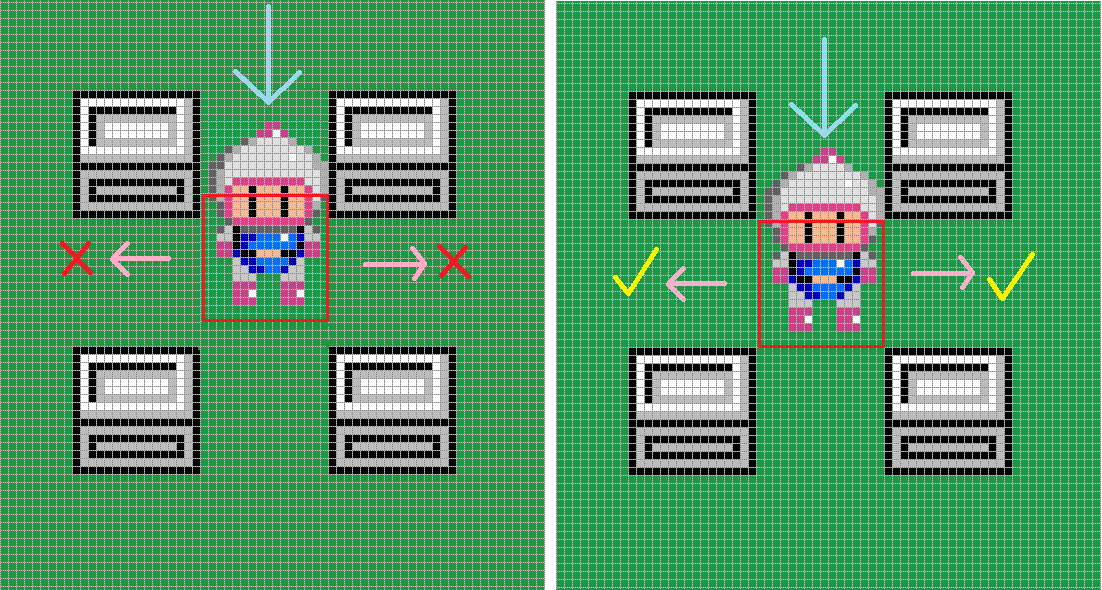
The x\_b\_hit and y\_b\_hit signals are further constrained as shown below, and are used in the final collision detection to avoid the corner case where Bomberman walks into block when going around a pillar.



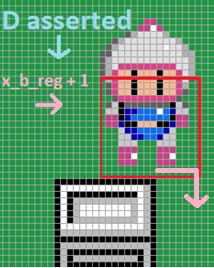
## Sprite X,Y Coordinate Registers Next State Logic

The pillar collision signals work to constrain the Bomberman sprite 16x16 hitbox to remain within the 16x16 gaps between pillars. Without this constraint it would be difficult for a user to accurately place the 16x16 hitbox between the 16x16 gaps to change from moving either up/down to left/right or vice versa.

For example, assume the Bomberman sprite is moving as shown in the image below, down a column. For the Bomberman sprite to move either left or right between the pillars, its hitbox must be aligned as shown. If it is not aligned with the left/right empty rows, a pillar collision signal will be asserted, and the Bomberman Sprite will not be allowed to move left/right. This makes sense, as the sprite hitbox is constrained to never collide with a pillar.

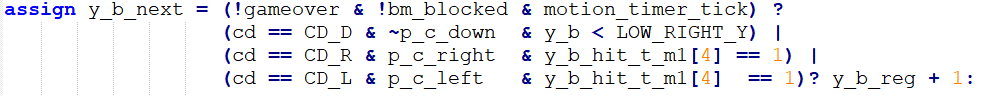


To make moving around pillars in the arena easy, Bomberman will slide adjacent to the pillar currently being run into to the nearest gap. In the instance shown below, when Bomberman walks down against the pillar, he will slide right until his hitbox clears the gap and can move down the column.



In short, this will allow Bomberman to automatically move around pillars when walking into them.

Considering just the next state logic that would allow y\_b for Bomberman moving down:

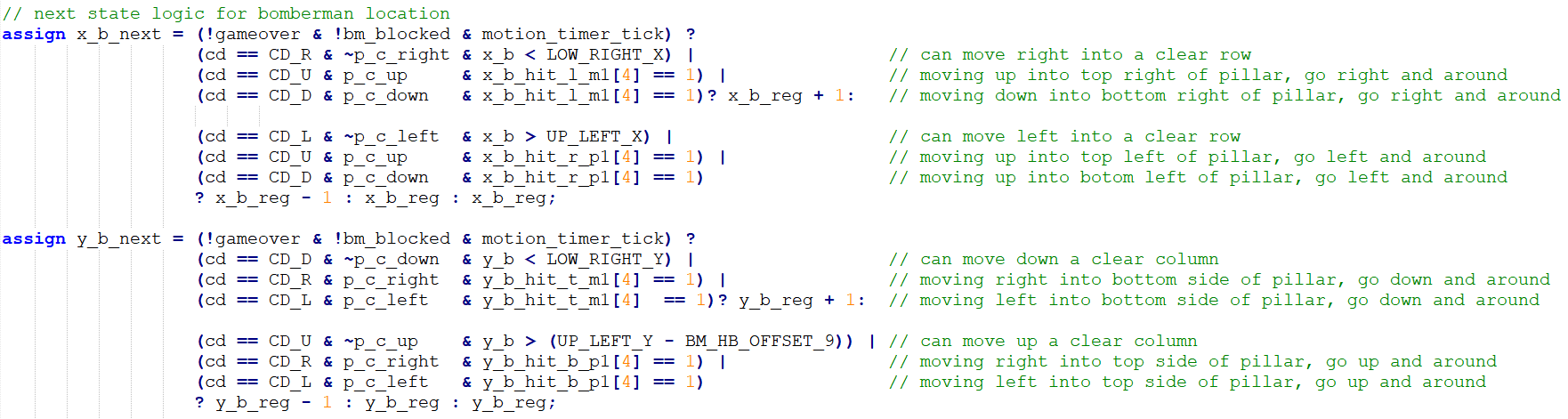


The first line checks if the game is not over, if Bomberman is not blocked by a block (as explained later), and the motion tick has been asserted. The second line allows for Bomberman to normally move down an empty column. If the current input is down (**CD\_D**), and there is no expected pillar collision, and the current coordinate isn’t at the bottom boundary of the arena, and the motion timer register ticks for the sprite update time delay, then increment the register by one and move Bomberman down.

The third line of logic allows for Bomberman to move down and around the bottom of a pillar if moving right into the bottom of a pillar: if the current input direction is right, and there is an expected pillar collision if moving right, and the top of the hit box is touching the pillar, and a motion timer tick, then move down, which will help Bomberman go down and around the pillar and move right.

The fourth line of logic allows for the same situation of going down and around the bottom of a pillar if moving left into it.

The next state logic for **x\_b** and **y\_b** to be incremented or decremented follows the same format:

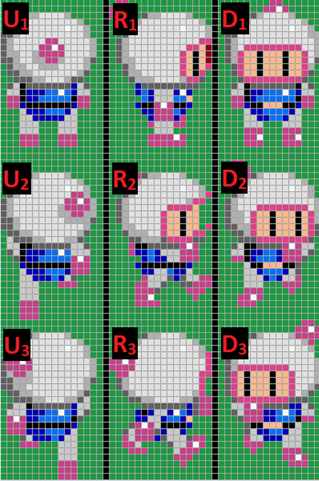


## 1. ) Animate Bomberman: Frame Timer & ROM Offset Registers

When Bomberman is not walking, the frame that is displayed on screen should be of him standing in the last direction input by the user, held in the current direction register input from the top module. This would be frames shown below, labelled U1, R1, D1, and frame R1 mirrored horizontally for left.

When any user direction input is asserted, a timer register called **frame\_timer\_reg** should count up, and cycle through a timing period of T = 50000000 cycles, with 4 sub periods of T/4. During the four sub periods, the frames drawn on screen should cycle between 1, 2, 1, 3, for the specified direction. This animates Bomberman’s movement on screen with a walking animation.

✎ Design the **frame\_timer\_reg** register and **frame\_timer\_next** next-state logic to satisfy the above requirements.



A rom offset register called **rom\_offset\_reg** should hold predefined offset values that are multiples of 25 up to 225, which are used to index into the ROM to the start of a particular frame to be displayed.

The next state logic for the **rom\_offset\_reg** should use the current direction register and frame timer register to determine which offset to use to draw the correct frame on screen.

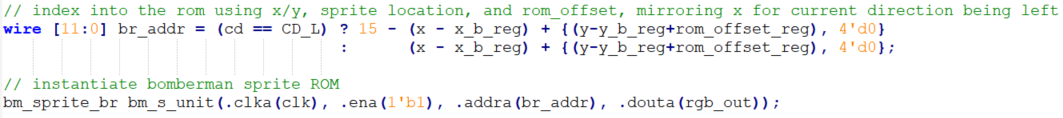
✎ Design the **rom\_offset\_reg**  register and **rom\_offset\_next** next-state logic to satisfy the above requirements.

With the above requirements undefined in the source, the **frame\_time\_reg** and **rom\_offset\_reg** are currently seen by synthesis as being unused, and are therefore automatically tied to 0, which causes the first frame in the ROM to be constantly displayed.

## Instantiate Sprite ROM and Indexing

The block RAM module for the Bomberman sprite, **bm\_sprite\_br**, is instantiated, with **clk** input routed to **clka**, indexing address **br\_addr** routed to **addra**, and color output **rgb\_out** taken from **douta**.

The indexing into the rom uses VGA **x**/**y** pixel coordinates, Bomberman coordinates **x\_b\_reg** / **y\_b\_reg** and the rom offset reg defined above. When the current direction is left, the indexing done in the x dimension is mirrored.



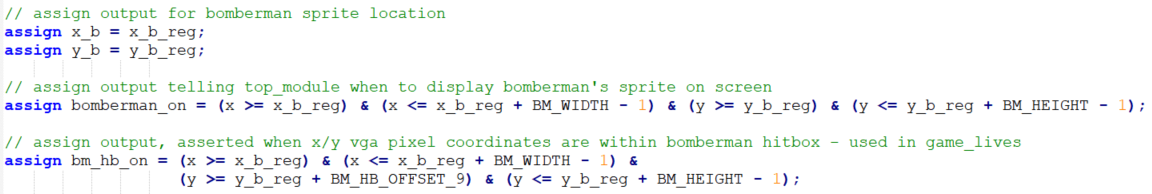
## Outputs

The module outputs are assigned at the end.

Bomberman location register is outputted to be used for collision detection in the **block\_module**.

The signal **bomberman\_on** is asserted when the current VGA pixel location **x**/**y** is located within the sprite drawn on screen and is used in the top module to route out the RGB color info to the VGA module when it is to be displayed on screen.

The signal **bm\_hb\_on** is asserted when the current VGA pixel location is located within the hitbox location on screen and is used in the **game\_lives** module for hit detection with the enemy and explosion, as explained later.



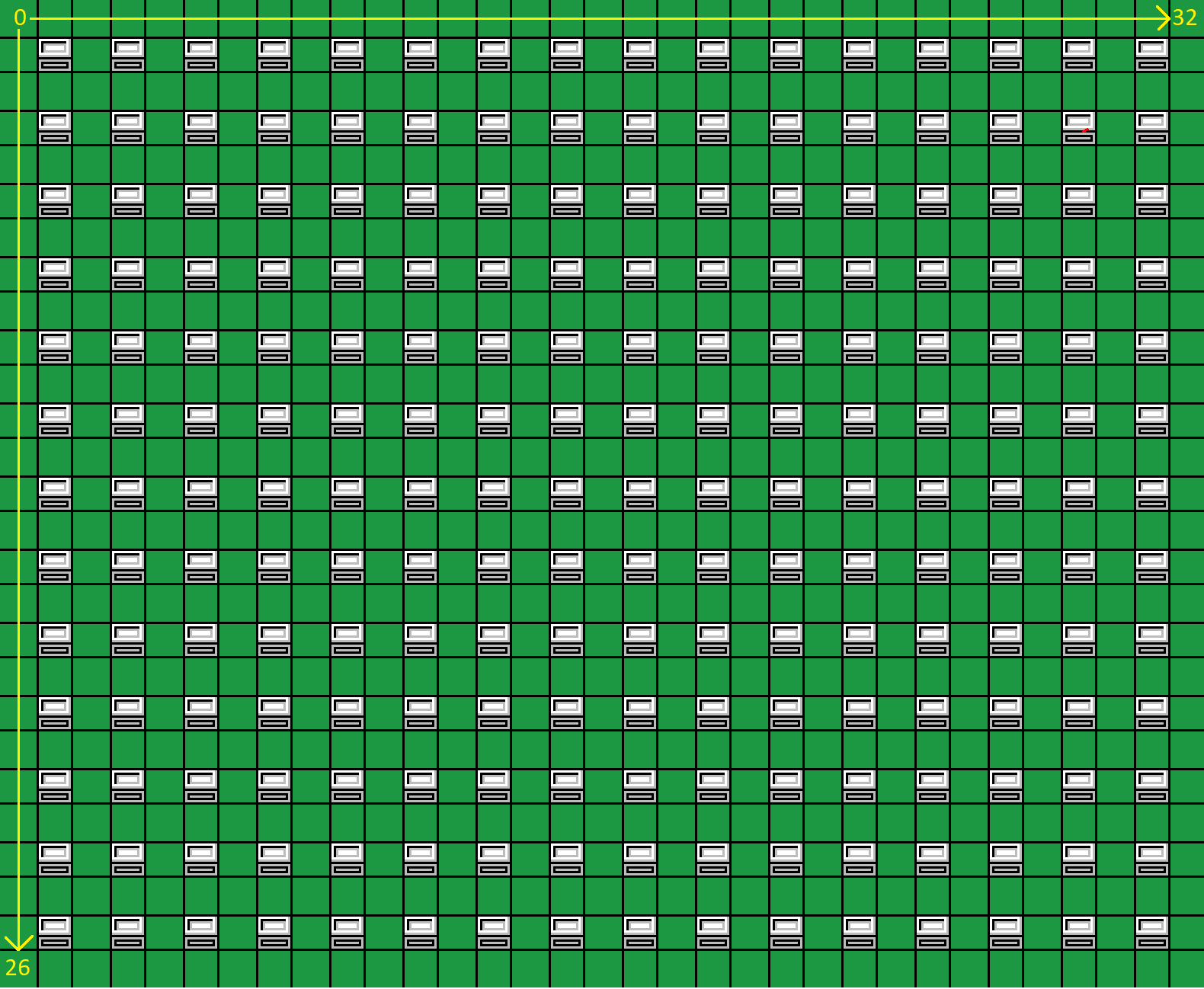
**With the Bomberman module up and working, Bomberman is able to easily walk around the arena. Let’s now explore how to place blocks on the screen that Bomberman will eventually contend with using bombs.**

# Block Module

In the game arena there are blocks that are placed in empty 16x16 spots which block Bomberman and can be destroyed by a bomb.



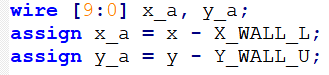
Since each block is 16x16, the arena can be divided evenly into an array of 33 columns and 27 rows. This will be referred to as the **arena block map coordinates**, or **ABM** coordinates. To transform x/y coordinates on the monitor to ABM coordinates, subtract the offset of the top left location of the arena to make the top left pixel of the arena 0. Next divide by 16, or use all bits above the lower 4, and this will give a number that fits within the ABM coordinate frame.



A dual port RAM is used to store 32x26 1-bit values that determine if a block is located at each spot in the arena. By converting the VGA **x**/**y** coordinates to ABM coordinates, the read index into the ROM is formed as:



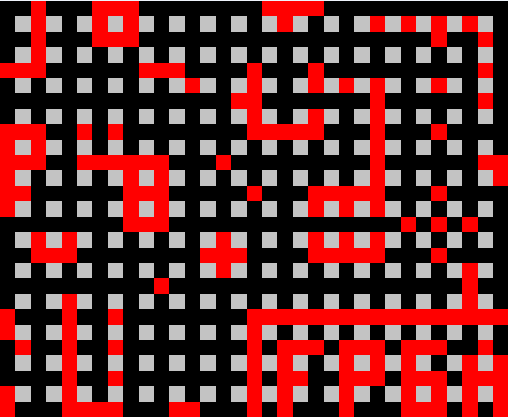
where



Add the source file **block\_module.v** to the project and uncomment its instantiation in the top module.

## Creating the Dual Port RAM

The dual port RAM used to store ABM locations is prefilled by the **block\_map.coe** file. The RAM contains 1-bit values for each ABM location, with 1 being a bomb, and 0 being empty. This coefficient file can be changed to create a new arrangement of blocks in the arena. It can be generated using the **block\_map\_2\_coe** Python code which works on the **block\_map.png** image, where red pixels denote blocks on the map.

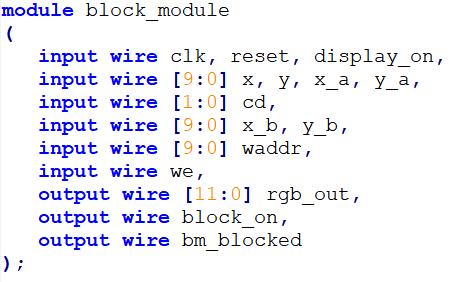


**block\_map.png**

To generate the dual port RAM, open the Distributed Memory Generator. Name the component **block\_map.** Use a data width of 1, and depth of 896. Include the coefficients file and generate.

Use the directions of creating a distributed memory ROM to create the ROM of the block sprite using the component name **block\_dm** from the **block.coe** file.

## Block Module IO



Inputs **x**/**y** are VGA pixel coordinates being drawn on screen.

Inputs **x\_a**/**y\_a** are VGA pixel coordinates in arena coordinate reference frame.

Input **cd** is registered current direction of Bomberman as given by control input.

Inputs **x\_b**/**y\_b** are coordinates of Bomberman on the screen, used for collision detection with blocks.

Input **waddr** is the write address to the **block\_map** RAM, used by the bomb\_module’s explosion tiles to clear blocks in RAM.

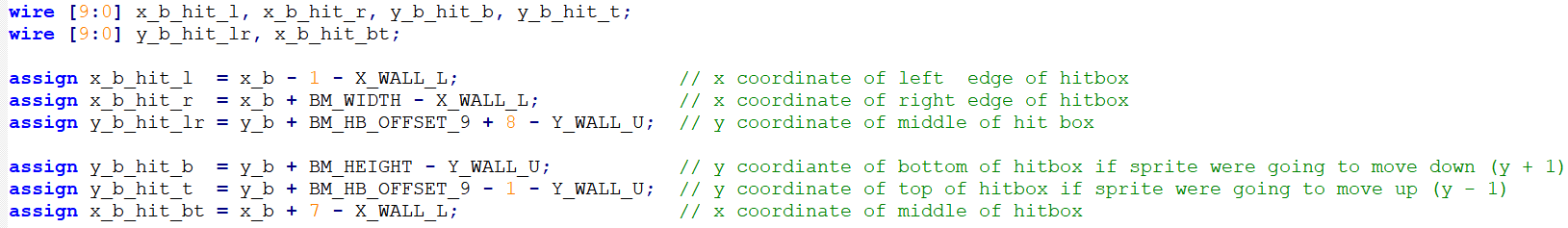
Input **we** is the write enable signal to the dual port RAM **block\_map**.

Output **rgb\_out** is the RGB color data for blocks to be drawn on screen when **block\_on** is asserted.

Output signal **bm\_blocked** is asserted when it is calculated using **x\_b**/**y\_b** and **cd** that a collision with Bomberman and a block is to occur.

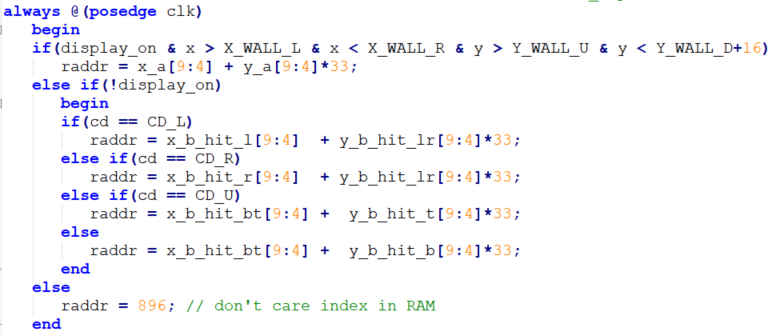
## Collision Detection With Blocks

To determine if Bomberman would collide with a block, moving in his current direction, signals are defined that represent the pixel location of the different edges of Bomberman’s hitbox moving in the different possible directions. These are formed below:



An important fact about the dual port RAM used to represent where blocks are on screen is that there is only one read and one write port. Reading for the purpose of displaying blocks on screen must be separate from reading for the purpose of collision detection. For this reason, the read address into the block RAM, **raddr**, is specified depending on if the **display\_on** signal is asserted, meaning pixels are being drawn on screen, or deasserted, meaning the VGA module is not drawing on screen.

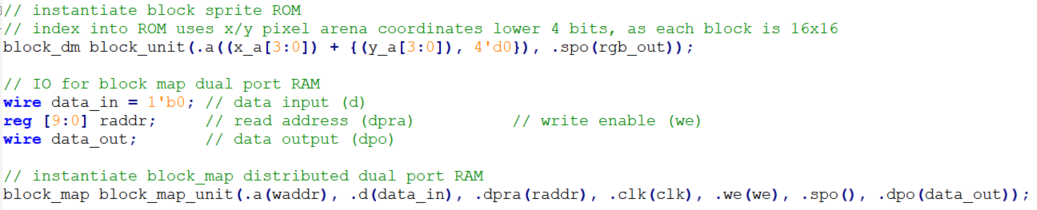
When the **display\_on** signal is asserted and the VGA **x**/**y** pixel coordinates are within the arena bounds, the read address is formed by using the **x**/**y** coordinates converted into ABM coordinates to index to the correct block in the RAM. When the **display\_on** signal is deasserted, the read address is formed depending on the current direction, using the signals defined above, converted to ABM coordinates.



## Block Sprite ROM & block\_map RAM

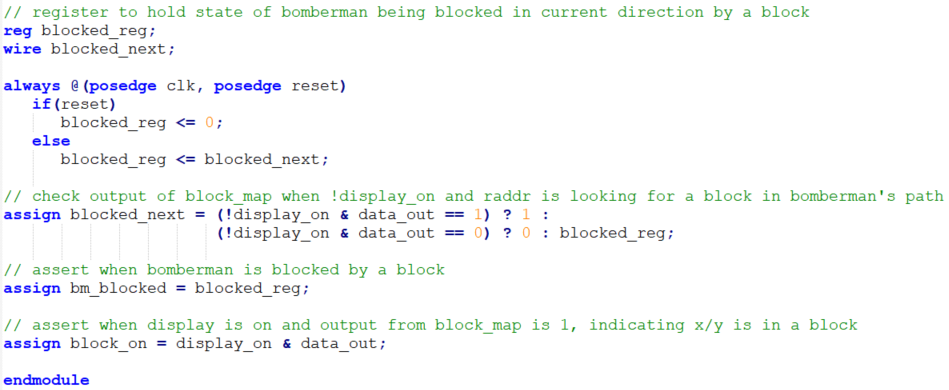
The block sprite distributed memory **block\_dm** is instantiated and indexed using the VGA pixel **x\_a**/**y\_a** arena coordinates lower 4 bits, as they repeat every 16 pixels for each 16x16 block.

The **block\_map** dual port RAM is instantiated, with data\_in tied to 0 in order to clear blocks when the **waddr** write address is set and the **we** write enable is asserted by the **bomb\_module**’s explosion tiles, as explained later.



## bm\_blocked

The **bm\_blocked** output is asserted when it is calculated using **x\_b**/**y\_b** and **cd** that a collision with Bomberman and a block is to occur. The **raddr** into the **block\_map** RAM is set using the cd and signals formed above such that the contents of the RAM is tested to be 1, indicated a block is in Bomberman’s path. The **blocked\_reg** registers this blocked status and is asserted when the display is off and the data from the RAM is 1.



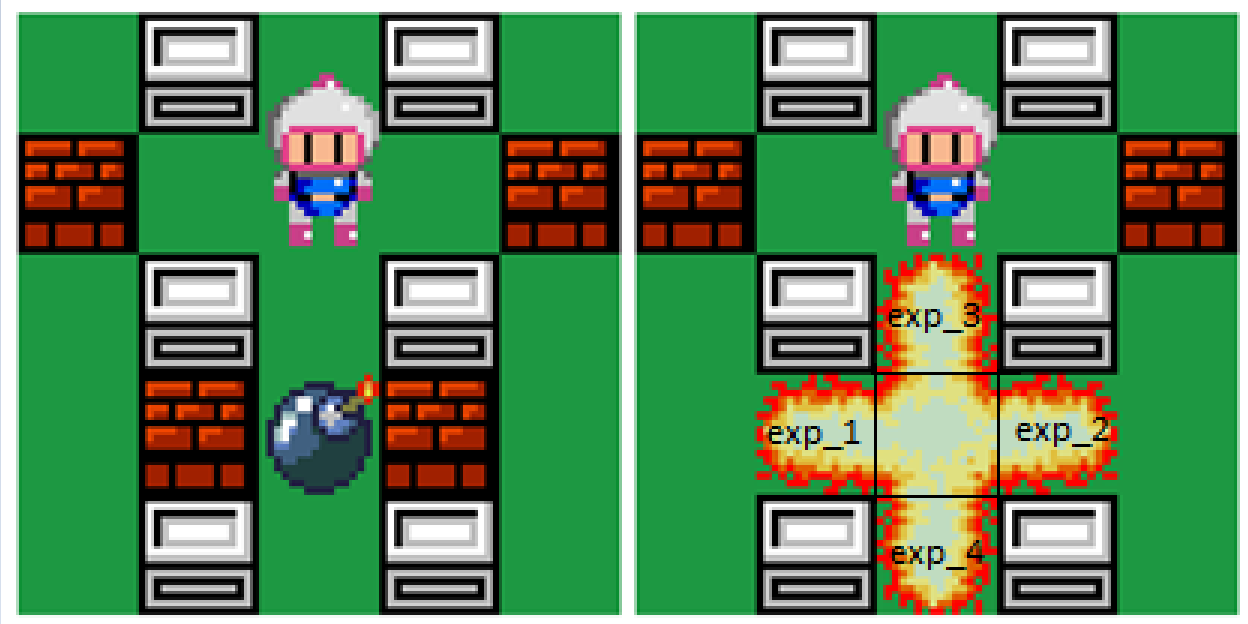
The **block\_on** output is asserted when the display is on, and the data\_out from the RAM indicates that the VGA **x**/**y** pixel coordinates are within a block on screen.

**With the block module up and working, Bomberman is stuck in the upper left-hand corner of the screen. Time to explore how he will blast through.**

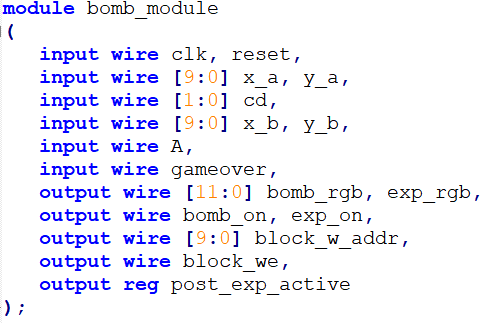
# Bomb Module

Add to the project the source file **bomb\_module.v** and uncomment its instantiation in the top module. Generate the **distributed memory** **ROM** for the **bomb.png** image, naming the component **bomb\_dm**. Generate the **block memory** **ROM** for the **explosions.png** image, naming the component **explosions\_br**.

The purpose of the bomb module is to drop a bomb to the ABM tile nearest to the center of Bomberman’s hitbox. This bomb remains on screen for a short period, and then explodes. The explosion encompasses the tile of the bomb, as well as the tiles to the left, right, top, and bottom. The explosion does not affect pillars or the walls and can destroy blocks. A carefully timed explosion will hit the enemy but can also hit Bomberman.



## IO



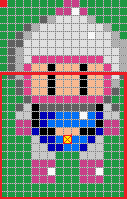
Input includes VGA pixel coordinates in arena coordinate frame **x\_a**/**y\_a**, Bomberman’s current direction **cd**, Bomberman’s location **x\_b**/**y\_b** , bomb input button signal **A**, and the **gameover** signal used to stop bombs from being placed when the game is over.

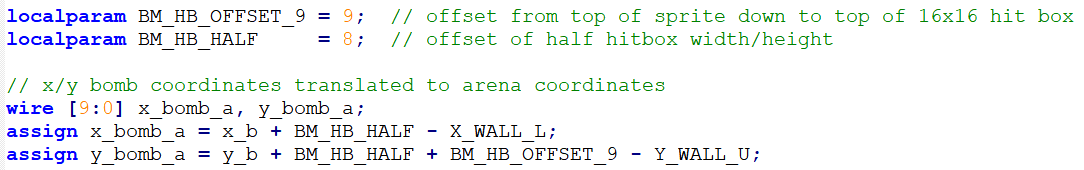
Outputs include RGB color data and on signals for the bomb and explosion tiles on screen, the write address and write enable to the **block\_map** RAM in the **block\_module**, and finally **post\_exp\_active** which is asserted when the explosion is active on screen and is used in the **enemy\_module**, as discussed later.

## Bomb Coordinates

The bomb coordinates **x\_bomb\_a**/**y\_bomb\_a** are derived from Bomberman’s coordinates, and roughly translates to the center of the hitbox.

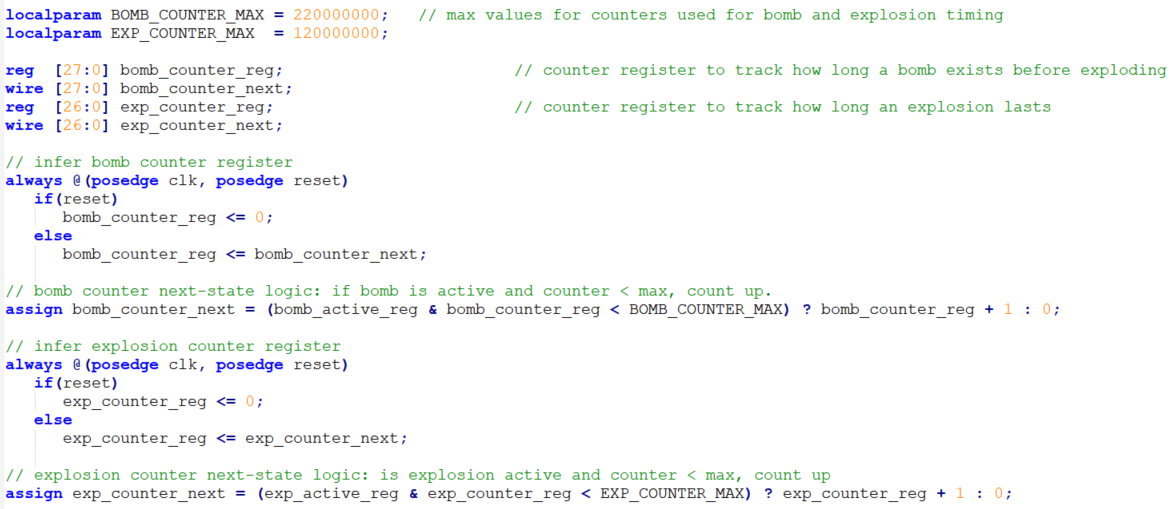
In the image below, Bomberman’s coordinates are defined by the upper left pixel of the frame (red), and the bomb coordinates are defined by the pixel marked with an x.





## Bomb & Explosion Counters

The **bomb\_counter\_reg** counts for a time period until it reaches the max value **BOMB\_COUNTER\_MAX**, upon which it will reset to 0. The counter is initiated by the **bomb\_active\_reg** signal which is part of the FSM described below and is asserted when the bomb is to be active on screen, during a state of the FSM.



The explosion counter works the same way, counting to a max value, and being initiated by a registered signal which is asserted when the explosion is to be active on screen, in a state of the FSM.

## 2.) Bomb and Explosion FSM

The state machine for the bomb and explosion sequence is shown below as an ASMD chart, where squares represent states, diamonds represent a condition to be tested, and an oval represents data that is changed in response to a condition.

✎ **Implement the state machine described below. Use the registers and state values defined in the bomb\_module.**

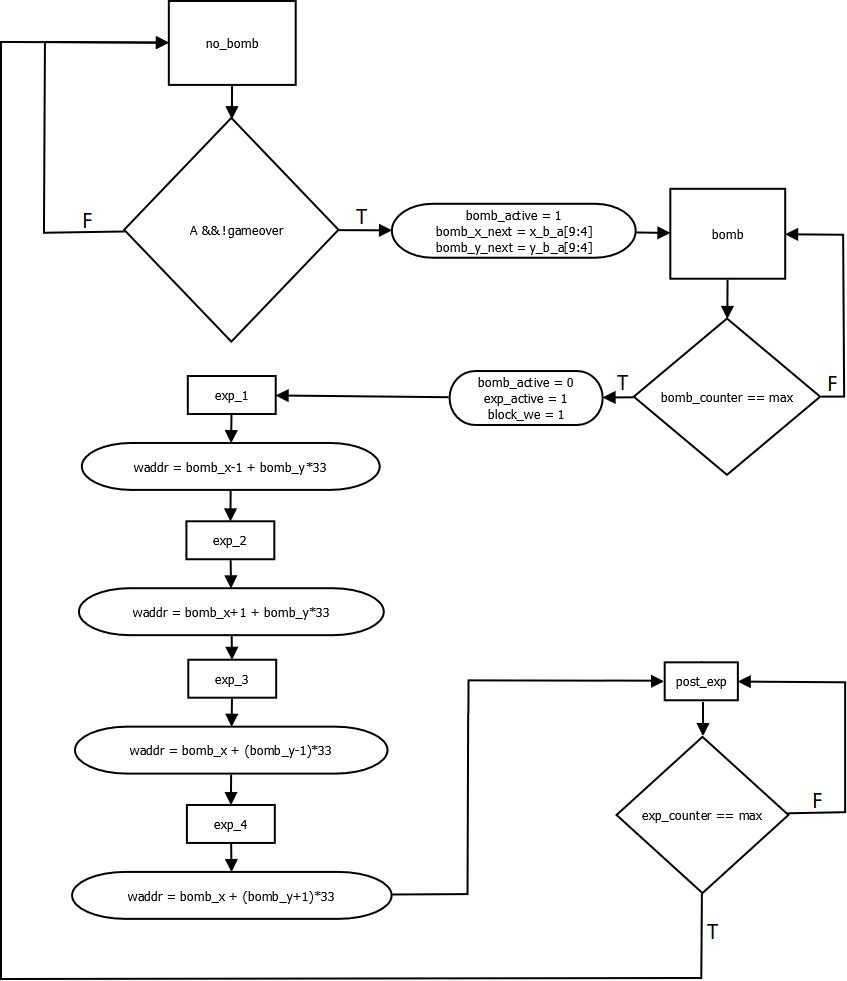
Before the user presses the bomb button **A**, there is no bomb or explosion on screen, and the FSM state is **no\_bomb**. If the user presses **A** and the game is not over, assert the **bomb\_active\_reg** which allows the bomb counter to run, set the bomb coordinates register to the current location of the middle of Bomberman’s hitbox, **x\_bomb\_a** / **y\_bomb\_a**, and go to the **bomb** state.

In the **bomb** state, wait for the **bomb\_counter\_reg** to reach the max value (keeping the bomb on screen for a period), then deassert the  **bomb\_active\_reg**, assert the **exp\_active\_reg** which allows the explosion counter to run, assert the **block\_we\_reg** to allow writing into the **block\_map** RAM for explosions to clear blocks, and go to the **exp\_1** state.

In the **exp\_1** state, the ABM coordinate to the left of the bomb is routed to the write address of the **block\_map** RAM so it will set that location to zero, clearing a block if it were there. This action only happens if the x coordinate of the bomb is not zero, thus making sure not to index into the RAM wrongly where there is a wall. The state machine then goes to the **exp\_2** state.

The **exp\_2**, **exp\_3**, **exp\_4** states are similar in that they set the write address to the **block\_map** RAM for the ABM coordinates to the right, top, and bottom of the bomb, clearing those coordinates in the RAM as well. Each state goes directly to the next, until **exp\_4** goes to the **post\_exp** state.

In the **post\_exp** state, the **post\_exp\_active** output is asserted, which is used in the **enemy\_module**, as discussed later. The state then allows the explosion counter to count up the max value (keeping the explosion on screen for a period), then set the **exp\_active\_reg**  and **block\_we\_reg** to 0. The FSM then goes back to the **no\_bomb** state.



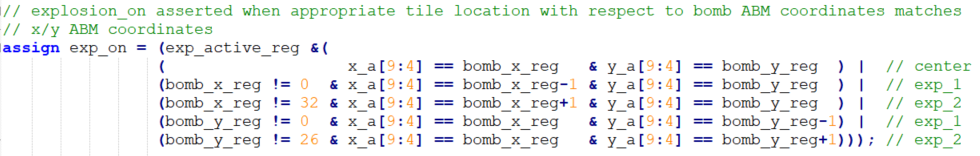
One aspect that is left out of this diagram that is in the final code is to test for the edges of walls when indexing into waddr, i.e. if using bomb\_x\_reg-1, make sure bomb\_x\_reg != 0.

## ROM indexing and Outputs

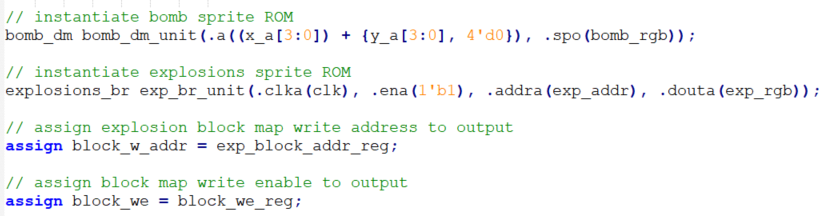
The output **bomb\_on** is asserted when the VGA pixel in ABM coordinates (**x\_a[9:4], y\_a[9:4]**) are equal to the bomb ABM coordinates, and the bomb\_active\_reg is asserted, meaning the bomb is active on screen. This signal is used in the **top\_module** to route RGB output from the **bomb\_module** to the VGA driver when the current VGA pixel is within where a bomb should be on screen.



The assignment for **exp\_on** is similar but is asserted for when the VGA pixel is in the center of the explosion, as well as 4 surrounding tiles.



The ROMs are instantiated and indexed into, and the **block\_w\_addr** and **block\_we** are assigned to the registers controlled in the FSM data path.



# Enemy Module

The enemy module creates a single enemy dragon on screen that moves around the arena. The dragon will randomly change its direction and can fly over blocks contained within the arena. When the enemy’s sprite overlaps an explosion tile on screen, it will be hit. When hit it will stop, look distressed, and then speed up.

This module follows a form that is nearly identical to the **bomberman\_module** for sprite ROM indexing.

Add to the project the source file **enemy\_module.v** and uncomment its instantiation in the top module. Generate the **block memory** **ROM** for the **enemy\_sprites.png** image, naming the component **enemy\_sprite\_br**.

## IO

The module has inputs for VGA pixel coordinates, Bomberman’s coordinates, and bomb ABM coordinates. Inputs also include **exp\_on** which is asserted when the VGA pixel is within an explosion tile on screen and is used to determine if an explosion is on at the same time as the enemy, denoting a collision or enemy hit. The **post\_exp\_active** signal controlled from the **bomb\_module** FSM is used to hold the enemy still when it is hit by an explosion for the entire duration of the explosion. The output **enemy\_hit** is asserted when the enemy is hit by an explosion and is used in the **game\_lives** module to increment a score counter.

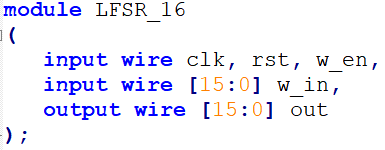
## 3.) Linear Feedback Shift Register

The enemy will need a pseudorandom register to achieve a random walk across the arena by taking random direction changes with a random probability.

✎ **Design a Maximal Length 16-bit Linear Feedback Register module with the following specifications:**

* clk / rst inputs
* write enable input
* 16-bit write data input that sets the shift register to a new starting value.
* 16-bit output
* Maximal length output sequence of 216-1 unique values.

Use the following module format:

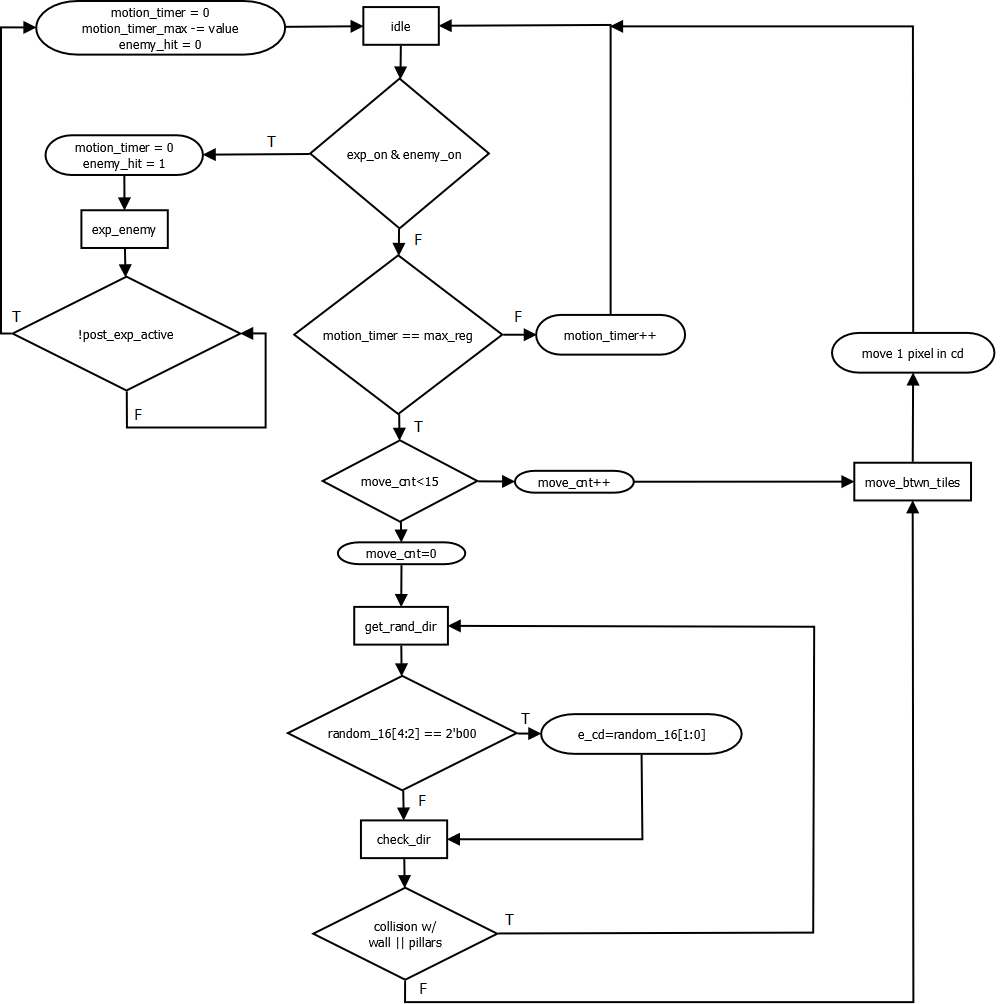


## 4.) Enemy Finite State Machine

✎ **Implement the state machine described below. Use the registers and state values defined in the enemy\_module.**

The enemy begins centered in an ABM tile with a current direction stored in **e\_cd\_reg**. The enemy moves 16 pixels in the current direction, as tracked by **move\_cnt\_reg**. The enemy is then at a new ABM tile. With a 1/8 probability, determined by the 3 bits in **random[4:2]** being 000, the **e\_cd\_reg** is set equal to the lower 2 bits **random[1:0]** and the enemy gets a new direction. It is then determined if this new direction is legal, meaning it will not allow the enemy to pass through the arena walls, or pass through a pillar. If it is illegal, a new random value is chosen until it is legal.

The ASMD for the FSM for the enemy module is shown below:



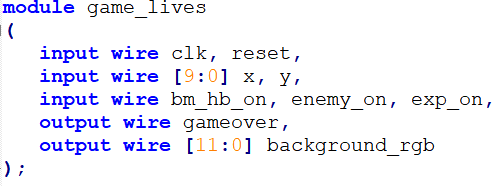
When the signal **enemy\_on** (asserted when VGA pixel is within enemy sprite on screen) and the input signal **exp\_on** are simultaneously asserted, it means that the sprites of the enemy and explosion overlap on screen and the enemy has been hit. The FSM state will go to the **exp\_enemy** state where it will remain until the **post\_exp\_active** signal from the **bomb\_module** is deasserted, meaning the explosion on screen is finished. During the time where the enemy remains hit, the final sprite of the enemy is shown, and the enemy does not move.

However, every time an enemy is hit, it should start moving faster and there should be higher probability of more random movement.

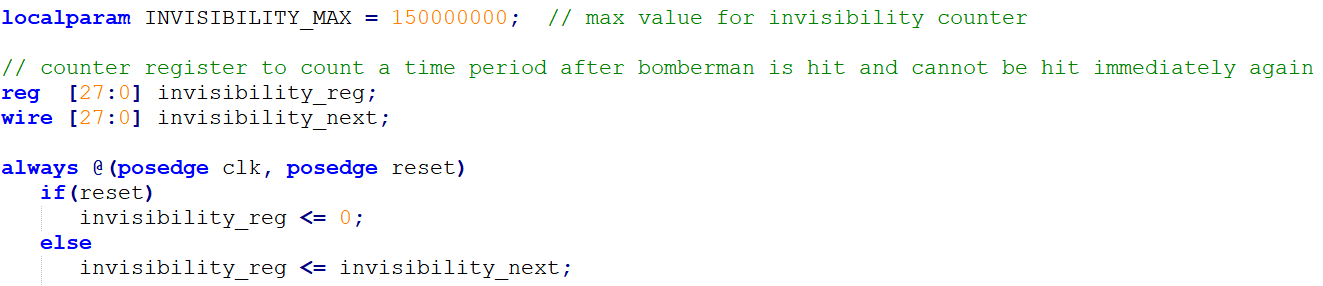
# Lives and Score: game\_lives Module

Add to the project the source file **game\_lives.v** and uncomment its instantiation in the top module.

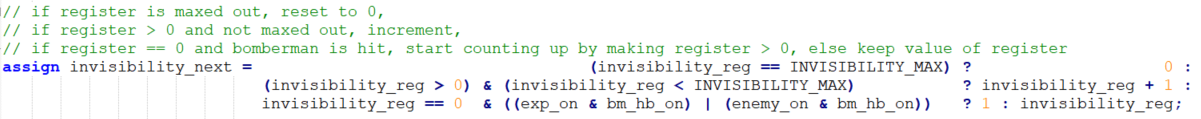
The **game\_lives** module maintains a “lives” counter, beginning at 5, and decrements each time the enemy collides with Bomberman’s hitbox on screen, or when Bomberman collides with an explosion tile on screen. The module outputs a background RGB value which is correlated to the lives counter, going from red to black, which gives an indication of lives left. The output **gameover** is asserted when the lives counter reaches zero and is used in the **bomberman\_module** to stop Bomberman from moving and dropping bombs when the game is over.



An invisibility counter begins counting when Bomberman is initially hit by an enemy or explosion and is used to prevent successive damage from being taken for a short period.



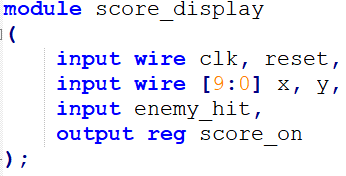
The next-state logic for the invisibility counter resets it to 0 if it reaches the max, increments the counter if the value is above 0 and below the max, and begins counting by setting the value to 1 when the explosion or enemy overlaps Bomberman’s hit box.



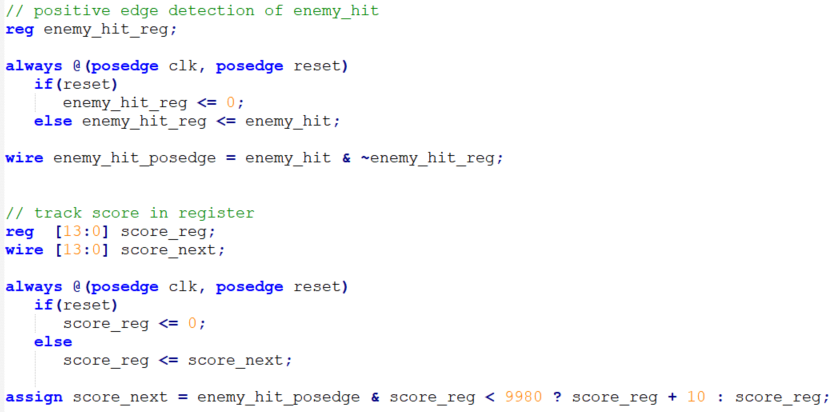
# Score Display & binary2bcd Conversion

Add to the project the source file **score\_display.v** and uncomment its instantiation in the top module. Add to the project the ROM template source file **numbers\_rom.v**.

The **score\_display** module keeps track of a score counter, incremented by 10 each time the enemy is hit by an explosion, counting from 0 to 9999. The score is displayed on screen using 4 digits. The score, stored as a binary value in **score\_reg**, is converted by the **binary2bcd** module to four binary-coded decimal values that are displayed on screen.



The **enemy\_hit** input is asserted by the **enemy\_module** when the enemy is hit by an explosion and its FSM is within the **post\_exp** state. A positive edge detection circuit is designed, and the positive edge of **enemy\_hit** is used to increment **score\_reg** by 10, if it remains below 9999.



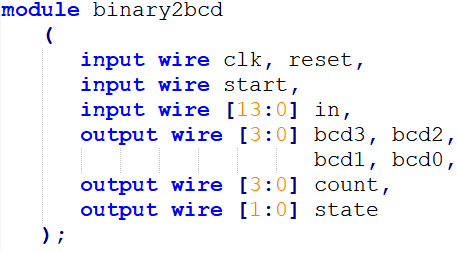
## 5.) Binary to BCD

✎ **Implement a module that converts a 14-bit binary number to four binary coded decimal numbers that represent the number.**

The **binary2bcd** module converts a binary value of **score\_reg** (0 to 9999) into a binary coded decimal format where four binary values represent the decimal numbers to be displayed on screen. For instance the binary number 100111000011112 (999910) would be represented as for four binary 9 values: 100110 100110 100110 100110.

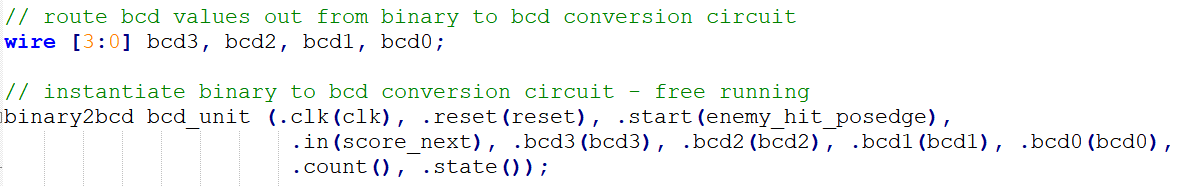
While a fully combinational implementation would suffice, your implementation should be **algorithmic** and **use a state machine**, should take in a 14-bit binary number, and output four BCD values that represent it. **A testbench along with correct waveforms and test values for inputs within the range of 0 to 9999, including the bounding values 0 and 9999, is required.**

The module should follow the following IO format:



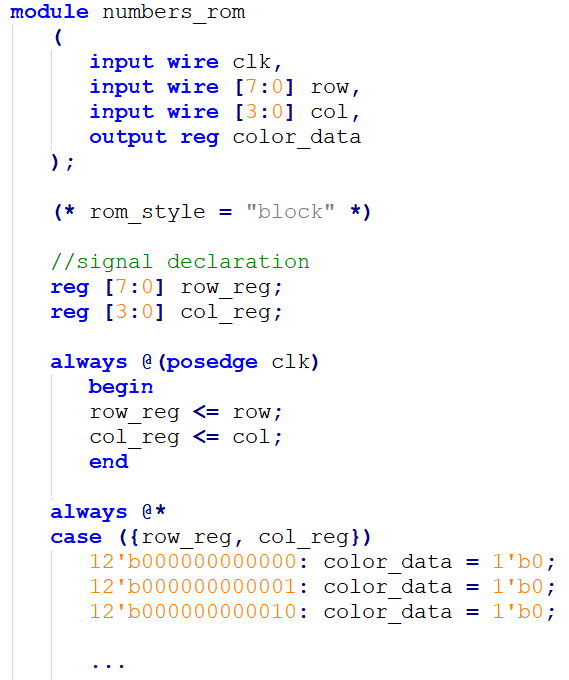
The module takes in a 14-bit value **in**, and a **start** signal to initialize the conversion. Since the conversion is used on numbers displayed on screen and occurs quickly when a score is changing, intermediate output values that are not valid are not an issue, so an output signal indicating the conversion has finished is not necessary. Both **count** and **state** are left unconnected in the instantiation of the module but are used in the testbench to display progress during the algorithmic conversion for testing purposes.

The **binary2bcd** module is instantiated in the **score\_display** module.

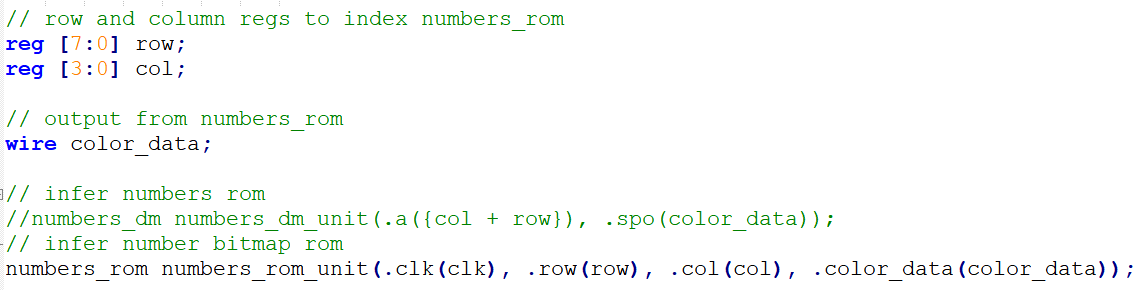


## Numbers ROM

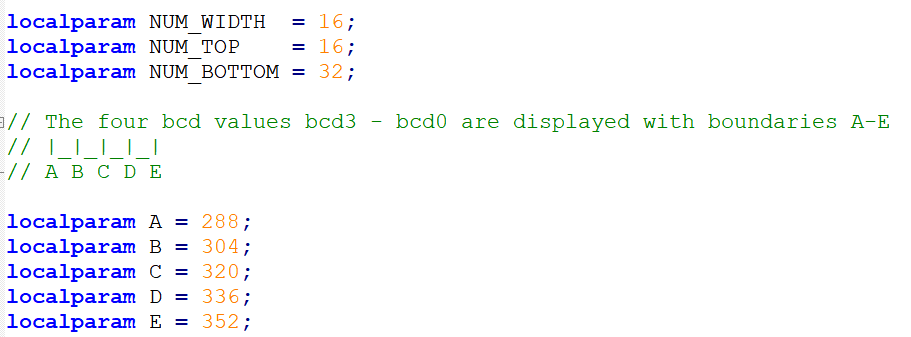
The numbers ROM is a ROM holding 1-bit values, where values coded 1 represent the font. The **numbers\_rom** is a ROM template that uses block RAM, and follows the format detailed in the Xilinx XST guide to inferring memories. The ROM is its own module and is instantiated the same way.

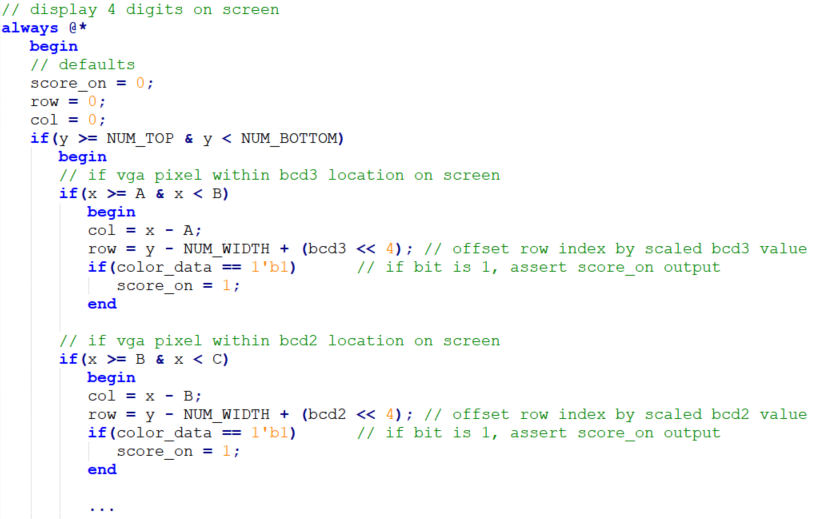
The **row** and **col** registers are used to index into the ROM by specifying the row and column of the pixel in the image, with the origin being in the upper left of the image.



The score is displayed roughly in the top center of the screen. The constants **NUM\_TOP** and **NUM\_BOTTOM** store the top and bottom y coordinates of the score on screen. The constants **A** – **E** represent the five boundaries of the four BCD digits on screen and are used to define the indexing into **numbers\_rom** depending on where the VGA pixel **x**/**y** is and the current BCD value.



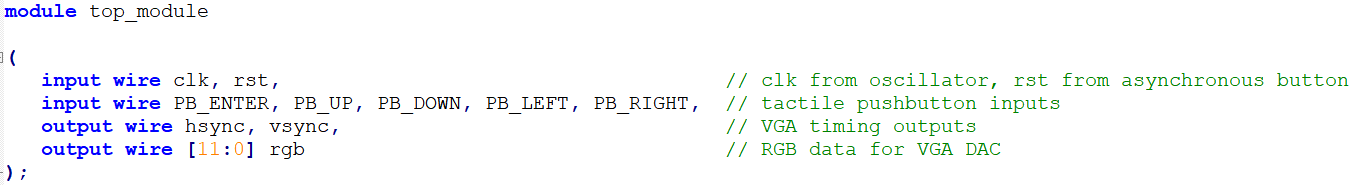
An always statement is used to set the **score\_on**, **row**, and **col** register values depending on what the current BCD registers are, and if the VGA pixel location is within the BCD value on screen. If **y** is within the score location on screen, and **x** is within **A** and **B**, i.e. is within **bcd3** of the score, the **col** register is formed as **x-A** and the **row** register is formed as (**y** – 16)\***bcd3\*16**, where **bcd3** is a value from 0 to 9 and when multiplied by 16 will index to the beginning row of the value in the **numbers\_rom**. If the **color\_data** output of the ROM is 1, then the **score\_on** signal is asserted, letting the top\_module know to draw pixel data on screen for the font of a number.



# Top Module Description

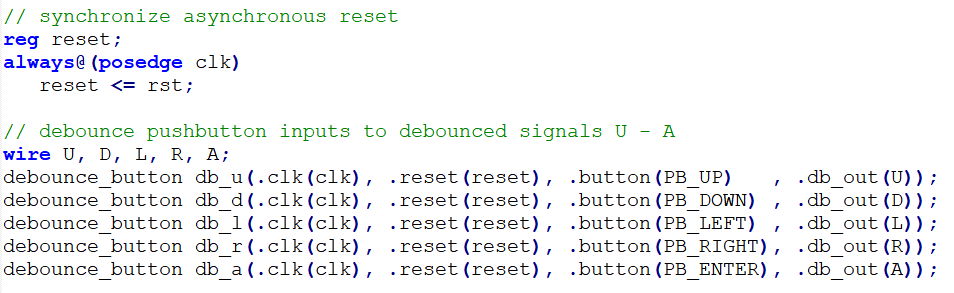
The top module handles IO to the FPGA peripherals, instantiates all the modules for the system and routes signals between them, passes color\_data from modules that generate it to the VGA driver when appropriate, and other small tasks that wrap the entire game up and make it functional.

IO are shown below with comments to indicate purpose:



The asynchronous reset button is synchronized to avoid metastability issues.

Inputs from 5 tactile pushbuttons are debounced using the provided **debounce\_button** module to prevent glitches during gameplay that allow input to break rules of the game.

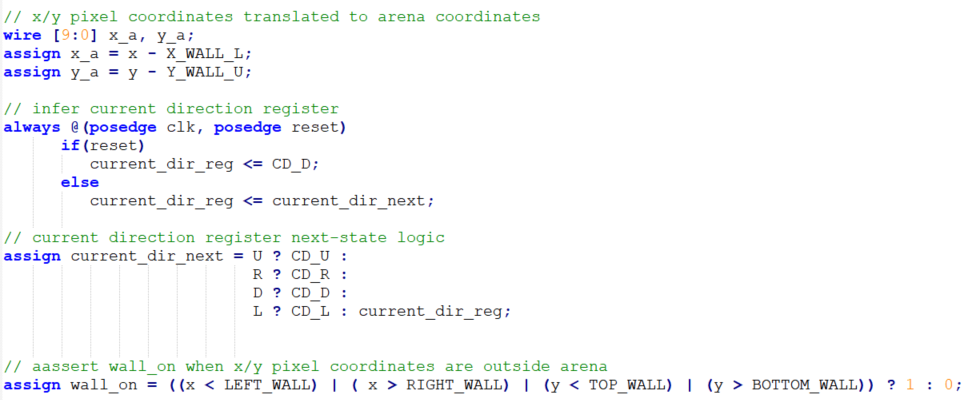


Up next are constants and various wires and regs used between module instantiations and in the rest of top.

The **x\_a**/**y\_a** VGA pixel coordinates in arena coordinate frame are generated and passed to modules that utilize them.

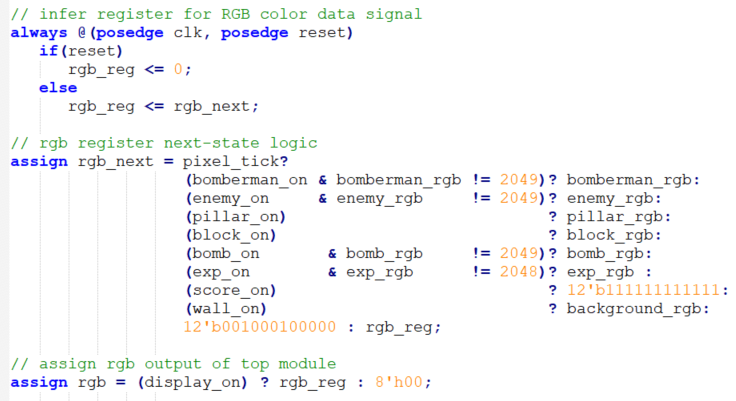
The current direction is a sampling of the directional input buttons and keeps a registered state of the direction of Bomberman.

The **wall\_on** signal is asserted when the VGA pixel coordinates are outside of the arena.



Next up are module instantiations.

The **rgb\_reg** contains the RGB data to route to the RGB DAC. The next-state logic uses the **pixel\_tick** pixel clock signal from the VGA module to determine when to update the RGB output data for a new pixel location. A priority routing network uses a module’s asserted “on” signal to route out its RGB data. The priority is given to RGB data to display RGB data in the correct way, i.e. Bomberman walking over a bomb, rather than under it. RGB data for Bomberman, enemy, bomb, and exp are tested to not be the arbitrary background color for the respective image before routing the RGB data out. If the **score\_on** signal is asserted, meaning the font of the score is to be displayed, the RGB output is set to all on to display white numbers. If no “on” signals or **wall\_on** are asserted, then the background green color is displayed. RGB data in **rgb\_reg** is only output to the RGB DAC output **rgb** when the **display\_on** signal is asserted.



## 6. )Bomberman Life Bar

The game\_lives module already takes care of calculating the lives of bomberman and displays a suitable background color. However, the experiment requires you to implement a 20 pixel \* 8 pixels (width and height respectively) red colored life bar for the bomberman. This bar should be displayed in the top right corner. Since the bomberman has 5 lives, a portion of the red bar should be filled with black color each time the bomberman receives a hit. Once all the 5 lives are lost, the entire life bar turns black. No coe file or pre-defined logic is provided for the red bar in the source code. You should write your own custom logic.

# 7. ) Your Creative Expansion

Now that you have completed the basic form of the game, it is time for you to choose your expansion of the game from the options below:

1. **Modify the game for a two-player mode**. Two Bomberman characters are playable on screen and are identifiable by their color by changing the white color of the original for the second player. You could have the game be a match of who can get the highest score in a certain time period, or a battle royale between the two players.
2. **Modify the game to have have a pause functionality**. One additional input can be added to pause the game. Pressing the pause button should freeze/unfreeze the movement of both the bomberman and the enemy.
3. **Modify the game to have a soundtrack and sound effects.** Use the audio output from the FPGA development board. Add a title screen to the game with title screen music. Add music to be played during game play. Add sound effects for a bomb being placed, bomb exploding, Bomberman being hit, and an enemy being hit.

# Grading Rubric

1. Animate Bomberman 20%
2. Bomb and Explosion FSM 25%
3. Linear Feedback Shift Register 5%
4. Enemy Finite State Machine 25%
5. Binary to BCD 10%
6. Animate Life Bar 15%
7. Your Creative Expansion 20%

Total 100% + 20%