Bomberman

On the Xilinx Virtex II Pro

Team Bombsquad

Mario Escalante
Neha Padhi
Jong Paek
Henry Teng

What we built

Bomberman is a classic Nintendo game, of which there are many versions. The object of the game is to be the last one standing among the multiple players (usually 4). Each bomberman character can drop exploding bombs, which can eliminate a player from the game if they are hit by its blast radius. These explosions can also be used to clear destructible walls from a player's path. In addition to this basic game play, Bomberman also incorporates power-ups. The version of Bomberman which we have modeled our game from, Super Bomberman 2, includes the following power-ups: the ability to kick bombs, the ability to throw bombs, fire power (allowing the bomberman's bombs to have a larger blast radius), the ability to drop extra bombs, and the ability to increase the bomberman's speed.

Our version of Bomberman is adversarial, allowing up to 4 players. The game lasts until there is only one bomberman left. We also implemented power-ups, choosing to use the following two: the ability to drop extra bombs and the fire power. When players break a destructible block with a bomb, randomly generated power-ups appear behind each of them.

Hardware/Software Partition

The game implementation runs *entirely* on the Xilinx Virtex-II Pro FPGA board, which contains two PowerPCs and an FPGA. The SDL graphics, I/O, and main game code run on the first PPC (ppc_0) and the FPGA, while the audio, including the sound mixing, runs in parallel with the other aspects of our game on the second PPC (ppc_1).

Hardware Description

The hardware modules we implemented for our project were the controller interface, a custom designed VGA framebuffer, a 32 pixel blitter, a timer module to handle bomb explosions, random number generator for power ups, the game state (what object is placed at each location in the map), and the blit image calculator.

The controllers are interfaced through Verilog modules tied into the On-chip peripheral bus (OPB). These modules are debounced using sample code from the EE108b course taught at Stanford University.

The VGA framebuffer we designed supports a single resolution (640x480), has software addressable registers for reading in the starting memory address, reads from continuous memory, and supports 24 bit color. There are 5 modules instantiated within our framebuffer for reading from memory and writing to the screen. These modules include a bus interface block, a BRAM interface, the HSYNC and VSYNC counters, and finally the TFT interface block. The VGA framebuffer module operates on two clocks, the plb clock running at 100 MHz, and the TFT clock running at 25 MHz. The bus interface block uses the plb clock to handle all of the bus protocol and initiates 4 double word burst transfers upon receiving the appropriate signals from

the HSYNC and VSYNC counters (when our HSYNC and VSYNC counters indicate that the screen is going into its active pixel state). The BRAM interface uses a dual ported BRAM (640 x 4 bytes) acting as a line buffer and both clock domains to synchronize reading and writing. The dual ports are used so that the plb bus data can be written to the bram on the plb clock, and pixel data can be read out on the tft clock. The tft read port has a width of 24 bits, and the plb write port has a width of 48 bits. Since the incoming bus data is 64 bits wide, our BRAM allows us to write 2 pixels off the bus in one plb clock, and read 1 pixel out on the tft clock. Each pixel laid out in memory takes up 32 bits, with the upper 8 bits zeroed out. This avoids using the extremely complicated non-aligned memory accesses the bus interface allows. The BRAM read data is passed to the TFT interface which ties in directly to the 8 bit R, G, B inputs of the DAC chip used to drive the screen. The VSYNC signal is an output of our framebuffer module so that it can be used to synchronize writes by other hardware writing to buffers.

The 32 pixel blitter works very similarly to the VGA framebuffer in that it reads in pixel data off the bus and into a BRAM. One of the main differences is that instead of the data in the BRAM being used to output to the DAC on the board, it is used to write back to memory. Each reading and writing bus transfer is a 12 double word burst. After each transfer, the hardware writes that line back into the destination pointer. It repeats this process until all of the data in a 32 x 32 block has been copied over. The address of the source pixels and the address of the destination pixels are stored in software addressable registers. If the VSYNC signal is asserted the module latches the addresses, and waits for it to go low before writing to the destination. We use 1 counter to synchronize the data transfers. The counter increments on each transfer, both reading and writing, and when it reaches 64, the finite state machine controlling our bus accesses asserts a ready signal so that the software knows it can overwrite the addresses currently in the registers to initiate a new transfer.

The timer module uses a number of cascaded counters to produce values in milliseconds. It contains a software addressable register that can be read from our game code to receive the current millisecond count since the board powered up. The millisecond count overflows after about 49 days.

We imported a random number generator into our game from source code we found at opencores.org (http://www.opencores.org/projects.cgi/web/systemc_rng/overview). This rng uses a Linear Feedback Shift Register (LFSR) and a Cellular Automata Shift Register (CASR). The random number is created by permuting and XORing 32 bits of LFSR and CASR. We used this module to determine what power-up would appear when a bomb exploded a destructible block on the board.

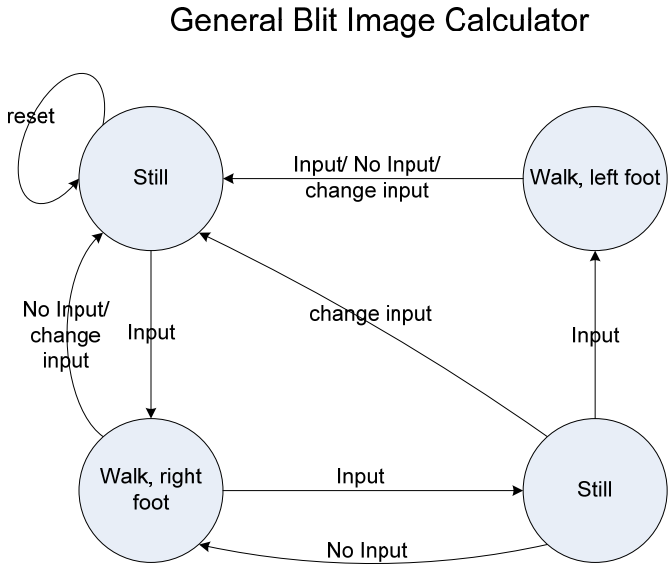
The game state in hardware is composed of several FSMs to aid in finding open positions as well as help the software check for collisions between bomberman and bombs, bricks, power-ups and walls.

The game state hardware takes an x,y position for reading and an additional parameter corresponding to a legal tile object (bombs, power-ups, destructible blocks, bomb explosion, etc) for writing. There is also a Read/Write signal to determine which operation to perform. The player is not physically contained in the map grid, since he can move around in between tiles and

be in multiple tiles at once. The map grid is stored in a BRAM generated by CoreGen to be an array of 255 tiles that corresponds to each tile in the game board. The BRAM is dual ported so that our state machine can read from two different addresses at the same time, or update multiple addresses at the same time. This is useful at the very beginning of a game because it allows us to initialize the game state twice as fast.

The map grid contains values for each tile object that are consistent the values given to them in software. The state machine within the module coordinates software driven requests and handshaking signals to the BRAM. The FSM allows the software to write any tile value to the board.

The other FSM that we implemented in hardware was the blit image calculator. This state machine would determine which image needed to be displayed given previous orientation and movement of the bomberman. This was needed to maintain the correct animation sequence. In any given direction, the bomberman begins at a rest state, walks with his right foot, becomes still again, and walks with his left foot. This motion is repeated until the player stops moving, when the bomberman returns to the rest state image. This FSM has four states that loop around whenever the new movement value is equivalent to the old value. If they are different or all of the movement buttons are deasserted, then the bomberman immediately goes back to the reset rest state.



This is the general implementation for the hardware blit image calculator. These sequences of movements happen regardless of which direction the bomberman is facing. If the bomberman ever changes direction in the middle of the sequence, he goes back to the reset rest state.

Software Description

We ported a custom built version of SDL to support the graphics for our game code. It is the main backbone for our software implementation of graphics and audio. We implemented the

following SDL methods for the graphics, colorkeys (setting transparencies), alpha blending (setting opacity), dirty rectangle tracking, loading any bitmap of arbitrary size and format, blitting color keyed, alpha, and normal surfaces, screen flipping, and clipping.

The software is the main driver of our game code, but the game state and the blit image calculation are stored in hardware.

The map dimensions are hard coded to 17 x 15 tiles with each tile being 32 x 32 pixels.
The map state is initialized in our game state hardware when the software asserts the reset signal, or when the global reset is asserted (this is asserted as soon as the Xilinx board boots up). The baseline map has an unbreakable wall on the outside edges of the map, as well as every (i,j)th tile where i and j are odd. This is already built in to the map background and cannot be changed. Ideally, we planned to place the bricks randomly tile-by-tile with about 90% hit rate (since most of the map should be covered by bricks), but since it was relatively unimportant (and it increased the complexity by a great deal), we decided to just read in from a file that has 15x13 tile information.

MapGrid:
There are several classes that keep track of the game state. The mapgrid.c file updates the map state on each loop, by writing new tile values to the hardware BRAM. These are all the tiles defined both in hardware and software.

```
#define TILE_EMPTY 0
#define TILE_BOMB 1
#define TILE_WALL 2
#define TILE_BRICK 3
#define TILE_BOMB_DET 4
#define TILE_BRICK_DET 5
#define TILE_BLAST_RIGHT 6
#define TILE_BLAST_RIGHT_END 7
#define TILE_BLAST_LEFT 8
#define TILE_BLAST_LEFT_END 9
#define TILE_BLAST_UP 10
#define TILE_BLAST_UP_END 11
#define TILE_BLAST_DOWN 12
#define TILE_BLAST_DOWN_END 13
#define TILE_EMPTY_NEXT 14
#define TILE_POWER_BR 15
#define TILE_POWER_BM 16
```

The game state is updated constantly and is the basis for game mechanics and screen updates. Mapgrid also keeps a list of all the bombs that have been planted, but have not yet exploded (list_t bomblist;). <list_t> is a customized implementation of a queue (found in queue.c). <list_t> has as its members a <Bomb> struct, which keeps track of the x-y coordinate of the bomb's location, as well as the time it was planted.

The players' status (dead or alive) and their x-y coordinate are also tracked in MapGrid. Finally, SDL_Rect values for every tile in the map are initialized in MapGrid (SDL_Rect grids[MAP_WIDTH][MAP_HEIGHT];). SDL_Rect is a struct defined by SDL, and is supposed to represent a rectangular area of pixels. SDL_Rect contains x and y values (pixel offsets for the top-left corner) and the width and height of the rectangle. The appropriate SDL_Rect values are calculated for each tile when the game is initialized and is kept until the program quits. This is so that we don't have to recalculate pixel coordinates for each tile every time we need to reference to it (mostly for collision detection and updating the screen).

Player:

Information for each player is stored in <Player> struct. Basic information such as the x-y location (pixel offsets, not map coordinates), player number, number of bomb and fire power ups, and score are stored. Animation information, such as the animation frame the player last displayed, are stored within Player. Since animation is just a sequence of clipped sprites, we need to keep track of which frame of which direction the player is currently at. Every time the player moves, the movement direction is fed into the hardware blit image calculator, which determines the next image to blit. xVel and yVel, which represent x and y velocity, are also stored. The concept of velocity had to be introduced to allow smooth movement of the players. This is further explained in the next section.

Player movement:

Player movement happens in the following sequence: handle input -> move -> show on screen.
<player_handle_input> function handles the input and sets the corresponding velocity and plants bombs. The concept of velocity had to be introduced to allow smooth movement of players. When the user holds down a directional button, the bomberman needs to move across at a constant speed. If the movement is invoked only on the button-pressing event, the user would have to press the directional button repeatedly in order to move. So, I needed to write the code so that: whenever a directional button is pressed, corresponding velocity is increased; and the same amount of velocity is decreased when the button is released.
<player_move> function looks at the velocities of each player and invokes the appropriate function: mg_move_player_right, mg_move_player_left, mg_move_player_up, mg_move_player_left. These functions determine where the bomberman's next location should be, given its x and y velocity and its current location. This of course requires collision detection, which is discussed in the next section.
<player_show> function takes the location of each player and displays it on the screen. This function shows its current frame of animation (stored in Player struct), and increments the frame. This function also determines which direction the player is facing so the correct set of sprites can be shown.

Collision detection:

Since this game is entirely based on tiles, the only collision detection necessary is for player movement.
int collision_detect_80(SDL_Rect* object1, SDL_Rect* object2)
This function takes two rectangles and returns 1 if there is collision. The code was adapted from a snippet of code found on GameDev (http://gamedev.net). This function in particular, performs

collision detection on rectangles that are 80% of its original size. This is to avoid over-detecting collision. In a situation like this:

there shouldn't be any collision detected. In fact, it is acceptable for players to overlap with other tiles, as it gives an illusion of a 3-dimensional space.
Collision detection is performed inside <mg_move_player_...> function. Initially, the move function determined which tile the player was currently in, and collision detected against the 3 tiles in the direction the player was moving in. For example, if the player was in tile (3,4) and moving right, collision detection was performed on tiles (4,3),(4,4),(4,5). (NOTE: In SDL, pixel locations start (0,0) at top-left corner, and increases as you go right/down. (w,h) notation used).
This method was changed later to implement the following. Before the change, when the player was in a position like this:

when the user pressed down, the bomberman stopped moving. While this is accurate, it was easy for a novice player to get "stuck" by trying to change direction before clearing the obstruction. In the actual SNES game, in the above situation, the bomberman automatically slides to the left until it can safely move down. This part of the code was hacked in towards the end of the project, so the code is quite ugly, but it works.
When all the work above is done, the new location of the player is calculated. This information is updated to Player.

Processing bombs:
Every time a player presses the B button, a new bomb is created and planted on the tile player is standing on. The bomb can only be planted perfectly on a tile, so in this case:

the bomb will be placed on the middle tile even though he is occupying both middle and left tiles. The current time is stored within the Bomb struct. The new bomb is then added to the list of bombs in MapGrid.
Within the main game loop, which is looping constantly, <mg_process_bombs> function is called in the beginning. This function checks the head of the bomb list to see if a certain amount of time has passed since it has been planted. Since bomb list is in a queue structure, the head of the list will always represent the oldest bomb that was planted. If the time passed exceeds 3 seconds, <mg_explode_bomb> is called.
<mg_explode_bomb> simulates the actual explosion. This function checks all 4 directions of the bomb, checking the nearest tiles first. If the blast encounters a brick or a wall, the explosion can no longer affect a player/tile beyond that point in that direction. The presence of a player, on the

other hand, does not affect the blast radius of an explosion. So, if a bomb explodes in (0,0), and there is a player in (1,0) and a brick in (2,0), the player will die and the brick will still be destroyed. When the brick is destroyed, the software reads in a value from a random number generator to see which power up to place in that spot. Once the result of each explosion has been determined, the correct sprites must be displayed. Each direction of blasts has different sprites that need to be loaded. The bricks that are destroyed must be replaced with the "brick_broken" sprite. The correct tile values are set in the hardware game state to indicate these changes.

Updating the screen:

SDL draws on screen by using SDL_Surfaces, which are graphical surface structures. Whenever a sprite needs to be displayed, it is blitted onto the screen which is represented as an SDL_Surface (using SDL_BlitSurface). For the surface to be actually seen on the monitor, the buffers must be swapped. Initially, I used SDL_Flip at the end of main game loop, which swaps the entire screen buffer. After we realized that swapping buffers takes too long on Xilinx, we decided to only swap parts of the screen that gets changed, instead of swapping after every loop. We decided to use SDL_UpdateRects, which takes in an array of SDL_Rect's that represent areas of the screen that needs to be flipped. So every time we blit anything onto the screen, global variables <int numrects> and <SDL_Rect *updaterects> are updated and SDL_UpdateRects is called at the end of the loop.

Sound Implementation

We implemented the sound for the game in software using the SDL_LoadWav function and the on-board AC'97 codec. This function imports a wav file into memory so that we can write this data to the FIFO buffers on the opb AC'97 codec module.

To implement sound mixing, both PowerPCs were used to allow one of the processors to purely mix sound and play the resulting bitstream while the other PPC handled all the other aspects of the game. PPC0 was responsible for loading all of the sounds, while PPC1 took prompts from PPC0 to mix in the sound files.

A static array in memory is read by PPC1 to see if there are any non-null pointers that are waiting to be mixed and sent to the opb_ac97 codec module. Simultaneously, PPC0 places pointers in the static array corresponding to wav files that need to be played. When PPC1 is finished with a sound clip it nullifies the pointer, thus allowing PPC0 to copy a new pointer to that index in the static array. The array has a size of 16, thus allowing us to mix 16 sound samples at once.

In PPC1, the main function is a continuous loop that constantly iterates through the static array and mixes bomb explosion sounds with the background theme song. For each loop, it checks a set of indexes to see if they are not null. If there are non-null sound clips, it mixes them and the main audio track together by adding them, and sends the combined value off to the FIFO buffers.

How we Built our Project

Initially, we decided to partition the design by the four main components: graphics, sound, I/O, and coding the game, with each person working on 1 of the components. We found that this was not the best way to partition it after submitting our initial proposal since coding the game was dependent upon completing the software portion of graphics (i.e. porting SDL). Until the graphics was complete, we were to work in pairs to (1) complete porting SDL and (2) setting up the I/O with one SNES controller.

Once we had partitioned the design, we felt it was best to begin on the software implementations since this would allow us to fix many details that would otherwise be obscured by hardware problems. The only exception to this approach was the I/O. We needed to connect the controllers and test them so that we could test the game code (namely, player movements) as soon as that was available. We began with software implementations for everything else: for graphics, we ported SDL, stripping it to a bare minimum; we used SDL for our audio and sound mixing, and the game code was initially written completely in C.

We opted during the first design review to use the working Sega Genesis controllers since its parallel read is much easier to work with than the serial protocol on other controllers. Interfacing the genesis controller was much less complicated than the proposed SNES controller because we didn’t have to worry about producing a suitable clock frequency for reading in the serial bits corresponding to each button. We later found that the Digital Clock Managers on the board were very difficult to work with and it was a much better use of our time getting them to work on our VGA framebuffer than to have to figure them out for the controllers.

Once we had ported SDL, we wanted to integrate hardware to improve the graphics performance which at that point was a bottleneck. We could find very little information on blitting algorithms from the books in class or from the internet, and we had to reverse engineer SDL's blitting functions to figure out how to implement them in hardware. We wanted to hard code as few portions of our project as possible to maintain compliance with SDL's hardware accelerated functions. For this reason, we decided to design blitters that had no restrictions in terms of the size of the images being copied. The following figure illustrates our architecture for the blitters.

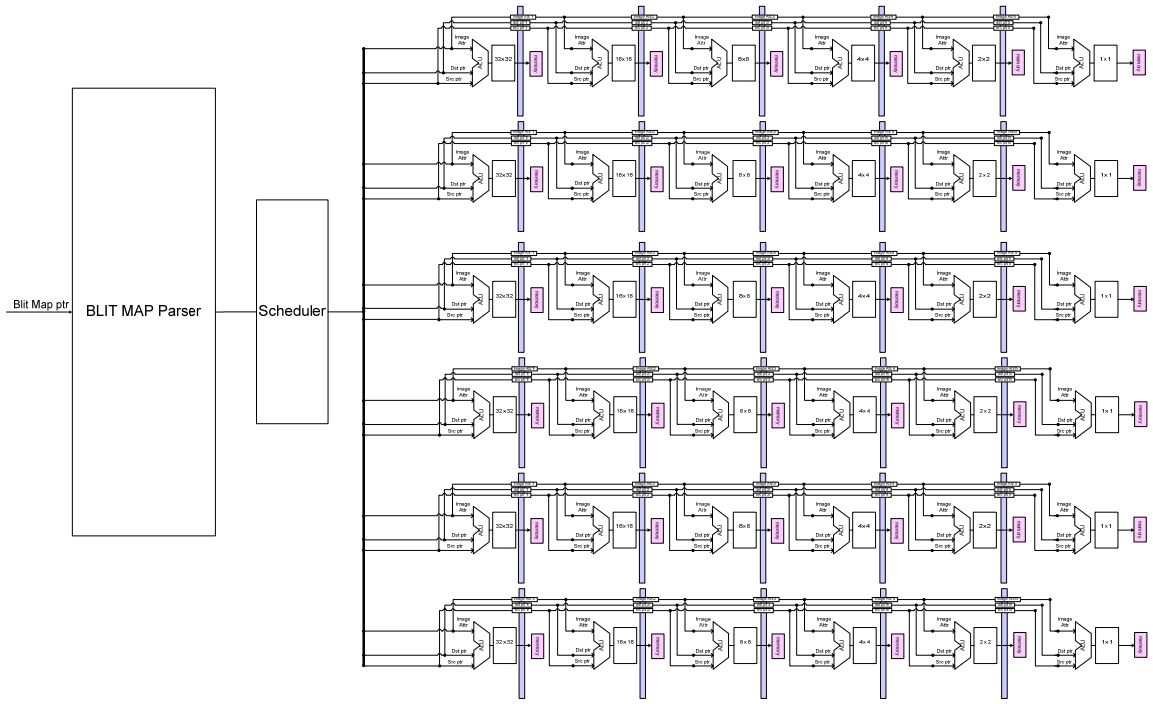


Figure 1: Baseline Configuration of the blitting units.

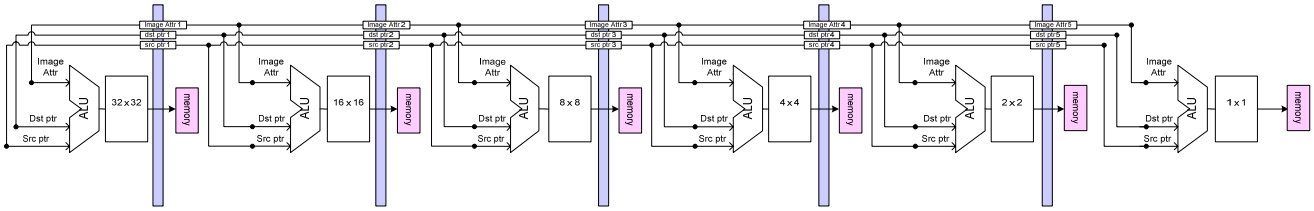


Figure 1: Blitting unit pipeline: Each image is passed through a series of blitters of different sizes. The sizes include 32 x 32, 16 x 16, 8 x 8, 4 x 4, 2 x 2, and 1 x 1.

Once we tried to synthesize our first blitter design, however, we realized we had no bram remaining in our project. We tried other options like using logic elements to hold our buffers, but it consumed too many elements (> 20 %). This meant we would have to break compliance with SDL. In fact, the only blitter we managed to fit was a 32 pixel line blitter.

Decoding the compression scheme used on alpha blending and the run length encoded transparencies would require more buffers, and we literally had 0 bram's remaining at this point, forcing us to abandon them. Removing these from the equation meant that our blitter would only have one use, which was to blit background tiles (which contained neither alpha nor transparency).

We implemented our own VGA framebuffer in our attempts to design a massively scalable blitting system. The primary reason we decided to make our own framebuffer was so that we could use the internal signals of the module (VSYNC) to synchronize any additional hardware that would write to the screen or another buffer. Unfortunately, by the time we tried to port everything onto the board, we had almost all of our BRAM in use already. In our final

implementation, due to the constraints of the hardware available, the most we were able to fit on the board with the remaining BRAM was a 32 pixel line blitter and the game state in hardware.

The VGA framebuffer was extremely hard to get working. It took us a week to figure out how to use the Xilinx plb IPIF to implement our first design. The first problem we encountered was with using the digital clock manager to produce a 25 MHz clock from the 100 MHz plb clock. It took us several days of experimentation to get this working, and when we finally did have it working, we had to introduce numerous signals to keep the plb and the tft portions of our design synchronized. When the DCM problems had been taken care of, we realized that, despite our best efforts to understand the bus interface, it did not respond at all the way we wanted it to. When we first tried to debug our hardware, we tried to pass out signals from our modules into the top level design so that we could view signals through software. However, this can only be done if you are familiar with the VHDL Xilinx IPIF wrapper that encapsulates your Verilog design. We had never had any exposure to VHDL before, but we were forced to learn exactly how the VHDL and Verilog were attached to be able to debug anything properly. After getting past this problem, we realized that the software could not always read the software addressable registers if there was a problem with the addressing within the module. For example, the IPIF has a signal IP2IP_Addr that holds the address of where you want your data placed when you assert a read request on the bus. The Xilinx documentation tells you that you can set it to an address within your module, but what it doesn't tell you is that this address is an absolute address, not a relative one. This meant that a user literally had to hard code the address of your registers in memory for a single configuration, and if you ever changed that configuration your module would instantly break. Once we got past this error, our bus interface was not working properly because our read request signals had to be de-asserted for a certain number of cycles before the bus would respond properly. This was impossible to debug using the software tools available to us, and we were forced to Chipscope the signals to figure this out. Like all Xilinx tools, there's a very steep learning curve to Chipscope, and we had to master this tool before it was of any use to us. Finally, using Chipscope we debugged the bus interface, and had to fix a few minor bugs in the BRAM interface to get our framebuffer to work properly. Fortunately, the line blitter was almost identical to the framebuffer in operation, and we didn't have to spend as long to debug it. What made this process so frustrating was that to make a simple change and see the results of what we changed took 20 minutes because we had to wait for our design to re-synthesize. This meant debugging the hardware was a slow painful process that was the most frustrating part of the course. It took us about 3 to 4 weeks to get our framebuffer working properly.

By mid-November, two of our group members were working on the VGA framebuffer and the blitters. Our other group member was working on sound mixing. We knew that the framebuffer, the blitter(s), and the sound mixing would be completed in time to be integrated into the game, but we had no idea where the game code development stood. One of our team members (the game coder) had the flu and we did not know his current progress nor the direction in which the game code was going, especially since it was difficult to contact him. At this point, the rest of us began to rethink how to implement the game code, and began creating different portions of the game in hardware. This included determining the player movement and collision detection in hardware and the blit image calculator. When we were finally able to contact our game coder, we found that much of the game had been coded and the hardware modules that determined the player movement and collision detection were largely incompatible with the game code that he

presented to us. To compromise, we stripped a large amount of functionality out of those modules so that we could integrate them into the game code. For instance, our game state calculator could move players along the board based on inputs from the Verilog controller module.

Our original plan was to look for a suitable game code online and make little changes to fit in our design. However, after a thorough and long search, we couldn't find anything that would have been any use to us. Almost all of the bomberman clones we found had nice screenshots, but were either incomplete or coded in a different language/library. Since the game seemed pretty simple (at the time!), we decided to code the game from scratch. This is a decision we very much regret in hindsight. If we had known how much time this would actually take we definitely would have tried harder to use some of the code online, or maybe even try to design a different game that has more resources available. We decided to code the game in C, using the SDL (Simple DirectMedia Layer) library (http://www.libsdl.org). The first thing we did was (after learning how to use SDL) to create wrapper functions that loaded images and set colorkeys. These functions are found in images.c.

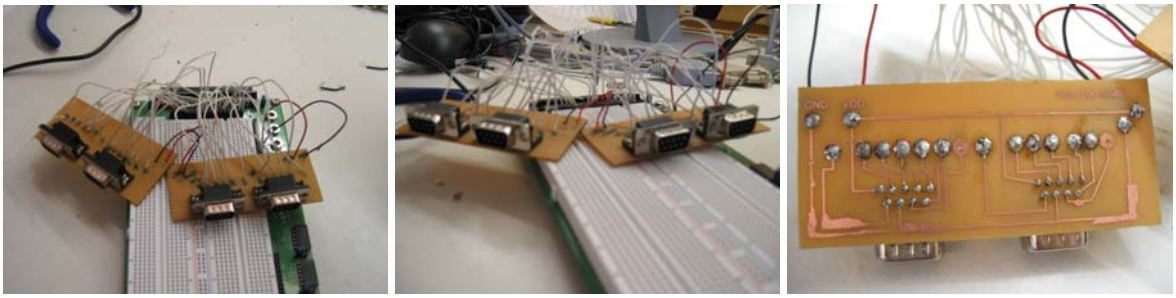
For audio playback, we utilized the methods in SDL_audio (SDL_LoadWav) to ease the loading of the wav file into memory. Once placed into memory, we just needed to dump sound data from memory into the FIFO buffers of the AC97 codec. We never had any problems with data corruption as a result of using SDL. The biggest challenge with audio playback is that we had to consider the endianness of the data being passed into the FIFO buffers. The main issue with the sound mixing in the final implementation was the bus traffic that results from using both PowerPCs. When PPC0 is writing to the LCD framebuffer and PPC1 is writing to the FIFO buffer, the two PPCs, both of which have equal priority (priority 3), compete for PLB bus traffic. This causes the screen to "shake" because the screen cannot be updated quickly enough.

The following is the status as of the final public demo.

Graphics – 100% working. The software aspect of the graphics (i.e. SDL) is complete. The hardware aspects of graphics are also complete given the constraints we dealt with including the Xilinx hardware and problems with our game code.

Audio – 100% working. We have single audio wav playback and software sound mixing working. While the first PPC writes to a static array in memory to place the sound clips, the second PPC reads the sound clips and nullifies the pointers corresponding to the sound clips that just played. This creates an open spot in the static array where that clip once was.

I/O – 100% working. We have all the Sega Genesis controllers connected to DB9 connectors which are soldered to a custom PCB which one of us made (see the picture below). Our code polls each controller for which buttons are being pressed and ties the response to our game code for movement or for dropping bombs.



Game play – 75% complete. There are still many bugs that need to be fixed. In terms of player movement, we are having minor problems with the clipping. Most of our problems stemmed from placing bombs and the bomb explosions. The biggest problem that we faced was the recursive bomb stack. Occasionally, the stack would overflow and cause our game to crash.

To improve what we currently have, we would like to fix the bugs in our game code. Additionally, we would like to integrate more power-ups, whether they are traditional to existing versions of Bomberman or our own (the most popular suggestion for a power-up from our classmates was a rocket-launcher). AI is also another possibility as an extension that we would like to implement, to allow 1-3 people to play in a game with 4 bombermen.

What we learned

It would have been nice to know at the beginning of the project the following things:

- Limitations of the board, including understanding that we had limited BRAM and that running out of BRAM makes designing more difficult.
- How to port ISE projects into EDK
- Learning basic VHDL
- Understanding bus architecture and how to use it effectively
- Knowing that XMD changes timing so that the framebuffer won't be aligned to the top left corner of the screen

The worse decision we made was to have only one person working on game code. The game code ended up being more complicated than we anticipated, and we would have had a more complete game had more than one person worked on it. Also, had we known that processor 1 would saturated our bus traffic, we probably would've opted to do sound channel mixing in hardware, instead of the framebuffer.

One of the best decisions we made was to use SDL for graphics as it allowed us to develop our game on our own computers and not have to use the horrible development tools provided within EDK.

We would like to impart the following words of wisdom to future 18545 students:

- Make sure you start on the hardware early as it takes an infinite amount of time to debug.
- Look at synthesis reports very carefully to see if the synthesis tools are killing your design.
- Don’t underestimate how hard it is to port your software to the Xilinx board.
- Don't take many hard courses with this class as it will put more pressure on group members and reduce your chances of success.

Individual Pages: Mario

I was the main person in charge of porting SDL to the board.

- I had help from Henry getting the reading and writing functions in SDL to work properly. He wrote a compact flash version of fseek and ftell.

Approximate Time: 100 hours

I debugged all of the Xilinx software to figure out weird bugs in our code. (i.e. sysace_fwrite not writing to the screen correctly).
Approximate Time: 20 hours

I reverse-engineered the algorithms SDL was using to perform their blitting operations so that I could pass these on to Neha for implementation.
Approximate Time: 16 hours

I created drivers to support functions like SDL_UpdateRects which have to implemented natively on each platform to accomodate for different framebuffer accessing schemes. I used the IPOD drivers as reference.
Approximate Time: 24 hours

I set up all the memory addresses in the project. I figured out how to partition memory so that we could successfully implement malloc, as well as static structures in memory that malloc couldn't touch.
Approximate Time: 40 hours

I helped get the baseline sound implementation working by porting the relevant sections of SDL_Audio. I also set up the static array in memory for both processors to read/write from.
Approximate Time: 8 hours

I wrote and simulated the framebuffer in ModelSim, and ported it to the board.
Approximate Time: 30 hours

I figured out the Xilinx IPIF interface with Neha. I debugged the framebuffer with Neha's help.
Approximate Time: 80 hours

I worked on the game code to try to remove some of the bugs. I got the recursive bomb explosion working.
Approximate Time: 6 hours

I was the main system integrator for all of the components since I had alot of experience with software and hardware integration.
Approximate Time: 130 hours

Individual Pages: Neha

I figured out how to get the I/O integrated into EDK. I determined that the SNES controller interface would be much harder than the genesis controller to get working, especially since I tried to get the SNES controller to work. I integrated the debouncer module on the Verilog controller module.
Approximate time: 45 hours

I helped debug all of the Xilinx software to figure out weird bugs in our code.
Approximate Time: 20 hours

Helped reverse-engineered the algorithms SDL was using to perform their blitting operations.
Approximate Time: 16 hours

I got the source for the Random Number Generator and ported it into the EDK project, including debugging and testing.
Approximate Time: 10 hours

I wrote the 32 x 32 block blitter and determined it would not fit on the board.

- I also wrote the alpha blended and the RLE block given a 32x 32 block, and also a 32x46 block (for players).
- These were too large to fit on the board as a a result of the BRAM drought.

Approximate Time: 45 hours

I helped figure out how to partition memory so that we could successfully implement malloc, as well as static structures in memory that malloc couldn't touch.
Approximate Time: 32 hours

I helped write and simulate the framebuffer in ModelSim, and helped port it to the board.
Approximate Time: 30 hours

I helped Mario figure out how to use the XILINX IPIF, and helped him debug the framebuffer using Chipscope and ModelSim.
Approximate Time: 80 hours

I wrote the 32 pixel line blitter after the bus interface was figured out.
Approximate Time: 12 hours

I extracted all the sprites used in the game (over 60 of them). I also resized them all to be uniform.
Approximate Time: 8 hours

I figured out the correct sequence of moves and to display explosions correctly.
Approximate Time: 15 hours

Helped search for explosion sounds for when bombs explode.

Approximate Time: 3 hours

I wrote software code for polling the controllers. Designed the circuitry for the controllers
Approximate Time: 6 hours

I found and connected temporary DB9 connectors from the digilent accessory bread board to the Genesis Controllers.
Approximate Time: 2 hours

I designed the circuitry, including schematic and layout, for the controllers to make PCBs.
Approximate Time: 14 hours

I made PCBs for the controllers, including etching and drilling holes.
Approximate Time: 8 hours

I tested the game code.
Approximate Time: 5 hours

Class Impressions:
I felt that there was a huge learning curve for this course, where we not only had to figure out how to create our project, but we also had to learn how to use the Xilinx boards as well as the software that comes with them. The software was very buggy and unstable. I would have also appreciated if the labs that we did in the beginning were more helpful in understanding what we need to do later for the project.

Individual Pages: Jong

Things done:

- Interfaced Sega Genesis 3-button gamepads to use in our design (2-3 weeks)
- Wrote game code using C and SDL library (9-10 weeks)
- Ported game code to Xilinx EDK (last 2-3 weeks)
- Random issues getting game to work on board (last 2-3 weeks)

Class comments:

- In-class lab on how to use EDK, specifically on how to interface with verilog modules for peripherals, would have been very helpful
- Some behaviors were just completely random. For example, it is impossible to change the core clock frequency after you start a new project. Everything in the EDK, including all menu settings and .MHS and .MSS files will say it is running at the new clock frequency, but it is actually running at old clock speed. I eventually found some obscure website that said you have to add a multiplier in DCM (Digital Clock Manager) and create a new clock signal and connect it to PPC manually. Bingo! Worked like a charm. Restarted the board and it was running at old clock speed again. At this point it is impossible to change the clock speed. Must start new project. I actually have no idea how the class can be changed to deal with issues like this. Just felt the need to mention it.

Resources

Link to SDL sources
http://www.libsdl.org/index.php

Tutorial on SDL
http://lazyfooproductions.com/SDL_tutorials/index.php

ALSA Sound Mixer
http://www.suse.de/~mana/alsa090_howto.html
http://gentoo-wiki.com/HOWTO_ALSA_sound_mixer_aka_dmix
http://alsa.opensrc.org/index.php?page=DmixPlugin

Sega Game Pad Documentation
http://www.stanford.edu/class/ee108a/documentation/gamepad.pdf

Xilinx Expansion Header Pinouts
Xilinx manual p.45

Debouncer and other FPGA programming information
http://www.fpga4fun.com/Debouncer.html

Source code for different controller for different FPGA board (for reference)
http://www.ugrad.physics.mcgill.ca/~beek/as2/

BYU Group Project Reference
http://www.et.byu.edu/groups/ececmpsysweb/cmpsys.2005.winter/teams/league/groups/league/final/

OpenCores
http://www.opencores.org/projects.cgi/web/systemc_rng/overview