strEEtfight

Alan Armero
Cansu Cabuk
Daniel Mesko

Overview

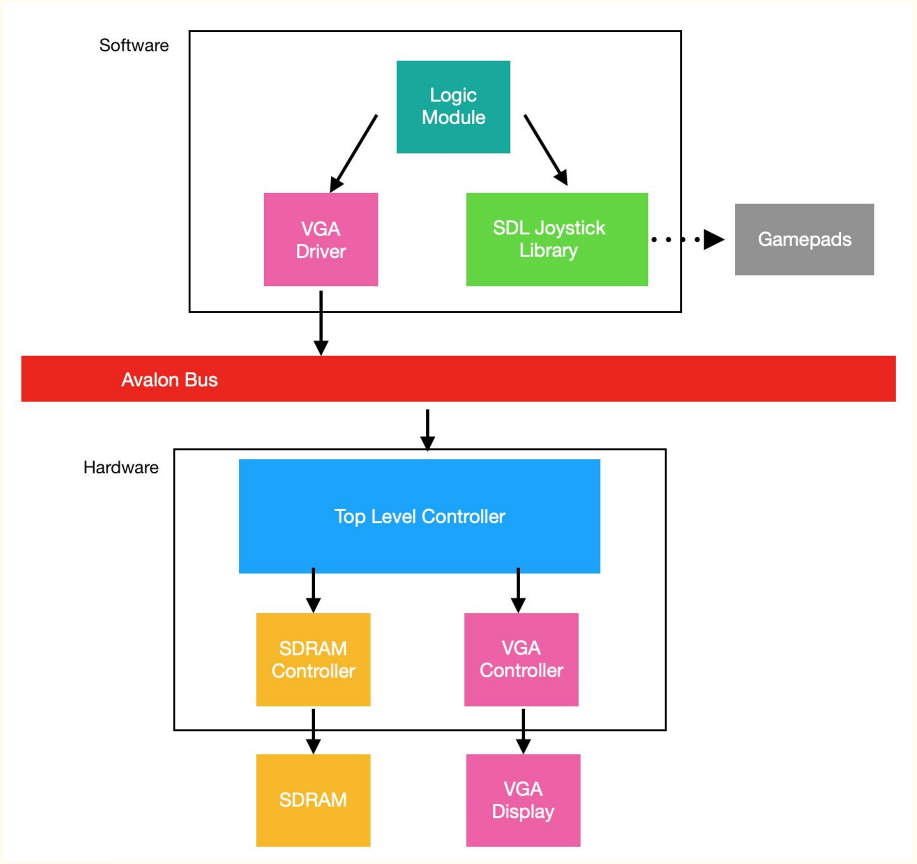
strEEtfight!

- A 2D, 2 player boxing game.
- Players move around and try to hit one another. When a player lands a hit, the other player loses a life. When a player loses all three lives, they lose the game.
- Players interact with the game through gamepads - a joystick and two buttons. Joystick controls movement, button 1 throws a punch, and button 2 a kick.

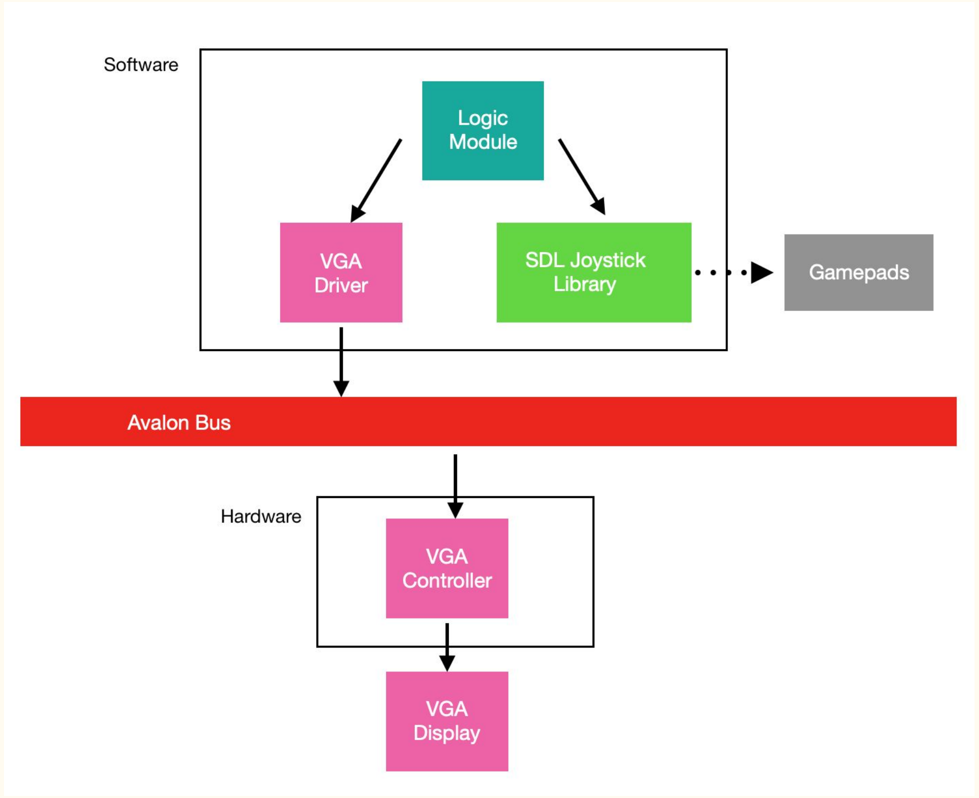
Gameplay

- With joystick, players can move left and right, and can jump. With buttons, players can throw a punch or a kick.
- Players have three lives. When one player lands a punch or a kick, the other player loses a life. A player loses the game when they run out of lives.
- If a player jumps when the other is throwing a punch, they do not incur damage.

System Design (Intended)



System Design (Final)



Hardware Components (Intended)

- Vga display Module
  - A line-out to interface with the sprite memory
    - Offsets, sprite magics, sizes, are all #define ‘d
  - Features an internal double-frame buffer
    - A read buffer and a write buffer to reduce screen tearing
  - Pulls render requests from a request fifo
  - Decodes render requests
    - Which sprite to write, lookup up its size and offset, then initialize counters for writing
    - Also has a special instruction for render -- means swapping the read and write buffer

Hardware Components (Intended)

- Vga display Issues:
    - Too much memory: double buffering is a cool feature, but stretched our resources too thin
        - But we couldn’t figure out how to read and write two different addresses safely at the same time. Even if we could deal with screen tearing, we couldn’t figure out how to read and write two locations in the buffer at the same time
    - Clearing the buffer was a problem. We tried destructive read, but this turns out to be an issue if it takes longer than one frame to render the whole queue

Hardware Components (Intended)

- SDRAM controller
  - ROM is too small to house sprite data, so we thought we'd hook into sdram through mem map interface and write to it before running the game, then rip it off at runtime into the 4.5MB on-chip memory
    - That was a mistake, but more on that later
  - Responsible for communication with the 64MB data store off chip
  - Connected to the the rest of the circuit through through a dual-clock, full-duplex fifo
  - On reset, the controller defaults to write mode
    - It'll write everything from the buffer to sdram sequentially, to reduce decode overhead
    - Once it receives a special toggle, it'll switch to read mode
  - In read mode, the hardware simply counts up and reads from memory until it's told to stop with an async pause signal

Hardware Components (Intended)

- SDRAM Controller Issues:
  - Complexity: Very hard to implement and debug. Might have worked if we hadn’t incurred issue 2
  - Flawed assumptions: we assumed we could load all of the sprite data into memory on-chip (there’s about 4.5 MB)
    - Why do this? There’s not good reason. In essence, we’re writing over avalon, to store data in sdram, to rip the data back out of sdram…. Such a system increases complexity and decreases throughput, but we didn’t make the connection at first :(
    - The better way to use SDRAM for making any gains in terms of space savings would be to implement a paging system which would swap sprites out of memory on an LRU basis, similar to linux page swapping

Hardware Components (Intended)

- SDRAM Controller Issues (cont.):
    - We tried the former, realized the mistake, then tried the latter. Complexity was too high though and we ripped out the SDRAM altogether, opting to write data to on-chip memory over the avalon bus straight-up
        - After much debugging, we still ran out of memory, but didn’t realize it due to a bug where Quartus wouldn’t allocate one of our blocks
        - Not enough on-chip memory for the game, and without implementing the paging system for SDRAM we were stuck and out of time

Hardware Components (Intended)

- Top Level Module, that coordinates communication between the following modules:
    - The top level module is responsible for decoding commands from the hps
        - Varieties include a read/write to sdram and reads/writes to vga display
    - VGA Display Controller / VGA Counters
    - Dual-clock FIFO Buffer between SDRAM and
    - Memory Controller
    - SDRAM Controller
- PLL for 4 different clocks

Sprites (Intended)

- 7 character sprites, that change according to gamepad input
- A heart sprite - 1,2,3 displayed above character according to health
    - Storing the sprites on chip was not possible, so we decided to use SDRAM
- Character loses a life when the characters are overlapping and one performs a punch or a kick
- Remove the sprite backgrounds (uniformly colored) in hardware using selection statements
- For opponent character, use mirror image

Hardware Components (Final)

- Based on Lab 3
- VGA Controller (Top Level)
  - Takes player 1 & 2’s x and y axis values, as well as their number of lives
  - Draws fixed-color blocks representing each player, hardcoded heart locations

Software Components

- Top Level
  - Initialize inputs (joysticks, mutexes), VGA device, scene object list.
  - Loops indefinitely, calling:
    - update _inputs
    - update_scene
    - update_render
  - Frees memory and shuts down joysticks on SIGINT
- Input Module
  - Fork a thread to handle joystick interrupts, then update global bitmask for joystick state (buttons pressed, joystick axes values).
  - Main thread later comes in and reads bitmasks, updating scene object state accordingly.

Software Components

- Scene Module
  - Maintain a linked list of scene_objects (players).
  - Expose an iterator function that allows other functions to traverse the list and call specific functions on each objects.
- Player State Module
  - Instantiate player scene_objects and their corresponding player-specific fields (health and player number).
  - Update player's scene_object based on input bitmasks - behavior (sprite), position.
  - Determine if player's positions overlap and if a punch/kick is thrown, update health accordingly.
- Render Module
  - Traverse scene list and call VGA Driver functions to add player sprite to in-hardware render queue.
  - Call VGA driver to add heart sprites corresponding to player health.

Software Components

- VGA Device Driver
    - Exposes ioctl calls for sending sprite magic numbers and positions to the VGA controller.
- Sprite Loader
    - Reads sprite pixel data into program memory
    - Intention was to map it into /dev/mem for hardware controllers to pull from.

Future Work

- Add more functionality in terms of movement
- Audio!
- Replace player blocks with character sprites
- Load sprites on SDRAM and be able to fetch from it in hardware

Lessons Learned

- Work on software and hardware at the same time - very easy to think you're ahead of the game if the two components work separately - but integration is the real test.
- Unit test everything, one feature at a time...especially hardware components, given the amount of time they take to compile.
- Plan the system design with hardware realities in mind - don't just assume you'll be able to get it to do what you want.