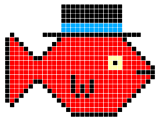

02113 - Digital systems design project
Course manual

Luca Pezzarossa

June 3, 2024

Preface

Welcome to the course 02113 - Digital systems design project!

In this 3-week course, you will create a 2D arcade-style game, entirely implemented in hardware using the Chisel hardware description language. This hands-on project is designed to solidify your understanding of digital design principles, enhance your problem-solving skills, and provide practical experience in hardware design.

Your objective is to design and implement the game logic for a 2D arcade game. You will leverage the provided graphical engine and VGA display logic to bring your game to life. This course will require you to think creatively, manage hardware resources efficiently, and ensure that your design meets the performance requirements of real-time gaming systems.

The game you design will be implemented and run on your Basys3 FPGA board. The Basys3 board will serve as the hardware platform for your game, utilizing its VGA output capabilities to display your game graphics on a monitor. The main player inputs are the 5 buttons integrated into the board. In addition, the switches and LEDs on the Basys3 board can also be used for testing and debugging.

This course is structured as a hands-on experience, encouraging full immersion in practical hardware design. Although some aspects of the course can be challenging, it is equally important to enjoy the process of implementing your project and exploring the creative sides of hardware and game design. Let's aim to maintain a relaxed and enjoyable atmosphere throughout the course!

Important note: Before starting with the design and implementation (i.e. coding), please read this document top-to-bottom, so you know the overall expectations and development flow.

Contents

1 Practical information

This section provides the practical information related to the course project, such as group forming, schedule, access to help, and deliverables.

1.1 Group forming

This assignment should be carried out in **groups of three people** (groups of two are also possible but less preferred). You are free to select your group members. Groups should be registered as soon as possible in the DTU-Learn group forming facility. If you experience difficulties forming a group, please contact the teacher.

When forming a new group, please make sure that you align expectations between the members. To achieve this, we recommend having a discussion about each member's availability, work habits, and goals for the course to ensure a smooth and collaborative experience.

1.2 Presence, schedule, and review meetings

We recommend being present during course hours in the course classroom to make the most of the available resources and support. Please note that the equipment, including screens and cables, cannot be moved from the classroom.

During the course, we will have three review meetings to ensure you are on track and to provide you with feedback. These meetings will be an opportunity to discuss your progress, address any challenges, and get guidance from the instructors. The exact dates and times for these review meetings will be announced on DTU-Learn.

Review meetings are mandatory and will be held at key milestones throughout the assignment period. At each meeting, you should be prepared to present your current progress, demonstrate any working components, and discuss any issues you are facing.

The review schedule is as follows:

- **Initial review meeting**: An overview of your game concept and initial design. Focus on the game dynamics and sketch the hardware design.
- **Midway review meeting**: Demonstration of partially implemented game logic and integration with the Basys3 board.
- **Final review meeting**: Full demonstration of your game running on the Basys3 board. This is the exam of the course.

1.3 Access to help

We encourage you to seek help whenever needed. We will try to have one teacher available most of the time in the classroom. When all the teachers are occupied with other academic activities, you can continue working independently. We will try to have at least one slot per day for questions with the teachers.

We will set up a discussion forum (Discord server) for you to ask questions, share ideas, and collaborate with your peers. The forum will also be actively monitored by the teachers to provide timely assistance. The forum access link will be announced on DTU-Learn.

1.4 Deliverables

By the end of the course, you must submit the following deliverables:

- **Short report**: A short report describing the approach used in your implementation, including any design decisions, challenges faced, and solutions implemented. The report should also include diagrams and explanations of your game logic and hardware design. More information about the report is provided later in the manual.
- **Source files**: All the Chisel source code and any other files necessary for your game logic implementation, bundled as a ZIP archive. Ensure your code is well-documented and organized. To keep the size of the ZIP archive manageble, ensure that you hand in the source code only, without including generated files. Also, add the learning tasks files.
- **Demonstration video**: A short video demonstrating your game running on the Basys3 board. The video should highlight the key features of your game and any unique aspects of your implementation.
- **README file**: A README file that includes instructions on how to set up and run your game on the Basys3 board, as well as how to run any test cases you developed.

All deliverables must be submitted via the DTU-Learn course assignment page at the following location:

`DTU-Learn/Course content/Assignments/Final game`

In addition to the electronic submission, you will be required to present your game and demonstrate its functionality during the final day of the course. This presentation is an opportunity to showcase your work and receive feedback from your peers and instructors. During this occasion, we will also have a small friendly competition for the '02113 Best Game Award'.

2 Install the tools: Chisel, GTK waves, and Vivado

For this course, you need to install the Chisel development environment, GTK waves (or similar), and Vivado 2023.1. **If you already have these tools installed from a previus course (e.g., Digital electronics 2), you can skip this section.**

2.1 Setting up and working with Chisel

This section[1] is a guide on how to install the Chisel toolchain Windows, Linux, and Mac. In addition, we will also provide useful pointers to Chisel documentation and material online. We have tested the steps presented in the following at our best using our own computers. However, this does not guarantee that they will work flawlessly to any computer due to different flavours/configuration of your own machines. If you encounter problems that you cannot solve yourself with a Google search, contact a TA or the teacher during the laboratory session.

Chisel is just a library for Scala. And Scala is just a language that executes on the Java virtual machine (JVM) and uses the Java library. Therefore, you need to have Java OpenJDK 8 installed on your laptop. **It is important to use version 8 (also called 1.8) of Java, not a newer one. If you have a newer one installed you do not need to uninstall it, but you need to install also the required version.**

For working on the command line you should also install sbt, the Scala build tool. Please note that installing sbt will make the IntelliJ-build process a lot easier as well. A nice editor for Chisel/Scala is IntelliJ. In summary, the tools we need to install are:

- Java OpenJDK 8
- sbt
- IntelliJ
- GTKWave

Install Chisel on Windows

1. Install OpenJDK 8 (HotSpot) from AdoptOpenJDK, which will forward you to the Adoptium website for download. **It is important to use version 8 (also called 1.8) of Java, not a newer one. It is also important that you enable the "Set JAVA_HOME" for installation.**

[1] This section contains material derived from the open-source documentation developed by Martin Schoeberl and available here and here. Martin Schoeberl is an associate professor at DTU-Compute in the Embedded System Engineering section.

2. Install sbt.
3. Install IntelliJ (the free Community edition) and create a desktop shortcut.
4. Start IntelliJ to finish the setup.
    - Select the UI theme you prefer better in the Customize tab.
    - Select Install for Scala in the Plugin tab.
    - If required when importing a project, select the JDK 1.8 you installed before **(not Java 11!)**.
        - On Project JDK select New.
        - Select JDK.
        - Select the path to your OpenJDK 8 installation, usually `C:\Program Files\AdoptOpenJDK\jdk-8.0.232.09-hotspot\`

Install Chisel on Linux

1. Install OpenJDK 8 **(it is important to use version 8 (also called 1.8) of Java, not a newer one)** with the terminal command:

    ```
    sudo apt install openjdk-8-jdk
    ```

2. Install sbt (if on Ubuntu, deb archive is preferable).
3. Install IntelliJ (the free Community edition).
4. Start IntelliJ to finish the setup.
    - Select the UI theme you prefer better in the Customize tab.
    - Select Install for Scala in the Plugin tab.
    - If required when importing a project, select the JDK 1.8 you installed before **(not Java 11!)**.
        - On Project JDK select New.
        - Select JDK.
        - Select the path to your OpenJDK 8 installation, for example `/usr/lib/jvm/java-21-openjdk-amd64/bin/`

Install Chisel on macOS

1. Install OpenJDK 8 from AdoptOpenJDK. **It is important to use version 8 (also called 1.8) of Java, not a newer one.**
2. Install sbt with the terminal command: `brew install sbt`
3. Install IntelliJ (the free Community edition).
4. Start IntelliJ to finish the setup.
    - Select the UI theme you prefer better in the Customize tab.

- Select Install for Scala in the Plugin tab.
- If required when importing a project, select the JDK 1.8 you installed before **(not Java 11!)**.
  - On Project JDK select New.
  - Select JDK.
  - Select the path to your OpenJDK 8 installation.

2.2 Installing GTKWaves and visualizing waveforms

As previously mentioned, in hardware, everything is parallel and debugging is mainly done graphically using waveforms that shows the value of all the signals of your circuit at any moment in time. Waveforms are generated by the Chisel simulator as `.vcd` file and can be loaded and visualized using GTKWaves. Please note that GTKWaves is not the only option. If you have another vaweform visualizer already installed in your machine, you do not need to install GTKWaves.

To install GTKWaves:

1. In Windows, install GTKWave and put a link to the executable on the desktop. When installing GTKwaves for Windows, please download the binary version form the web-page.
2. In macOS, install GTKWave.
3. In Linux, Install GTKWave from here or with the terminal command:

   ```
   sudo apt install gtkwave
   ```

Figure 1 shows the main parts of the GTKWaves user interface. You need to move the signals you are interested in from the 'Available signals' area into the 'Selected signals' area.

Note: In the 'Available signals' area, you may find a lot of generated signals named `_GEN_[...]`. Just ignore these signals.

Note: In GTKWave, you can just reload waves when you re-run a simulation (File -> Reload waveform or use the CTRL + r shortcut) to update the waveform without the need to open a new file and set up GTKWave again.

2.3 Install AMD Vivado 2023.1

Vivado 2023.1 is a comprehensive tool suite for FPGA design and development, available for Windows and Linux operating systems. Unfortunately, it is not available for macOS. Follow the steps below to install Vivado 2023.1 on your system.

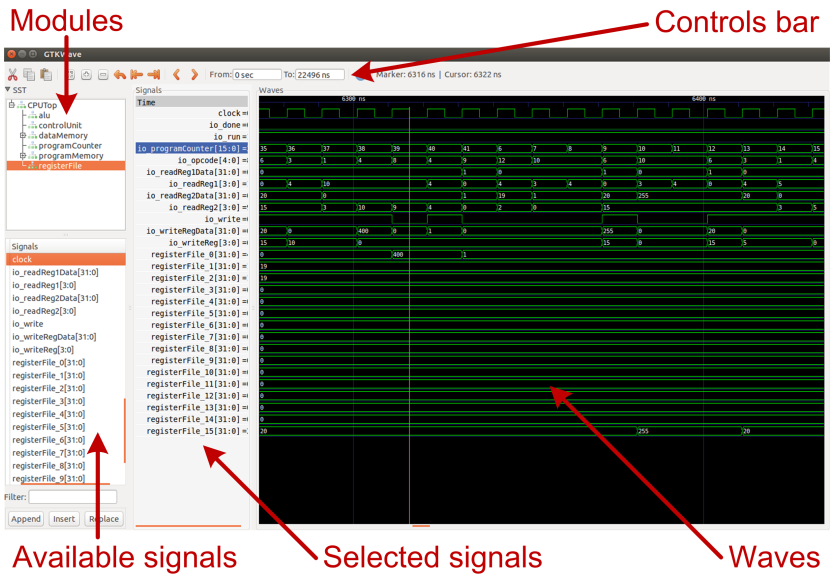

Figure 1: The main parts of the GTKWave user interface.

1. Visit the Xilinx Vivado download page here.
2. Select the Vivado 2023.1 version and choose the proper installer (Windows or Linux). You might need to register to be able to download Vivado.
3. Locate the downloaded executable file and double-click to run it.
4. After extraction, the installation wizard will start automatically.
5. Accept the license agreements and select the installation directory.
6. Choose the desired installation options (see Figure 2).
7. Click “Install” to begin the installation process.
8. If asked, accept the installation of the drivers to connect to the FPGA.
9. Once the installation is complete, click “Finish”.
10. Launch Vivado from the Start menu or desktop shortcut.

2.4 Known issues and solutions

In the following, we explain how to overcome some known issues with the installation of the Chisel development environment. These solutions are based on issues encountered in other courses. Since tools are continuously evolving, the solutions might not work or new issues might appear.

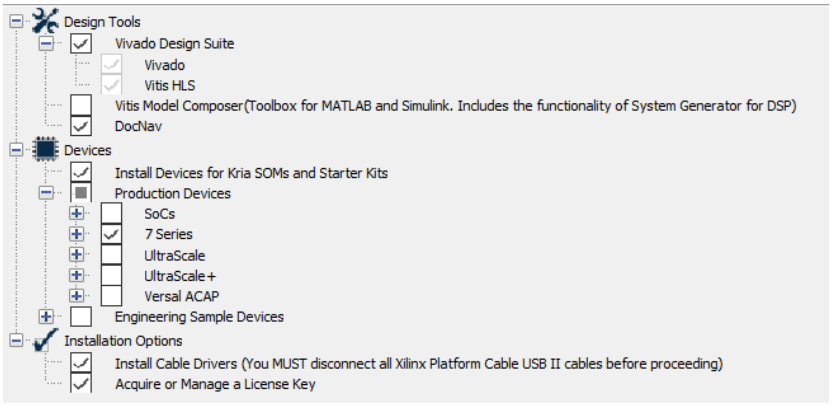

Figure 2: Selection of the Vivado installation options.

Issue 1: Mac laptops with ARM-based processor

If you have a Mac laptop that uses the new ARM-based processors (e.g., M1 chip), you may not be able to install sbt with the provided instructions. Please try the following.

1. Install the package manager SDKMAN with:

   ```
   curl -s "https://get.sdkman.io" | bash
   ```

2. Close and reopen terminal or run the command:

   ```
   source "$HOME/.sdkman/bin/sdkman-init.sh"
   ```

3. Install Java 8 with:

   ```
   sdk install java 8.302.08.1-amzn
   ```

4. Install sbt with:

   ```
   sdk install sbt
   ```

5. Compile sbt with:

   ```
   sbt compile
   ```

Issue 2: Missing JAVA_HOME or wrong Java version

When running the tests in Chisel, the sbt commands expects to use Java 1.8. Sometime it is sufficient to change the Java version in IntelliJ by going to File → Project structure → Project SDK and selecting the right Java version. However, in some case this does not apply to the terminal integrated into IntelliJ. If this happens, try the following solution:

In Windows and Linux: Go into File → Settings → Tools → Terminal → Environment Variables (open it, do not write in the text box). Add an environment variable using the + button. The variable name is JAVA_HOME and the content is the path of the installation of Java 1.8

In MacOS: Go into IntelliJ IDEA → Preferences → Tools → Terminal → Environment Variables (open it, do not write in the text box). Add an environment variable using the + button. The variable name is JAVA_HOME and the content is the path of the installation of Java 1.8 You can have an hint of the path of the installation of Java 1.8 by looking at the sbt shell in IntelliJ. The path is printed on the first line (please just copy the path without the executable name).

Online Chisel documentation and material

The main reference book for learning Chisel is:

- *Martin Schoeberl, Digital Design with Chisel, 2nd edition, 2019* (free, open-access book available here). The book source is available here.

In addition, you can take a look at the following online documentation and material:

- The official Chisel website. Click Chisel3 for documentation.
- A collection of FAQ from another course using Chisel at DTU.
- The Chisel Cookbook: large FAQ and introduction to Chisel.

3 Background

3.1 Objectives of the course

This course provides a hands-on approach to hardware design, focusing on the development of a 2D arcade-style game entirely in hardware. The course is structured to guide you through the process of designing, implementing, and testing a hardware-based game. We provide a robust graphical engine and VGA display logic, allowing you to focus on game logic without the need to design basic graphics functionalities from scratch. This engine handles sprite rendering, background management, and VGA signal generation.

You are required to design and implement the game logic within a hardware block named `GameLogic`. This involves creating finite state machines (FSMs) and datapaths to manage game states, sprite movements, collision detection, scoring, and other game mechanics.

At first, you will be guided through a series of learning tasks, where you can get acquainted with the provided code, tools, and graphic engine functionality. Then you will move on to developing your own game.

3.2 Getting the code

All the code is provided as a ZIP archive on the DTU-Learn course page at the following location:

`DTU-Learn/Course content/Content/Code`

3.3 Code structure

Figure 3 shows the folder/file structure of the provided code.

```
<project root>
├── memory_init
│   ├── backbuffer_init.mem
│   ├── backtile_init_[0-31].mem
│   └── sprite_init_[0-15].mem
├── tile_generator
│   ├── tile_generator.html
│   └── ...others...
├── src
│   ├── main
│   │   ├── scala
│   │   │   ├── GameLogic.scala
│   │   │   ├── GameLogicTask[0-8].scala
│   │   │   ├── GameTop.scala
│   │   │   ├── GameUtilities.scala
│   │   │   ├── GraphicEngineVGA.scala
│   │   │   ├── Memory.scala
│   │   │   └── Top.scala
│   │   └── ...others...
│   └── test
│       ├── scala
│       │   └── GameLogicTester.scala
├── vivado
│   ├── Basys3Game
│   └── NexysA7Game
└── ...others...
```

Figure 3: The folder/file structure of the provided code.

The folder `.../src/main/scala/` contains the hardware modules written in Chisel. The files you need to explore and work on are:

- `GameLogic.scala`: Contains your game logic implementation implementation.
- `GameLogicTask[0-8].scala`: These files contain your game logic implementation for the learning tasks from 0 to 8. Task 0 and task 8 are given to you already solved.
- `GameTop.scala`: Defines the top-level module for the game. Here you select which game logic Moule to use (see comments in the file).

The folder `.../src/test/scala/` contains the tests written in Scala/Chisel/-Java. An empty test file is given to you and you need to implement your tests if needed.

The folder The folder `.../vivado/` contains the project files and configurations for synthesizing and implementing the game on FPGA boards. Open the right project with Vivado.

- `Basys3Game`: Contains Vivado project files and configurations specific to the Basys3 FPGA board.
- `NexysA7Game`: Contains Vivado project files and configurations specific to the Nexys A7 FPGA board.

The folder `.../memory_init/` holds memory initialization files used for setting up the initial memory contents for the background buffer, the sprites ROMs, and the background tiles ROMs.

- `backbuffer_init.mem`: Initializes the background buffer memory.
- `backtile_init_[0-31].mem`: 32 files to initialize the ROMs for background tiles, with each file corresponding to a specific tile.
- `sprite_init_[0-15].mem`: 32 files to initializes the ROM for sprites, with each file corresponding to a specific sprite.

The `.../tile_generator` folder contains the web-based tool for generating and editing tiles used in the game, named `tile_generator.html`. Use this tool to generate the tile ROM content in a graphical way.

3.4 The graphic engine

The provided graphic engine offers functionality for managing sprites and backgrounds. Specifically, it has the ability to control sprite positions, visibility, and flipping, combined with a larger-than-screen background that supports scrolling.

In the following, we explain the features offered by the graphic engine.

Sprites and background tiles

In 2D games, sprites and background tiles are essential elements that define the visual aspects of the game. Sprites are the graphical representations of characters or objects that move and interact within the game environment. Backgrounds are static images that make up the scenery and provide context for the gameplay. Very often, these images are formed by putting together a set of background tiles. Figure 4 shows an example of background and sprites.

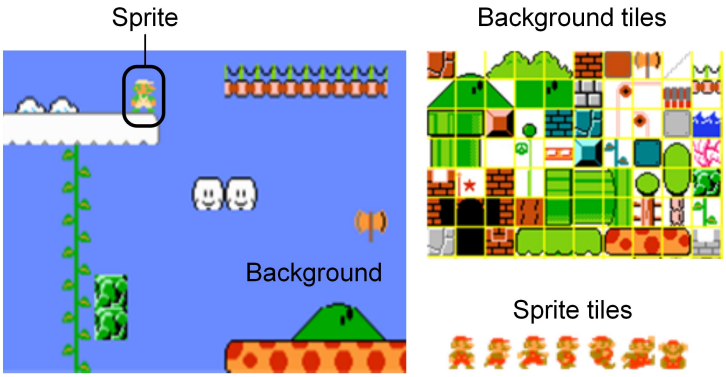

Figure 4: An example of background and sprites (and the relative tiles).

In our game engine, both sprites and background tiles are 32 by 32 pixels. The texture (bitmap) of these sprites and background tiles is stored in a Read-Only Memory (ROM). We use memory initialization files to load these tiles, and we have a total of 16 sprites and 32 background tiles bitmaps available for use. If sprites overlap when shown on the screen, sprite0 is the top-most, and sprite 15 is the bottom-most. Figure 5 shows an example of a sprite tile and a background tile for our system.

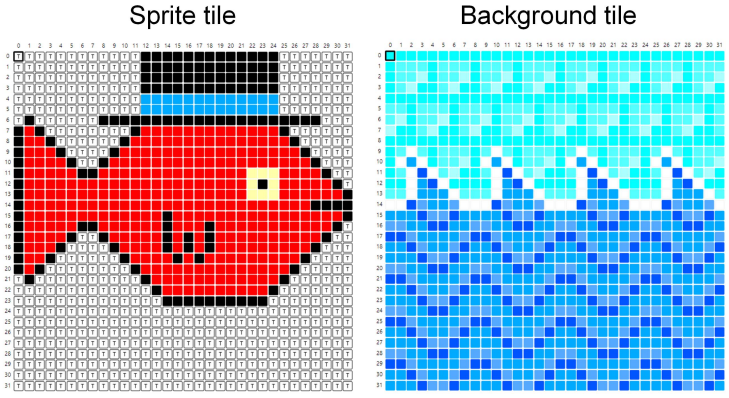

Figure 5: An example of a sprite tile and a background tile for our system.

The tile generator tool (provided to you) can be used to design the sprites and background tiles, generating the ROM initialization memory content. Each

entry in the ROM (i.e. each line in the initialization file) describes one pixel. There are 7 bits per pixel: the Most Significant Bit (MSB) is for transparency, followed by 2 bits each for red, green, and blue color values, as shown in Figure 6.

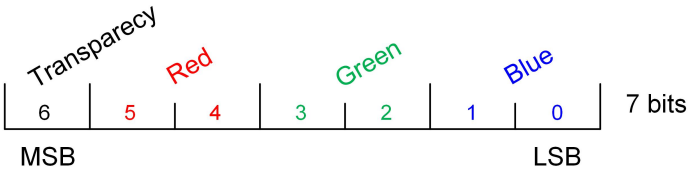

Figure 6: Pixel format: 1 bit for transparency, 2 bits for red, 2 bits for green, and 2 bits for blue.

Screen and coordinate system

The game screen resolution is 640 by 480 pixels (horizontal by vertical). The origin of the screen coordinate system is located at the top left corner of the screen. For each sprite, the graphic engine allows control over its position, visibility, horizontal flipping, and vertical flipping. The origin of each sprite is also located at the top left corner. Figure 7 illustrates this concept.

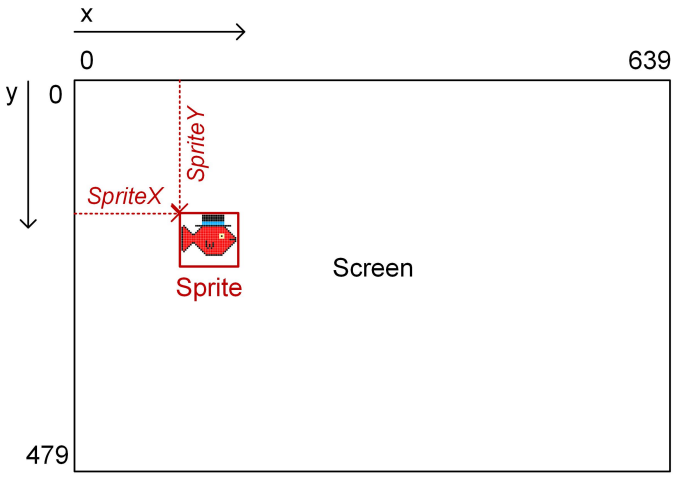

Figure 7: The screen and sprite coordinate system.

3.4.1 Background and viewbox

The actual background in our game is larger than the screen, which we refer to as the "larger background". This larger background consists of 40 by 30 tiles, making it four times the size of the screen. The section displayed on the screen is the part that falls inside the viewbox, which is 640 by 480 pixels. This setup

Figure 8: Large background mapping and viewbox configuration.

allows for background scrolling (an advanced feature that you do not need to use if you do not want to). By default, the viewbox is placed in the top left corner

The large background is initialized using the `backBufferInit` file according to the mapping shown in Figure 8. Each memory entry (from address 0 to 1199) in this file contains the number of the background tiles in the ROMs to be displayed. Since we have available 32 background tiles, each entry of the `backBufferInit` is a 5-bit value.

3.5 System architecture

Figure 9 shows the block diagram of the system architecture used in the project. The architecture consists of several key blocks, which are described in the following.

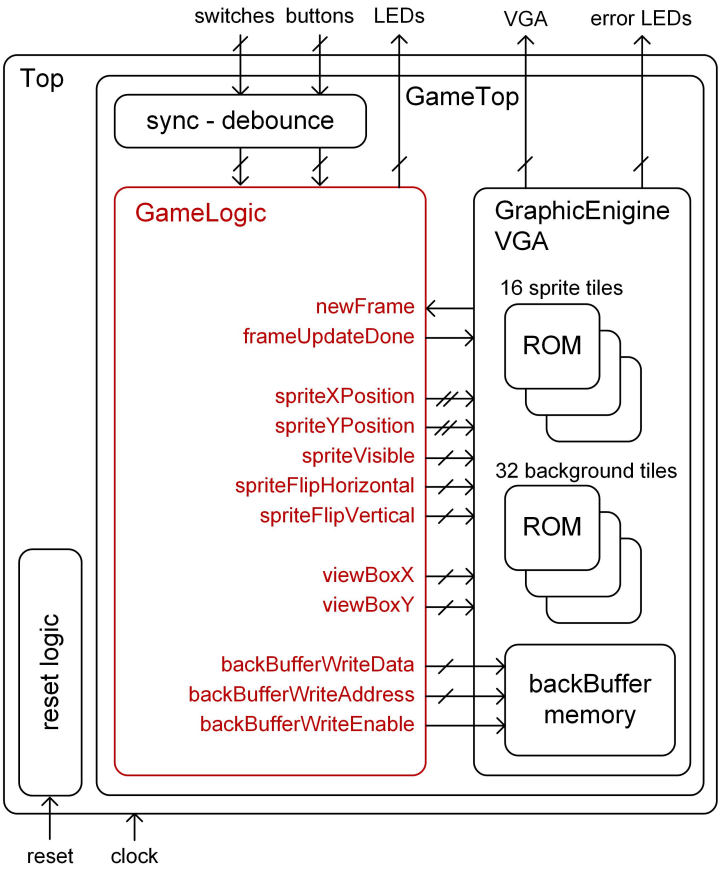

Figure 9: The block diagram of the system architecture provided to you. The block in red is where you need to write your game logic.

Top: The `Top` module is the very top-level module of the system. It just includes

reset synchronization logic.

GameTop: The `GameTop` module is the inner top-level module. Within this module, you can select which game logic to use by uncommenting specific lines. It also contains the button and switch synchronization and debouncers, as well as some reset delay logic.

GraphicEngineVGA: The `GraphicEngineVGA` module is the provided graphic engine that manages the rendering of sprites and background tiles on the VGA display. It handles the position, visibility, and flipping of sprites, as well as the display of the background based on the initialized memory content.

GameLogic: The `GameLogic` module is the block where you will develop your custom game logic. This is the module highlighted in red in the block diagram. Your task is to implement the game mechanics, sprite interactions, and other game functionalities within this block. Pay special attention to the interface toward this block, as it defines how your game logic interacts with other parts of the system. The details of this interface are explained in the following sections.

3.6 Interface of the `GameLogic` module

The `GameLogic` module in Chisel is responsible for implementing the core game logic for a 2D arcade-style game. This is the module when you implement your game logic. Below is a description of each input and output in the `GameLogic` module interface.

Buttons (inputs)

There are 5 button inputs. They are of `Bool` type and they get `true.B` when pressed, `false.B` otherwise. These are the main inputs for the player.

- `btnC`: Center button input
- `btnU`: Up button input.
- `btnL`: Left button input.
- `btnR`: Right button input.
- `btnD`: Down button input.

Switches (inputs)

- `sw`: An 8-element `Vec` of `Bool` inputs representing the states of eight switches. Each element get `true.B` the shitch is pulled 'up', `false.B` otherwise. The switches can be used for various game configurations or modes or for debugging purposes.

LEDs (outputs)

- `led`: An 8-element `Vec` of `Bool` outputs representing the states of eight LEDs. These should be used for debugging purposes.

Sprite control (outputs)

- `spriteXPosition`: A 16-element `Vec` of 11-bit `SInt` (signed integers) representing the X positions of sprites. Each position ranges from -1024 to 1023 and refers to the top-left corner of the sprite relative to the top left of the screen. The number is signed since the sprite can be partially or totally moved out of the screen. See Figure 7.
- `spriteYPosition`: A 16-element `Vec` of 10-bit `SInt` representing the Y positions of sprites. Each position ranges from -512 to 511 and refers to the top-left corner of the sprite relative to the top left of the screen. The numbers is signed since the sprite can be partially or totally moved out of the screen. See Figure 7
- `spriteVisible`: A 16-element `Vec` of `Bool` outputs indicating the visibility of each sprite. `true.B` means the sprite is visible, and `false.B` means it is hidden.
- `spriteFlipHorizontal`: A 16-element `Vec` of `Bool` outputs controlling the horizontal flip state of each sprite. `true` indicates the sprite is flipped horizontally.
- `spriteFlipVertical`: A 16-element `Vec` of `Bool` outputs controlling the vertical flip state of each sprite. `true` indicates the sprite is flipped vertically.

Viewbox control (outputs)

- `viewBoxX`: A `UInt` output representing the X coordinate of the viewbox, ranging from 0 to 640. This defines the horizontal position of the viewbox within a larger background in pixels. It refers to the top-left corner of the viewbox relative to the top left of the larger background. See Figure 8.
- `viewBoxY`: A `UInt` output representing the Y coordinate of the viewbox, ranging from 0 to 480. This defines the vertical position of the viewbox within a larger background in pixels. It refers to the top-left corner of the viewbox relative to the top left of the larger background. See Figure 8.

Background buffer memory write port (outputs)

- `backBufferWriteData`: A `UInt` output representing the data to be written to the background buffer. The width of this `UInt` is determined by the logarithm (base 2) of `BackTileNumber` (which is 32), so the width is 5 bits.

- `backBufferWriteAddress`: A `UInt` output representing the address in the background buffer where the data should be written. This address is 11 bits wide (tile 0 to 1199 in the 40-by-30 larger background).
- `backBufferWriteEnable`: A `Bool` output enabling the write operation to the background buffer. When `true`, the data at `backBufferWriteData` is written to the address specified by `backBufferWriteAddress`.

The timing diagram for one write transaction to the background buffer is shown in Figure 10.

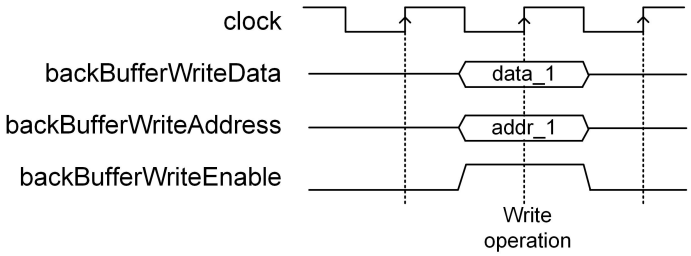

Figure 10: The timing diagram for one write transaction to the background buffer where `data_1` is written in `addr_1`.

Frame status (input/output)

- `newFrame`: This `Bool` input signals the start of a new frame. It goes high for 1 clock cycle every frame (60 times per second) and is used to synchronize the finite state machine (FSM) computation for updating the game logic.
- `frameUpdateDone`: A `Bool` output indicating the completion of a frame update. This signal should go high for 1 clock cycle to communicate that the game logic has computed the values for the next frame.

3.7 Frame timing

The frame rate for the game is 60 frames per second, which means that at a clock frequency of 100MHz, all calculations for sprite, viewbox, and background control need to be completed within a specific number of clock cycles (CC) per frame. Given the frame rate and clock frequency, each frame must be processed within $\frac{100\text{MHz}}{60} \approx 1.67$ million clock cycles.

The `newFrame` signal goes high for 1 clock cycle at the beginning of every frame, occurring 60 times per second. This signal should be used to start the FSM performing the computations required for updating the game status (i.e. sprite positions, viewbox positions, and writing to the background memory). When the `newFrame` signal goes high, all the input values (i.e., sprite positions, visibility, etc. are sampled and will be used for the frame rendering).

Once all calculations and memory writes are completed within the allocated time, the `frameUpdateDone` signal must be raised for 1 clock cycle to indicate the completion of the frame update. This process must be completed before the next `newFrame` signal arrives.

If the computation extends beyond the duration of a frame and a new `newFrame` signal is received before the `frameUpdateDone` signal is asserted, an error condition occurs. This error is indicated by turning on an error LED, and the current frame is skipped to maintain synchronization (more on errors later).

Additionally, there is a reserved period at the end of each frame, approximately 4000 clock cycles, during which the background memory cannot be written. Writing to the background memory during this period also results in an error condition, indicated by turning on an error LED, and the memory content will not be updated (more on errors later).

Figure 11 shows a visual representation of this process.

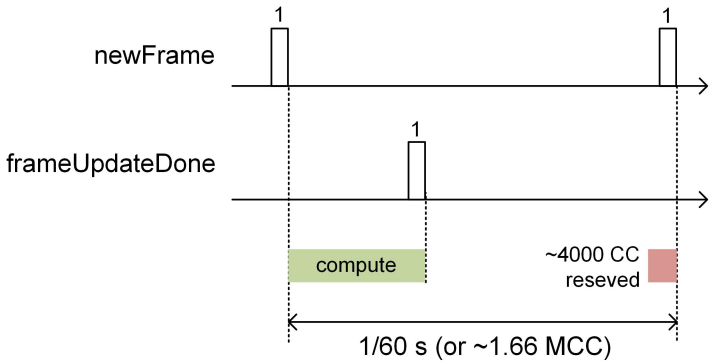

Figure 11: Frame timing diagram.

3.8 Mapping to the Basys3 board

The input and output signals are mapped to the Basys3 board as shown in Figure 12. Please note that the reset signal of the system is connected to the leftmost switch. The meaning of the error LEDs is described in the following subsection. If you use a Nexys A7 board, the mapping is similar (check the XDC configuration file for more details).

3.9 Error LEDs

Three error signals from the graphic engine are connected to LEDs to indicate various error conditions that may occur during the execution of the game logic, as shown in Figure 12. These error signals help in diagnosing issues related to frame timing, memory writes, and viewbox positioning.

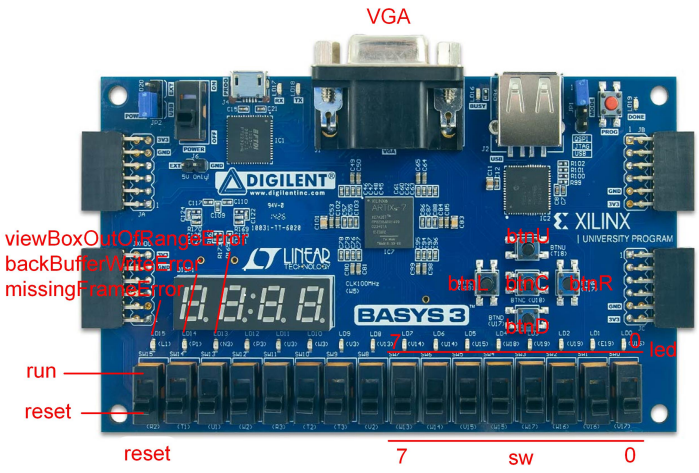

Figure 12: Mapping of the input and output signals for the Basys3 board.

The `missingFrameError` signal is triggered when the computation for updating the game logic, including sprite positions and background buffer updates, exceeds the duration of a frame. Specifically, if the next `newFrame` signal is received before the `frameUpdateDone` signal is asserted, this error occurs.

The `backBufferWriteError` signal is triggered when an attempt is made to write to the background memory during the reserved period at the end of each frame. This reserved period lasts for approximately 4000 clock cycles, during which background memory writes are not allowed.

The `viewBoxOutOfRangeError` signal is triggered when the coordinates of the viewbox, represented by the `viewBoxX` and `viewBoxY` signals, are set to values outside the allowable range. The `viewBoxX` coordinate must be between 0 and 640, and the `viewBoxY` coordinate must be between 0 and 480. If either of these coordinates is out of range, the `viewBoxOutOfRangeError` signal is activated.

4 Learning Tasks

This section contains a set of learning tasks to guide you in interacting with the graphic engine. All tasks are based on a very simple demo game named *Sea Explorer*. The tasks are designed to be incremental, meaning that each task builds upon the previous ones to gradually develop a comprehensive understanding of the system.

For each task, a link to a video is provided so you can see the expected result. Feel free to draw the sprites and background tiles differently, as the primary objective is to understand the interaction between the game logic and the graphic engine.

To work on each task, you need to uncomment the corresponding `GameLogicTask#` module in the `GameTop` module. This will enable the specific learning task you are focusing on.

Important Note: You will need to show a demo of the completed tasks to the teacher and include the solved tasks with your final deliverables.

Note on simulation

This is just a note, no need to do this as part of this task.

To run the simulation of a module, you need to run in the IntelliJ terminal the command `sbt "test:runMain <TESTER_NAME>"` where you specify the tester name you want to use. For example, to run the full Game Logic Tester (file: GameLogicTester.scala), just run the following command:

```
sbt "test:runMain GameLogicTester"
```

When the simulation is finished, you can examine the output in the terminal (which can be created using 'prints' in the tester file) and the generated waveforms file by opening it with GTKWave.

4.1 Task 0: Run the demo game

The aim of this task is to build, synthesize, and run the demo game to ensure the toolchain is functioning correctly.

To generate the Verilog code for your architecture, run the following command in the IntelliJ terminal:

```
sbt run
```

Then, synthesize and implement the design using Vivado. Follow these steps:

1. Connect the Basys 3 board to the VGA screen and to your computer.
2. Program the FPGA with the generated bitstream.
3. Observe the demo game running on the VGA screen.

By pressing the buttons, you can move the red fish in the desired direction. Please look and study the provided implementation for this task, as the subsequent tasks will be very similar.

Pay particular attention to how the registers are forwarded to the sprite X and Y positions and the `spriteFlipHorizontal` signal, and how the FSM controls

these values based on the button presses. Note how the `newFrame` signal is used to start the FSM and how the `frameUpdateDone` signal is asserted in the last FSM state.

The sprites and the background tiles used in the demo game are shown in Figure 13. Feel free to open the ROM initialization files with the `tile_generator` to familiarize yourself with the tool.

Watch the video of the expected outcome of this task by following this YouTube link.

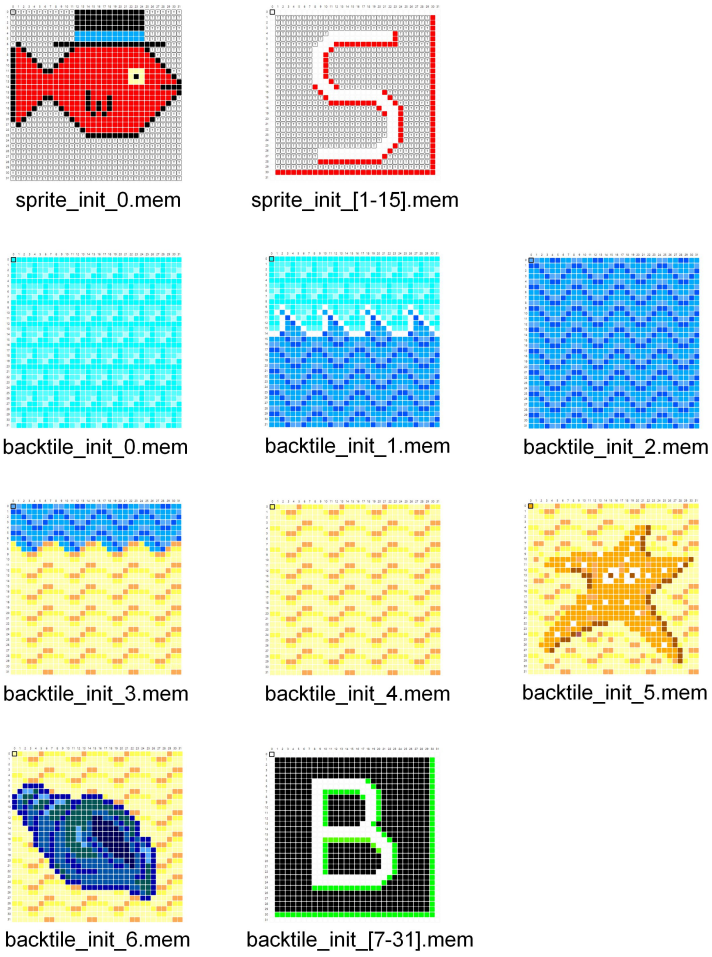

Figure 13: The sprites and the background tiles used in the demo game.

4.2 Task 1: Draw and show your own sprites

In this task, you will use the tile generator tool to draw your own sprites and make them appear on the screen. Specifically, create a sprite of another fish

(which will later move horizontally) and a sprite of a jellyfish (or a similar creature) (which will later move vertically).

First, use the tile generator tool to design your sprites. Save the generated sprite initialization files in the `memory_init` folder. Remember that sprite 0 is the existing red fish, so use different indices for your new sprites (for example 1 and 2).

Next, write the game logic to make these two new sprites visible. Place the new fish at the top left corner of the sea by setting the appropriate values for `spriteXPosition(1)` and `spriteYPosition(1)`. Similarly, place the jellyfish at the top right corner of the sea by setting `spriteXPosition(2)` and `spriteYPosition(2)`.

Ensure that the `spriteVisible` signal for these new sprites is set to `true`. This will make them appear on the screen.

Watch the video of the expected outcome of this task by following this YouTube link

4.3 Task 2: Move a sprite

In this task, you will implement the logic to move two sprites autonomously across the screen. Specifically, you will control the new fish sprite (sprite 1) to move horizontally and the jellyfish sprite (sprite 2) to move vertically.

Start from where you left in the previous task. Next, write the game logic to update the positions of these sprites autonomously using counters. For horizontal movement, create a register to act as a counter for `spriteXPosition(1)`. Similarly, create a register to act as a counter for `spriteYPosition(2)` for vertical movement.

In each frame, use the FSM to increment or decrement the `spriteXPosition(1)` counter to move the fish sprite left and right autonomously. Similarly, increment or decrement the `spriteYPosition(2)` counter to move the jellyfish sprite up and down autonomously.

Also, remember to change the spriteFlipHorizontal for the fish moving horizontally when changing the direction of counting (see Task 0).

By the end of this task, the new fish sprite should move back and forth horizontally, and the jellyfish sprite should move up and down vertically, both autonomously.

Watch the video of the expected outcome of this task by following this YouTube link

4.4 Task 3: Animate a sprite

In this task, you will implement the logic to animate a sprite. Specifically, you will animate the autonomous fish sprite (sprite 1). Optionally, you can choose to animate all the sprites.

To create the animation effect, design a second sprite that is similar to the original fish sprite but with small changes to depict motion. Save this new sprite in the `memory_init` folder with a different index.

Ensure that both sprites share the same `spriteXPosition`, `spriteYPosition`, `spriteFlipHorizontal`, and `spriteFlipVertical` values. However, only one of the sprites should be visible at a time to create the animation effect.

Use your FSM and a register to act as a counter to control the visibility of the sprites at regular intervals. Toggle the `spriteVisible` signal between the two sprites to alternate their visibility.

Here's an example logic flow:

- Initialize the counter and set `spriteVisible(1)` for the original sprite to `true`.
- In each frame, increment the counter.
- When the counter reaches a predefined value, toggle the visibility, negate (not operator) the `spriteVisible` for the original sprite and for the animated sprite.
- Reset the counter and repeat the process to create a continuous animation effect.

By the end of this task, the autonomous fish sprite should appear animated as it moves horizontally across the screen.

Watch the video of the expected outcome of this task by following this YouTube link

4.5 Task 4: Change the background

In this task, you will modify the background to add seagulls in the sky and a small volcano in the sand at the center of the screen (we will use this later to have bubbles coming out from there).

Start by using the existing background tiles for the sky and the sand surface. Open the `tile_generator` tool to modify these tiles by adding seagulls and a volcano. Save the new tiles and place them in the `memory_init` folder. Remember that `backtile_init_0` to `backtile_init_6` are already used, so use different indices for your new tiles.

Next, edit the `backBufferInit.mem` file to insert the new tile IDs (written in binary) at the appropriate memory addresses to display them in the correct positions on the screen. There is no tool for this step, so you will need to manually edit the file. Use the background layout presented earlier to determine the correct memory addresses to place the IDs of your new tiles (Tip: the line in the file equal to the address + 1).

By the end of this task, your background should feature seagulls in the sky and a volcano in the center of the sand.

Watch the video of the expected outcome of this task by following this YouTube link

4.6 Task 5: Animate the background (optional)

This task is optional (and hard) as it involves an advanced feature. Only do this task if you have time and if you plan to have an animated background (you can always do it later if you change your mind). In this task, you will animate the background to create a more dynamic scene.

To animate the background, start by designing a new tile that is slightly different from an existing background tile to depict the animation. For example, you could create animated seagulls flapping their wings. Use the `tile_generator` tool to design this new tile (same seagulls, but different wing positions). Once the new tile is designed, save it in the `memory_init` folder with a unique index, such as `backtile_init_9.mem`.

Next, you need to modify the FSM to periodically update the background memory with the new tile. This involves writing the new tile's ID to the appropriate position in the background buffer at regular intervals. The background buffer write interface is used for this purpose. The interface includes the signals `backBufferWriteData` for the data to be written (the ID of the new tile), `backBufferWriteAddress` for the address in the background buffer where the new tile ID should be written, and `backBufferWriteEnable` to enable the write operation.

Use a counter in the control FSM to trigger the write operation at regular intervals to switch between the two tiles. When the counter reaches the specified value, the FSM should set `backBufferWriteData` to the ID of the two animation tiles alternatively, set `backBufferWriteAddress` to the address of the tile you want to update, and set `backBufferWriteEnable` to `true` for one clock cycle to perform the write operation.

This periodic update will create an animated background that changes over time, enhancing the visual appeal of your game.

Watch the video of the expected outcome of this task by following this YouTube link

4.7 Task 6: Add one more moving sprite

Using the skills you learned in Task 1 and 2. create a sprite of a bubble coming out from the little volcano you created in Task 4. Make the bubble rise to the water's surface. When the bubble reaches the surface, make it disappear. Make this cycle repeat endlessly.

You will use this new bubble sprite in the next task to explore task interaction.

Watch the video of the expected outcome of this task by following this YouTube link

4.8 Task 7: Sprite interaction (optional)

This task is optional, but recommended. In this task, you will learn how to make sprites interact with each other by detecting collision.

The aim is to make the bubble disappear if the red fish (the one controlled with the buttons) touches it. Be aware that this task is quite complex.

There are two possible solutions to achieve this interaction. One approach is to compare the X and Y coordinates between the fish and the bubble sprite. If they overlap, set the visibility of the bubble to `false`. This method involves checking if the bounding boxes of the sprites intersect.

Another approach is to calculate the Euclidean distance between the fish's mouth (or more simply the fish's center) and the bubble's center. If this distance becomes less than the bubble's radius, make the bubble disappear. Note that calculating the square root is not feasible in hardware. However, this is not an issue since we can use the distance squared in the comparison instead.

Here is how you can implement these solutions:

Bounding Box Overlap:

Check if the bounding boxes of the fish and bubble overlap. Assuming that the sprites drawings are aligned in the top-left corner of the 32-by-32 tile, the bounding box overlap equation is:

```
spriteXPosition(fish) < spriteXPosition(bubble) + width(bubble)
  &&
spriteXPosition(bubble) < spriteXPosition(fish) + width(fish)
  &&
spriteYPosition(fish) < spriteYPosition(bubble) + height(bubble)
  &&
spriteYPosition(bubble) < spriteYPosition(fish) + height(fish)
```

If all these conditions are met, set `spriteVisible(bubble)` to `false`.

In general, the bounding box overlap equation between box 1 and box 2 is:

```
x1min < x2max && x2min < x1max && y1min < y2max && y2min < y1max
```

Distance Calculation:

Calculate the squared distance between the fish's mouth and the bubble's center. First, determine the coordinates of the fish's mouth based on the fish's X and Y positions and the `spriteFlipHorizontal` signal. If the fish is not flipped, the mouth's X coordinate is `spriteXPosition(fish) + mouthOffsetX`. If the fish is flipped, adjust the calculation accordingly. The Y coordinate of the mouth remains the same as `spriteYPosition(fish) + mouthOffsetX`.

The Euclidian distance (squared) is:

$$distance^2 = (x_{fish\_mouth} - x_{bubble})^2 + (y_{fish\_mouth} - y_{bubble})^2$$

If $distance^2$ is less than or equal to the bubble's radius squared, set `spriteVisible(bubble)` to `false`. Tip: The square operation can be computed using a multiplier.

Watch the video of the expected outcome of this task by following this YouTube link. Here the solution with the Euclidian distance is implemented.

4.9 Task 8: Moving the viewbox

This task is provided to you already completed. It involves an advanced feature of the engine: the scrolling background.

In this task, the viewBox is moved on top of a large background. The buttons are used to move the viewBox by controlling the `viewBoxX` and the `viewBoxY` outputs. Examine the given code, configure the board as usual, and try to move the background using the buttons. The tiles containing the letter 'B' indicate the region of the background that use a default background tile initialization content.

Watch the video of the expected outcome of this task by following this YouTube link

Important note: You do not need to use this feature in your game if you do not want to. Be aware that managing this feature can be very complex.

5 Main project: Develop your game

This section provides guidelines for creating your own arcade-style game leveraging the provided graphic engine, which is the main project of the 3-week course. You are expected to implement your game logic in the GameLogic block as you

did for the learning tasks. Of course, you can create new Chisel modules if you want to develop a hierarchical design.

We recommend that you base your design on finite state machines (FSMs). Dedicated architectures (e.g., custom designs based on counters, arithmetic pipelines, dataflow, etc.) are also welcome, but they are usually more complex to design and extend.

5.1 Game concept

Before diving into the technical details, start by designing the game itself (i.e. the gameplay). You have the option to replicate a classic arcade game or invent your own unique game concept.

Here are a few examples of classic arcade games that you might consider. **Please note that these are just for inspiration. You will need to implement a simplified version.**

- **Pong**: A simple two-player game where each player controls a paddle to hit a ball back and forth.
- **Space Invaders**: A game where the player controls a spaceship and must defend against waves of incoming aliens.
- **Pac-Man**: A maze game where the player navigates through a maze, eating pellets while avoiding ghosts.
- **Asteroids**: A space-themed game where the player controls a spaceship that must shoot and destroy asteroids while avoiding collisions.
- **Snake**: A game where the player controls a snake to eat food, growing longer with each piece consumed, while avoiding collisions with itself and the walls.
- **Arkanoid**: A brick-breaking game where the player controls a paddle to bounce a ball and break bricks.
- **Tetris**: A tile-matching game where the player must fit falling tetrominoes together to complete lines and clear them from the screen.
- **Flappy Bird**: A game where the player controls a bird to navigate through a series of pipes without hitting them.

Replicating (in a simplified manner) an existing game is fine. However, we think it is more fun to invent your own game. In any case, begin by conceptualizing your game idea, drawing inspiration from existing games while considering gameplay mechanics, characters, thematic elements, and engaging objectives.

We recommend starting with a simple yet captivating design that aligns with your skill level and project scope. Sketch out preliminary drafts to visualize

the game's structure and logic before proceeding to the design phase. In other words, have a clear vision of what you want to create (layout, characters, rules, and objectives).

5.2 Design

Transition to the design phase by breaking down your game concept into discrete components, emphasizing the implementation of game logic using FSMs or specialized hardware architectures. Develop a detailed design outlining the behavior and the interactions of each game element. Define the background layout and sprite designs, considering factors such as movement patterns, collision detection, scorekeeping, etc.

- **Background design**: Design a static (or scrolling) background that complements your game. Determine the tile layout and how the background will interact with game elements. Decide whether your background will be purely decorative or if it will include interactive components that influence gameplay. Please note that implementing a scrolling background using the background view box can be very hard.
- **Sprite design**: Create detailed designs for the game sprites, including the main character, enemies (if any), and other interactive objects. Consider the sprite's size, appearance, and animation frames. Define how each sprite will behave, including movement patterns, collision properties, and interactions with other sprites and the background.
- **Interaction logic**: Outline the logic for interactions between sprites and between sprites and background. Specify how collisions will be detected and handled, how the game state will change in response to these interactions, and how the FSMs will manage these processes.
- **User input handling**: Design the user input system, using the 5 buttons available on the Basys3 board. Define how player inputs will be mapped to in-game actions and how these inputs will influence the game state and sprite behaviors.
- **FSM design**: Develop FSMs for managing game states, sprite behaviors, and user interactions. Clearly define the states, transitions, and conditions for each FSM, ensuring a cohesive and efficient game logic implementation.

5.3 Implementation

Execute your design by translating it into the Chisel hardware description languages, focusing on implementing the defined FSMs and hardware components. We suggest to follow these tips to ensure a smooth and effective implementation process:

- **Start simple**: Begin by implementing basic functionality to test some core behaviors. Avoid writing a full FSM initially; instead, focus on simple, individual components. For example, start by moving a sprite across the screen or detecting a button press.
- **Iterative development**: Use code from the learning tasks as inspiration. Implement each new feature incrementally. After completing a small part of your game logic, test it on the Basys3 board to ensure it works as expected. This iterative approach will help you identify and fix issues early, making the development process more manageable.
- **Test each feature**: As you add each new feature to your game, thoroughly test it on the hardware or by writing Chisel tests for the `GameLogic` module. This could include testing sprite movements, collision detection, or input handling. Make sure each component works correctly before moving on to the next one.
- **Incremental FSM implementation**: Once you have tested basic behaviors, gradually build up your FSMs to manage more complex game states and interactions. Ensure each FSM state and transition is functioning properly through rigorous testing on the Basys3 board.
- **Optimize and refine**: After implementing the core features, optimize your code for performance and resource usage (if needed). Refine your implementation based on the feedback from testing, ensuring smooth and responsive gameplay.

5.4 Enhancements

With the core gameplay mechanics implemented, explore opportunities for enhancing the gaming experience within the hardware constraints. This part of the development aims to let you experiment with more advanced features (especially if you have extra time). Experiment with more complex features such as dynamic background effects, advanced sprite animations, interactive elements, and complex physics.

- **Dynamic background effects**: Introduce scrolling backgrounds, parallax effects, or dynamic environmental changes to enhance the visual appeal and complexity of your game.
- **Advanced sprite animations**: Enhance your sprites with more complex animations, such as multi-frame movements and transformations (flip horizontal or vertical).
- **Interactive elements**: Add new gameplay mechanics, such as power-ups, obstacles, or interactive objects that respond to player actions.
- **Complex physics**: Add physics aspect to the sprite movement. Please note that doing physics calculations in hardware can be complex and generate large and slow architectures. Usually, only addition, subtraction,

multiplication, and division by a power of 2 (shifting) are easy to implement.

6 Time management, report requirement, and evaluation criteria

In this section, you will find a time management plan, the report requirement, and the evaluation criteria.

6.1 Time management

In order for you to check if you are on track or behind, we provide the following indicative timeplan table with reference to the 15 course days.

Days	**What to work on**
1	Form groups and register on DTU-Learn
1	Setup development environment and carry out Task 0
1 - 3	Work on the learning tasks
4	Brainstorm and finalize game concepts
4 - 5	Begin initial design drafts
5	Start implementing basic functionality in Chisel
5	Test and verify initial implementations on the board
6 - 10	Incrementally add and test new features
11 - 12	Begin integrating enhancements
13	Finalize all core features and enhancements
11 - 13	Conduct thorough testing (and debugging)
11 - 14	Write and finalize the project report
14	Prepare the final demonstration video
14	Submit the final report and source files
15	Present your game and demonstrate its functionality

6.2 Report requirements

The short report should not be longer than 4 pages (everything included) and a mandatory template (for font size and style) for the report is available in DTU-Learn. Please write "directly-to-the-point", without re-explaining and presenting the basic ideas already presented in this document. Use the report to explain the most important ideas about your game, its design, its implementation, and the enhancements. Do not include the full code in the report, but you can include some code snippets (or point to the code) if these are very relevant to explain certain aspects of the implementation.

In the following, you can find the expected structure of the report (also present in the template).

- **Contributions**: Clearly state what each team member contributed to the project. **This section is crucial for evaluating individual contributions and ensuring fair grading.**
- **Introduction**: Briefly introduce your game, including its concept and main features.
- **Design overview**: Summarize the key aspects of your game design, including the background layout, sprite designs, and game logic.
- **Implementation details**: Describe the main features of your implementation, focusing on how you translated your design into Chisel code. Highlight any significant challenges and how you addressed them.
- **Enhancements**: Discuss any additional features or improvements you implemented, explaining their purpose and how they were integrated into the game.

6.3 Evaluation criteria

Your project will be evaluated based on the following criteria:

- **Learning tasks**: Completion and understanding of the initial learning tasks, demonstrating a grasp of the basic concepts and techniques in Chisel and hardware design.
- **Design**: The clarity, completeness, and feasibility of your game design. This includes the background layout, sprite designs, and the overall architecture of your game logic.
- **Implementation**: The accuracy and efficiency of your implementation in Chisel. This includes adherence to the initial design, successful testing of individual components, and the final integration.
- **Enhancements**: Creativity and effectiveness of any additional features or improvements beyond the core functionality.
- **Report**: The quality of your project report, including clear explanations of your game and its implementation, and a detailed account of each team member's contributions.