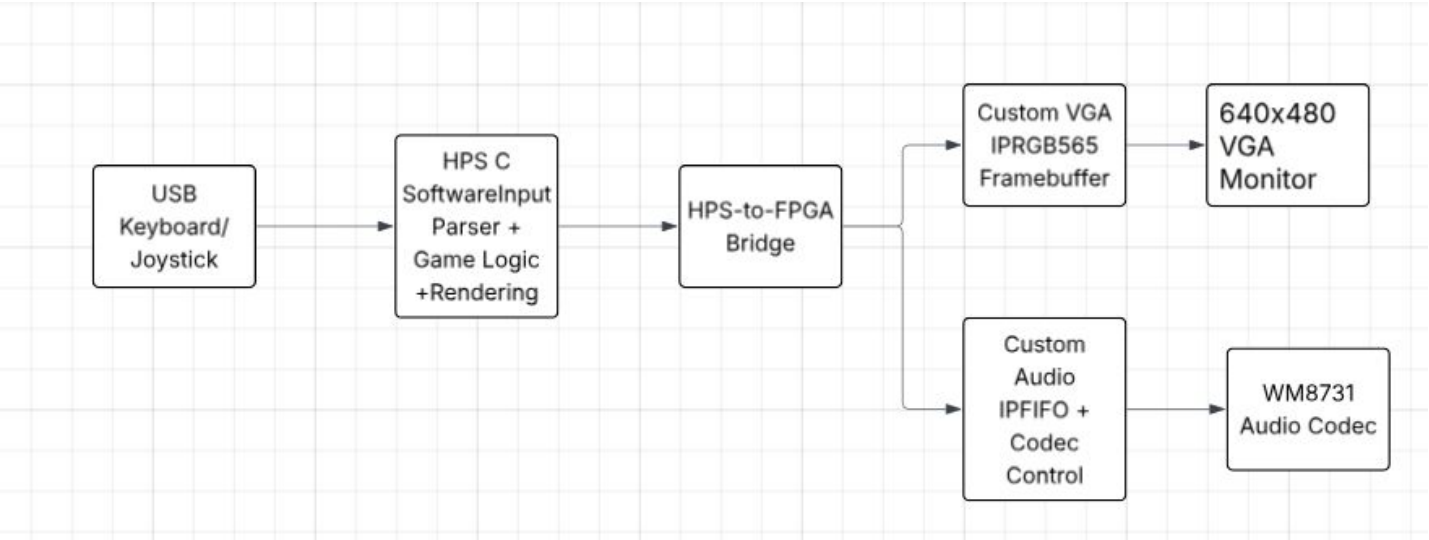
Embedded System Presentation

Street Fighter Game

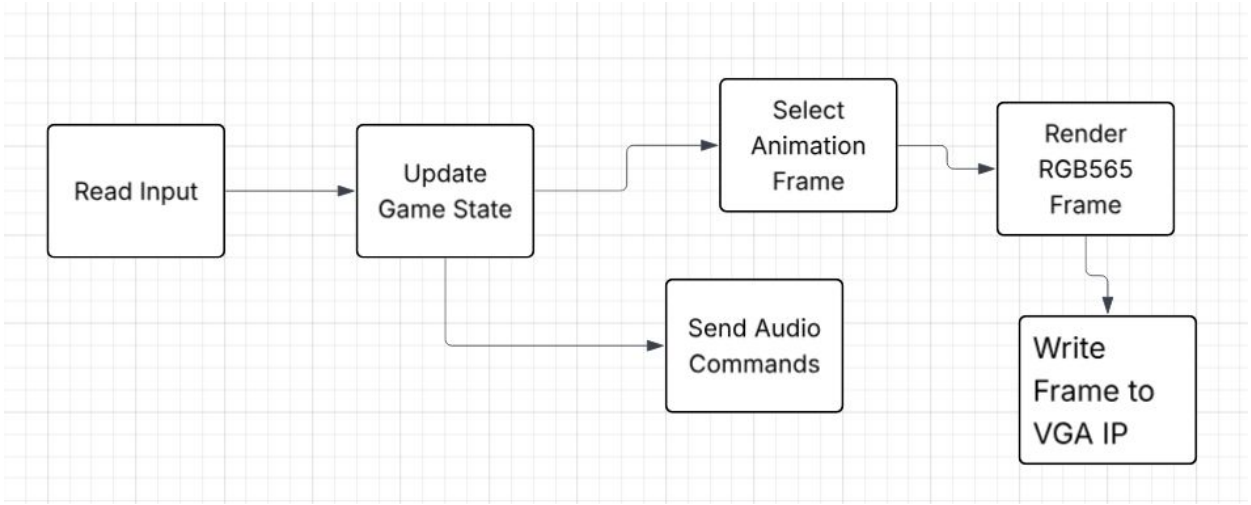
High-Level Block Diagram



Hardware-Software Register Map

Module	Qsys Offset	Linux Physical Address	Purpose
Audio IP	0x0000	0xFF200000	WM8731 audio FIFO
VGA IP	0x40000	0xFF240000	RGB565 framebuffer
Bridge reset	N/A	0xFFD0501C	enable HPS-FPGA bridge

Software Pipeline

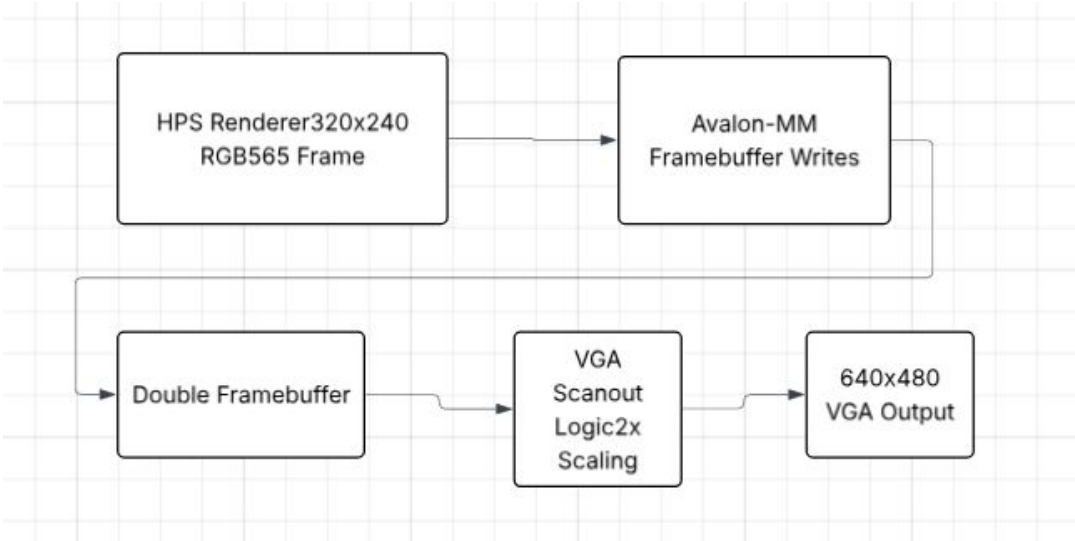


VGA Register Map

Offset	Meaning
0	control/status
1	width = 320
2	height = 240
3	stride = 640
31	ident = "VPGA"
1024+	framebuffer words

- 32-bit framebuffer word
  - [31:16] pixel 1 RGB565
  - [15:0] pixel 0 RGB565

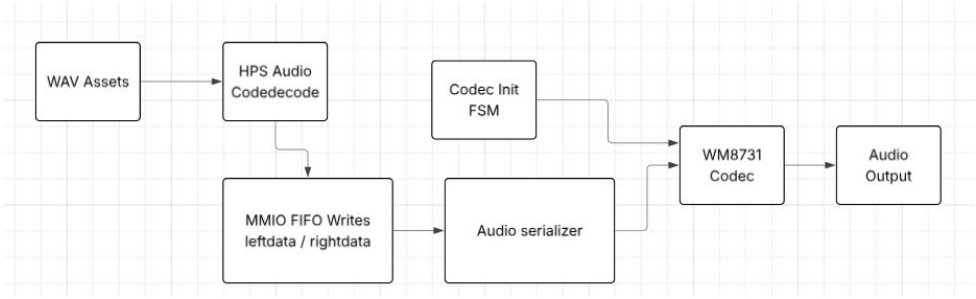
VGA Hardware Block Diagram



From PPM to RGB565

- Read PPM Image File
- Read Header (P6, width, height, max color)
- Parse Pixel Data, R G B values per pixel
- Scale / Store
- Output RGB565

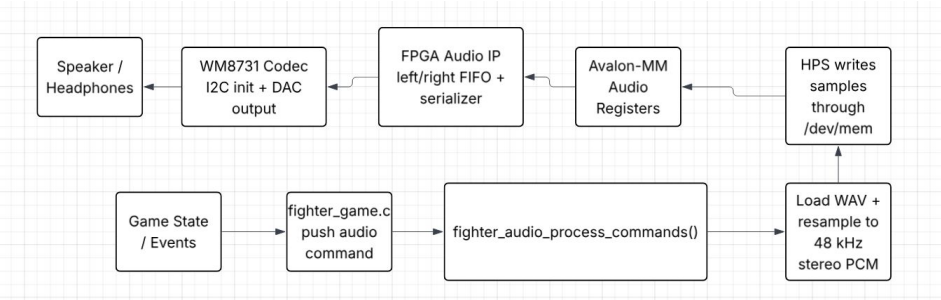
Audio Hardware Block Diagram



Implemented Audio MMIO Contract

Offset	Name	R/W	Description
0x00	`control`	RW	Write `bit2` to clear the read FIFO, write `bit3` to clear the write FIFO; read returns status bits
0x04	`fifospace`	R	`[31:24]` is left-channel available write space, `[23:16]` is right-channel available write space
0x08	`leftdata`	W	Left-channel sample word; software writes `int16 « 16`
0x0C	`rightdata`	W	Right-channel sample word; software writes `int16 « 16`

Audio Overview




```
735
736         state->audio_regs[FIGHTER_AUDIO_MMIO_REG_LEFTDATA] =
737             ((uint32_t)(uint16_t)left_sample) << 16;
738         state->audio_regs[FIGHTER_AUDIO_MMIO_REG_RIGHTDATA] =
739             ((uint32_t)(uint16_t)right_sample) << 16;
740         state->current_frame++;
741     }
742 }
```

Gamepad Input

- The gamepads are handled on the HPS side through the Linux input event interface
- Each player uses one /dev/input/eventX device (normally event0 and event1)
- The device is opened with O_RDONLY | O_NONBLOCK, so pooling the gamepad will not block the game loop
- Every frame the software drains all pending input events and updates the current button state
- The gamepad layer converts raw Linux events into the existing game input structure `fighter_player_result_t`

Gamepad Input

```c
// 打开 Linux input event 设备
int fighter_gamepad_open(fighter_gamepad_t *gamepad, const char *device_path) {

  if (!gamepad || !device_path || device_path[0] == '\0') {
    return -1;
  }

  // 如果之前打开过设备，先关闭并清空状态，避免 fd 泄漏
  fighter_gamepad_close(gamepad);

  // 非阻塞打开，每帧 poll 时没有新事件不会卡住游戏主循环
  gamepad->fd = open(device_path, O_RDONLY | O_NONBLOCK);

  if (gamepad->fd < 0) {
    return -1;
  }

  // 标记该手柄已连接
  gamepad->connected = 1;
  //保存路径，主要用于启动日志和调试
  snprintf(gamepad->device_path, sizeof(gamepad->device_path), "%s", device_path);

  // 打开新设备后，当前按钮状态从全松开开始
  memset(&gamepad->current_buttons, 0, sizeof(gamepad->current_buttons));

  // 上一帧按钮状态也清空，避免刚启动时产生假边沿
  memset(&gamepad->previous_buttons, 0, sizeof(gamepad->previous_buttons));

  return 0;
}
```

```c
static void fighter_gamepad_handle_key(fighter_gamepad_t *gamepad,unsigned short code,int value) {
  if (!gamepad) {
    return;
  }

  //EV_KEY value=1按下，value=0松开,这里非 0 都按按下处理
  pressed = value != 0;
  //code 是 Linux input-event-codes.h里的BTN_*宏
  switch (code) {
    // A 键上报 BTN_THUMB,用作普通攻击键
    case BTN_THUMB:
      gamepad->current_buttons.attack = pressed;
      break;
    //B键上报 BTN_THUMB2,用作防御键
    case BTN_THUMB2:
      gamepad->current_buttons.guard = pressed;
      break;
    //X 键上报 BTN_TRIGGER，用作发波快捷键
    case BTN_TRIGGER:
      gamepad->current_buttons.fireball = pressed;
      break;
    //Y 键上报 BTN_TOP，用作升龙拳快捷键
    case BTN_TOP:
      gamepad->current_buttons.dragon_punch = pressed;
      break;
    //SELECT 上报 BTN_BASE3，用作退出/返回菜单
    case BTN_BASE3:
      gamepad->current_buttons.exit_game = pressed;
      break;
    // START 上报 BTN_BASE4，用作菜单确认/结算重开
    case BTN_BASE4:
      gamepad->current_buttons.start = pressed;
      break;
    //左肩键上报 BTN_TOP2，作为防御备用键
    case BTN_TOP2:
      gamepad->current_buttons.guard = pressed;
      break;
    //右肩键上报 BTN_PINKIE，作为攻击备用键
    case BTN_PINKIE:
      gamepad->current_buttons.attack = pressed;
      break;

    default:
      break;
  }
}
```

```
root@de1-soc:~/sw# ./test_gamepad /dev/input/event0
Reading input from: /dev/input/event0
Press buttons or D-pad on the controller.
Use Ctrl+C to stop.

type=3 (EV_ABS), code=1 (ABS_Y), value=0
type=3 (EV_ABS), code=1 (ABS_Y), value=127
type=3 (EV_ABS), code=1 (ABS_Y), value=255
type=3 (EV_ABS), code=1 (ABS_Y), value=127
type=3 (EV_ABS), code=0 (ABS_X), value=0
type=3 (EV_ABS), code=0 (ABS_X), value=127
type=3 (EV_ABS), code=0 (ABS_X), value=255
type=3 (EV_ABS), code=0 (ABS_X), value=127
type=4 (EV_MSC), code=4, value=589825
type=1 (EV_KEY), code=288 (BTN_TRIGGER), value=1  [PRESS]
type=4 (EV_MSC), code=4, value=589825
type=1 (EV_KEY), code=288 (BTN_TRIGGER), value=0  [RELEASE]
type=4 (EV_MSC), code=4, value=589828
type=1 (EV_KEY), code=291 (BTN_TOP), value=1  [PRESS]
type=4 (EV_MSC), code=4, value=589828
type=1 (EV_KEY), code=291 (BTN_TOP), value=0  [RELEASE]
type=4 (EV_MSC), code=4, value=589826
type=1 (EV_KEY), code=289 (BTN_THUMB), value=1  [PRESS]
type=4 (EV_MSC), code=4, value=589826
type=1 (EV_KEY), code=289 (BTN_THUMB), value=0  [RELEASE]
type=4 (EV_MSC), code=4, value=589827
type=1 (EV_KEY), code=290 (BTN_THUMB2), value=1  [PRESS]
type=4 (EV_MSC), code=4, value=589827
type=1 (EV_KEY), code=290 (BTN_THUMB2), value=0  [RELEASE]
type=4 (EV_MSC), code=4, value=589829
type=1 (EV_KEY), code=292 (BTN_TOP2), value=1  [PRESS]
type=4 (EV_MSC), code=4, value=589829
type=1 (EV_KEY), code=292 (BTN_TOP2), value=0  [RELEASE]
type=4 (EV_MSC), code=4, value=589830
type=1 (EV_KEY), code=293 (BTN_PINKIE), value=1  [PRESS]
type=4 (EV_MSC), code=4, value=589830
type=1 (EV_KEY), code=293 (BTN_PINKIE), value=0  [RELEASE]
type=4 (EV_MSC), code=4, value=589833
type=1 (EV_KEY), code=296 (BTN_BASE3), value=1  [PRESS]
type=4 (EV_MSC), code=4, value=589833
type=1 (EV_KEY), code=296 (BTN_BASE3), value=0  [RELEASE]
type=4 (EV_MSC), code=4, value=589834
type=1 (EV_KEY), code=297 (BTN_BASE4), value=1  [PRESS]
type=4 (EV_MSC), code=4, value=589834
```

type=1 (EV_KEY), code=297 (BTN_BASE4), value=0 [RELEASE]

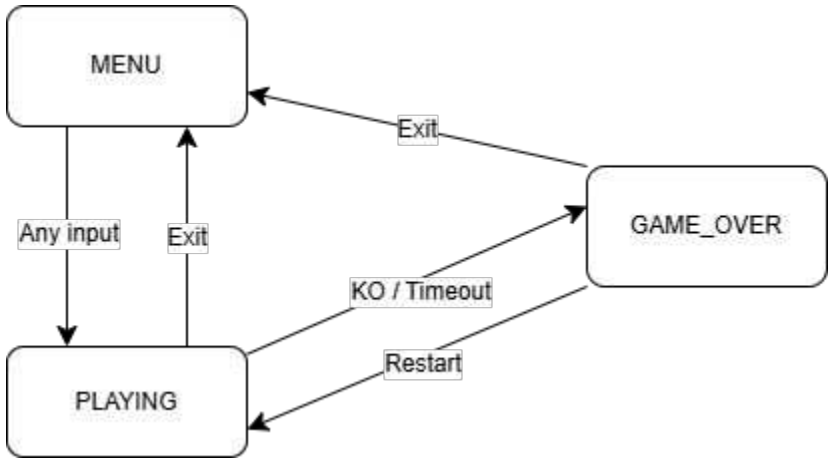
Gamepad Input

```c
static int fighter_gamepad_drain_events(fighter_gamepad_t *gamepad) {

  if (!gamepad || gamepad->fd < 0) {
    return -1;
  }

  //非阻塞fd下循环read，直到暂时没有更多事件
  while (1) {
    //Linux input 的原始事件结构，包含 type，code，value
    struct input_event event;
    //从 /dev/input/eventX 读取一个完整事件
    ssize_t bytes_read = read(gamepad->fd, &event, sizeof(event));

    // bytes_read < 0 表示 read 没成功
    if (bytes_read < 0) {
      // 非阻塞模式下，EAGAIN/EWOULDBLOCK 表示事件已经读完
      if (errno == EAGAIN || errno == EWOULDBLOCK) {
        return 0;
      }
      //EINTR 表示被信号打断，重试即可
      if (errno == EINTR) {
        continue;
      }

      return -1;
    }
    //正常情况下 read 应该刚好返回一个struct input_event 大小
    if (bytes_read != (ssize_t)sizeof(event)) {
      return -1;
    }

    //返回EV_ABS用于处理方向
    if (event.type == EV_ABS) {
      fighter_gamepad_handle_abs(gamepad, event.code, event.value);
    //返回EV_KEY 用于处理 A/B/X/Y/START/SELECT/肩键
    } else if (event.type == EV_KEY) {
      fighter_gamepad_handle_key(gamepad, event.code, event.value);
    }
  }
}
```

Game Logic Overview

Frame Based State Update:

- The game logic runs once per frame through fighter_game_tick()
- The game has 3 top-level states: MENU, PLAYING and GAME_OVER
- Each frame, the game
  - Clears temporary combat/audio outputs
  - Consumes each player's input
  - Update movement and attack phases
  - Resolve collision, hits, blocks, projectiles,KO, nd timeout
  - Outputs visual state and audio commands



Game Logic Overview

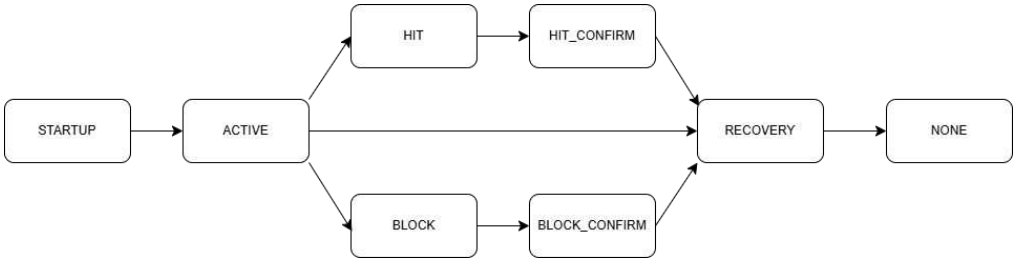
Playing State Update Order :

- Check exit request
- Update player facing
- Update P1 and P2 movement / jump / attack phase
- Resolve body overlap
- Resolve melee attacks: hit / block / trade
- Update fireball projectiles
- Update visual states
- Check KO from HP
- Decrement timer and handle timeout

player positions and attack phases are updated before hit detection, and visual/audio outputs are updated after combat results.

Combat Logic:

Attack phase flow:



- Round ends when:
  - a player HP reaches 0 → KO
  - both HP reach 0 → double KO
  - timer reaches 0 → higher HP wins, or draw

Game Logic Overview

Priority Order in Visual State Selection

- Keep VICTORY if already in victory state
- If HP <= 0, show KO
- If hurt_visual_frames > 0, show HIT
- If block_stun_frames > 0, show BLOCK_STUN
- If attack_phase != NONE, show ATTACK
- If airborne, show JUMP
- If guarding while crouching, show CROUCH_GUARD
- If guarding, show GUARD
- If crouch is held, show CROUCH
- If exactly one horizontal direction is held, show WALK
- Otherwise show IDLE