Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais
PCS 3732 – Laboratório de Processadores
Professor Bruno Abrantes Basseto
Professor Marco Túlio Carvalho de Andrade

Kernel Panic no HDMI

Relatório final

Nome Completo	**NºUSP**
Bruno Maia de Oliveira Duarte	12551481
Guilherme Castelo Branco de Brito	12552520
Renato Naves Fleury	11805269

Introdução

O projeto do grupo tem por objeto implementar um *driver* específico para a saída HDMI da Raspberry Pi, especificamente o modelo 0W. O *driver* em questão permite a execução do clássico jogo de Atari Pong[1], onde 2 jogadores podem movimentar suas respectivas barras para cima ou para baixo a fim de evitar que a bola ultrapasse a barreira e atinja o seu lado.

Para a implementação do jogo, além de ser preciso manipular a saída HDMI, era necessário existir alguma forma de comunicação entre os jogadores e a placa, uma vez que eles serão responsáveis por tomar ações que afetarão a interface e funcionamento do jogo.

Com isso, ao longo do desenvolvimento, o grupo utilizou alguns periféricos da Raspberry Pi 0W que permitem o cumprimento dos requisitos de projeto. Os principais periféricos trabalhados foram o canal 1 da *Mailbox 0*, a Mini UART e o *Core Timer*.

Por fim, com as integrações realizadas, o grupo se concentrou em aprimorar a jogabilidade para os usuários e em implementar a lógica da física do jogo de maneira fiel. A física do jogo é bastante simples, quando a bola colide com as paredes superior ou inferior, essa sofre uma reflexão total, ou seja, tem o ângulo de saída igual ao de entrada. Já as colisões entre a bola e as barras têm o seguinte funcionamento: O ângulo de saída da bola dependerá do ponto de contato que ela teve com a barra, quanto mais próximo das extremidades for o contato, maior será o ângulo de saída. A seguir, segue uma representação da explicação anterior:

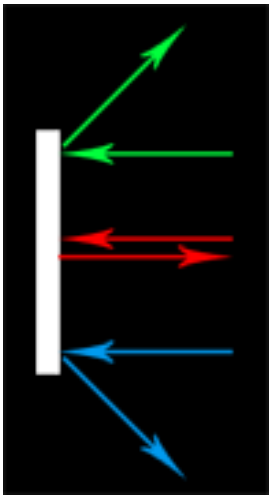

Imagem 1: Representação da lógica de colisão entre a bola e as barras dos jogadores

Com isso, o projeto do *driver* HDMI Pong foi implementado e os resultados finais serão mostrados posteriormente.

Fundamentação teórica

Durante o projeto, conforme já mencionado, foram usados 3 periféricos em especial, o canal 1 da *Mailbox 0,* a Mini UART e o *Core Timer*. Com isso, antes de partir para a integração feita no projeto, detalha-se o funcionamento geral de cada periférico.

Mailbox 0 - Canal 1

O *Mailbox*[2] é um periférico usado para a troca de informações entre a CPU e a GPU da Raspberry Pi que possui diversos canais de mensagens com diferentes propósitos de comunicação. Especificamente, o canal 0 da *Mailbox* é utilizado com o único intuito de alocar um *framebuffer* para a CPU. Embora tal método de se alocar um *framebuffer* seja depreciada, e possa se fazer isso usando o canal 8 da *Mailbox 0* também, isso só é verdade para versões mais recentes da Raspberry Pi, como o grupo trabalhou com a versão 0W, adotou-se tal abordagem para alocar o *framebuffer.*

A troca de mensagem na *Mailbox* se dá pela escrita e leitura, tanto da CPU quanto da GPU, em endereços de memória, para isso, ocorre a sincronização desse processo através de um registrador denominado *status*. O bit 31 deste registrador indica que a *Mailbox* encontra-se cheia, ou seja, a CPU deve esperar que tal bit seja zerado antes de fazer uma escrita na *Mailbox.* Já o bit 30 indica que a *Mailbox* está vazia, assim, a CPU deve esperar que tal bit seja zerado para poder fazer uma leitura e obter a resposta da GPU.

O formato da mensagem enviada da CPU para a GPU deve ter 32 bits, estruturados da seguinte forma:

- 31-4 (28 mais significativos): Mensagem a ser enviada para a GPU;
- 3-0 (4 menos significativos): Número do canal da *Mailbox.*

A escrita na *Mailbox,* por parte da CPU, é feita em um local específico, denominado de registrador *write*, enquanto a leitura, pela CPU, se dá em um registrador chamado *read*. Com isso, chega-se nas seguintes funções para manipulação da *Mailbox:*

```
/*
* Read from a specific channel in mailbox 0.
 *
 * Params: channel - the channel to read from.
 * Returns: the response from the mailbox channel
 *
  *  TODO:  presently  the  response  is  just  stripped  of  the  channel
information, but
 * it may be more appropriate to right shift the response.
 */
uint32_t readMB0 (MB0_CHANNEL channel)
{
  uint32_t response = 0;
  while (1) {
    while (*MB0_STATUS & MAIL_EMPTY) {};
    response = *MB0_READ;
    if ((response & CHANNEL_MASK) == channel) {
      return (response & (~CHANNEL_MASK));
    }
  }
}
```

```
/*
 * Write to a specific channel in mailbox 0.
 *
 * Params: data     - the data to write
 *         channel  - the mailbox channel to write to
 * Preconditions: data must be left shifted four bits or a 16-byte
aligned address
 * Returns: none
 */
void writeMB0 (uint32_t data, MB0_CHANNEL channel)
{
  data |= L2_BYPASS_MASK | channel;
  while (*MB0_STATUS & MAIL_FULL) {};
  *MB0_WRITE = data;
}
```

Funções para manipulação da *Mailbox*

Mais detalhadamente sobre o funcionamento do canal 1[3], a mensagem escrita no registrador deve conter o endereço para um endereço na memória que contenha as seguintes informações:

1. Quantidade de pixels a ser usada na largura do *display* físico;
2. Quantidade de pixels a ser usada na altura do *display* físico;
3. Quantidade de pixels a ser usada na largura do *framebuffer* virtual;
4. Quantidade de pixels a ser usada na altura do *framebuffer* virtual;
5. *Pitch,* número de *bytes* entre cada linha do *framebuffer*, tal espaço deve ter 0s na requisição e é preenchido na resposta;
6. *Depth* (Profundidade), a quantidade de bits a serem usados para representar a cor/brilho de cada pixel;
7. *Offset* horizontal para o *framebuffer* virtual, em pixels;
8. *Offset* vertical para o *framebuffer* virtual, em pixels;
9. Ponteiro para o *framebuffer* alocado, tal espaço deve ter 0s na requisição e é preenchido na resposta;
10. Tamanho do *framebuffer* alocado, tal espaço deve ter 0s na requisição e é preenchido na resposta.

Vale ressaltar, ainda, que todos os parâmetros acima devem ocupar 32 bits na memória, e que o endereço dessa estrutura deve estar alinhado em 16 bits. A partir disso, chega-se na seguinte representação em código da estrutura:

```
typedef struct {
  uint32_t width;   // frame width in pixels
  uint32_t height;  // frame height in pixels
  uint32_t vWidth;  // virtual width in pixels
```

```
  uint32_t vHeight; // virtual hieght in pixels
  uint32_t pitch;   // pitch (bytes per row)
  uint32_t depth;   // pixel bit depth
  uint32_t xOffset; // horizontal offset in pixels
  uint32_t yOffset; // vertical offset in pixels
  uint32_t fb;      // pointer to the framebuffer to write to
  uint32_t fbSize;  // size of the framebuffer, ^, in bytes
}   fb_info_t   __attribute__((aligned(16)));   //   see   comment   above
regarding 16-byte alignment
```

Estrutura utilizada para configuração do *framebuffer*

Por fim, segue o código utilizado para alocar um *framebuffer* no projeto, o qual utiliza todos os conceitos vistos até aqui:

```
/*
 * Initializes a framebuffer struct, sends to the VC via mailbox 0,
then awaits the response
 *
 * Params: fbInfo - pointer to the framebuffer information structure
 *         width  - the requested frame width
 *         height - the requested frame height
 *         depth  - the requested bit depth
 * Side effects: the VC overwrites the fbInfo structure with the
provided framebuffer information
 * Returns: none
 */
void initializeFrameBuffer (fb_info_t * fbInfo, uint32_t width,
uint32_t height, uint32_t depth)
{
  fbInfo->width = width;
  fbInfo->height = height;
  fbInfo->vWidth = width;
  fbInfo->vHeight = height;
  fbInfo->depth = depth;
  fbInfo->yOffset = 0;
  fbInfo->xOffset = 0;
  fbInfo->fbSize = 0;
  fbInfo->pitch = 0;
  fbInfo->fb = 0;

  writeMB0((uint32_t) fbInfo, FRAMEBUFFER);
  readMB0(FRAMEBUFFER);
}
```

Código para alocação de *framebuffer*

Mini UART

A Mini UART faz parte do chamado “periférico auxiliar” da Raspberry Pi, o qual implementa outras 2 interfaces síncronas chamadas de SPI, além da Mini UART que é uma interface assíncrona.

Para a utilização da Mini UART, existe um conjunto de registradores importantes[4], sendo os principais, aqueles usados no projeto, mostrados adiante:

- Registrador IO: Usado para transmissão/recepção de dados durante a comunicação, o que é feito por meio de acesso aos *buffers* de transmissão ou recepção, respectivamente. Vale ressaltar que cada *buffer* possui 8 *bytes* de espaço;
- Registrador IER: Utilizado para habilitação de interrupções de transmissão ou recepção. Isso é controlado pelo bit 7 do registrador LCR, DLAB, quando tal bit é zerado, as interrupções são habilitadas;
- Registrador LCR: Usado para configuração da comunicação, sendo possível estabelecer a quantidade de bits de dados utilizada, por exemplo. Usado para indicar que se trabalha com dados de 8 bits, o que é feito habilitando o bit menos significativo;
- Registrador CNTL: Registrador que dá direito a utilização de funcionalidades diferenciadas com relação a UART padrão. No projeto, foi usado para habilitar o TX e RX apenas, escrevendo 1 nos dois primeiros bits do registrador;
- Registrador BAUD: Usado para controlar o *baud rate* da comunicação. Colocou-se o valor de 270 neste registrador, o que significa um *baud rate* de 115200;
- Registrador STAT: Registrador de *status* usado para facilitar o uso da Mini UART, fornecendo informações úteis de forma centralizada. Tem-se os seguintes bits de interesse para o projeto:
    - 0: Quando está habilitado, indica que o *buffer* de recepção contém um símbolo a ser lido;
    - 1: Quando está habilitado, indica que o *buffer* de transmissão ainda possui espaço disponível.
- Registrador AUX ENB: Registrador do conjunto do periférico auxiliar que indica qual interface será utilizada. Para escolher a Mini UART, deve-se habilitar o bit menos significativo.

Com todas as configurações relacionadas ao funcionamento da Mini UART explicadas anteriormente, resta, apenas, especificar a configuração dos pinos a serem usados para comunicação serial, no caso, GPIO 14 e GPIO 15. Para isso, basta configurar tais pinos para o modo ALT5 e habilitar a configuração *pull-up*[5]. Portanto, chega-se no seguinte código de configuração e uso para transmissão/recepção da Mini UART:

```
void mini_uart_init(void) {
  // configura GPIO14 e GPIO15 como função ALT5 (mini UART)
  uint32_t sel = GPIO_REG(gpfsel[1]);
  sel = (sel & (~(7<<12))) | (2<<12);
  sel = (sel & (~(7<<15))) | (2<<15);
  GPIO_REG(gpfsel[1]) = sel;

  // habilita pull-ups em GPIO14 e GPIO15
```

```
  GPIO_REG(gppud) = 0;
  delay(150);
  GPIO_REG(gppudclk[0]) = (1 << 14) | (1 << 15);
  delay(150);
  GPIO_REG(gppudclk[0]) = 0;

  // habilita perifério e configura
  AUX_REG(enables) = 0x00000001; // habilita a mini UART
  MU_REG(cntl) = 0;
  MU_REG(ier) = 0;
  MU_REG(lcr) = 3; // 8 bits
  MU_REG(mcr) = 0;
  MU_REG(baud) = 270; // para 115200 bps em 250 MHz
  MU_REG(cntl) = 3; // habilita TX e RX
}

void mini_uart_putc(uint8_t c) {
  while((MU_REG(stat) & 0x02) == 0) ; // não há espaço
  MU_REG(io) = c;
}

void mini_uart_puts(char *s) {
  while(*s) {
    mini_uart_putc(*s);
    if(*s == '\n')
      mini_uart_putc('\r');
    s++;
  }
}

uint8_t mini_uart_getc(void) {
  while((MU_REG(stat) & 0x01) == 0); // não há dados a ler
  return MU_REG(io);
}

static uint8_t char_table[16] = {
  '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D',
'E', 'F'
};

void mini_uart_put_dword(uint32_t value) {
  mini_uart_putc( char_table[(value >> 28) & 0xF]);
  mini_uart_putc( char_table[(value >> 24) & 0xF]);
```

```
  mini_uart_putc( char_table[(value >> 20) & 0xF]);
  mini_uart_putc( char_table[(value >> 16) & 0xF]);
  mini_uart_putc( char_table[(value >> 12) & 0xF]);
  mini_uart_putc( char_table[(value >>  8) & 0xF]);
  mini_uart_putc( char_table[(value >>  4) & 0xF]);
  mini_uart_putc( char_table[(value >>  0) & 0xF]);
}

void mini_uart_putc_hex(uint8_t value) {
  mini_uart_putc( char_table[(value >>  4) & 0xF]);
  mini_uart_putc( char_table[(value >>  0) & 0xF]);
}

void mini_uart_debug_puts(char *s, uint32_t value) {
  mini_uart_puts(s);
  mini_uart_put_dword(value);
  mini_uart_puts("\n");
}
```

Configuração da Mini UART e funções para manipulação da transmissão e recepção

Por fim, como um ponto importante do projeto consistia na recepção de ações por parte dos jogadores, vale ressaltar como que a recepção serial foi utilizada no projeto. Trabalhou-se com o tratamento da recepção via interrupção, assim, além das configurações mencionadas, é preciso habilitar a interrupção do periférico auxiliar, o que é feito ativando o bit 29 do grupo 1 de interrupções da Raspberry Pi.

Com a interrupção ativada, fez-se a identificação da interrupção de Mini UART dentro da função de tratamento de interrupções do projeto, por meio da checagem dos bits 29 do registrador *pending_basic* do IRQ e do bit 0 no registrador de interrupção do periférico auxiliar. Com isso, foi possível diferenciar esta interrupção da interrupção de relógio, a ser comentada posteriormente. Seguem os trechos destinados a tais tópicos:

```
/*
* Interrupção da mini UART
*/
if(bit_is_set(IRQ_REG(pending_1), 29)) { // verifica interrução AUX
         if(bit_is_set(AUX_REG(irq), 0)) { // verifica interrupção mini
UART
            mini_uart_puts("        Uart_interrupt\n");
            uint8_t reg_io = MU_REG(io) & 0xFF;
            check_action(reg_io);
        };
        // ... demais interrupcoes sao dos spi_1 e spi_2
    }

/*
```

```
* Configura interrupção da mini UART
*/
void enable_mini_uart_irq(void) {
    MU_REG(ier) = __bit(3) |
                  __bit(2) |
                       //__bit(1) |          // habilita interrupcao na
transmissão
                  __bit(0);         // habilita interrupcao na recepção

    IRQ_REG(enable_1) = __bit(29); // habilita interrupção da mini UART
}
```

Configuração da interrupção de Mini UART e o seu respectivo tratamento

Core Timer

O *Core Timer*[6] pode ser usado na Raspberry Pi para a execução de tarefas rotineiras, uma vez que cada núcleo possui seu próprio *timer* independente, o qual é capaz de gerar uma interrupção periódica. O *timer* é um contador *free-running* com frequência fixa de 1 MHz, cujo funcionamento é: carrega-se um valor no contador, o qual decrementa tal valor a cada ciclo de relógio e, quando se chega no 0, é gerada uma interrupção. Após a interrupção, restaura-se o contador com o valor de recarga e o processo continua.

Para a utilização dele, existem alguns registradores a serem manipulados[7], sendo os principais ressaltados a seguir:

- Registrador LOAD: Valor de recarga carregado no *timer*;
- Registrador ACK: Registrador onde se deve escrever para reconhecer o tratamento da interrupção de relógio;
- Registrador CONTROL: Usado para ativar/desativar diversas funcionalidades do *timer*, no projeto, foi trabalhado com os seguintes bits deste registrador:
    - 1: Quando habilitado, indica que o *timer* terá 23 bits;
    - 5: Usado para habilitar a interrupção;
    - 7: Usado para habilitar o *timer*;
    - 9: Usado para habilitar o contador *free-running*;

Além da configuração do *timer*, é preciso habilitar a interrupção do *Core Timer* no registrador de interrupção IRQ, de maneira análoga ao que foi feito para a Mini UART, o que é feito pelo bit 0 do registrador *enable_basic* de IRQ. Com isso, chega-se na seguinte função de configuração do *timer*:

```
/*
* Configura interrupção do timer.
*/
void enable_timer_irq(void) {
     TIMER_REG(load) = 1000;                // 1MHz / 10000 = 100 Hz ->
Definir intervalo de tempo
      TIMER_REG(control) = __bit(9)           // habilita free-running
counter
```

```
                        | __bit(7)        // habilita timer
                        | __bit(5)        // habilita interrupção
                        | __bit(1);       // timer de 23 bits

    IRQ_REG(enable_basic) = __bit(0);    // habilita interrupção básica
0 (timer)
}
```

Configuração do *Core Timer*

Finalmente, resta especificar o tratamento da interrupção de relógio, procedimento que é feito, novamente, de maneira similar ao feito para a Mini UART. Na função de tratamento de interrupções, é adicionada uma verificação para a interrupção de relógio, o que é feito checando se o bit 0 do registrador IRQ *pending_basic* está ativo. Assim, reunindo o código desta interrupção com o código para a interrupção de Mini UART, tem-se a seguinte função de tratamento de interrupções:

```
/**
 * Chamado pelo serviço de interrupção irq
 * Deve ser a interrupção do core timer ou de UART
 */
void trata_irq(void) {
    /*
     * Interrupção do timer a cada xx ms -> definir intervalo de tempo
entre interrupcao de timer
    */
    //mini_uart_puts("Trata_irq\n");
    if(bit_is_set(IRQ_REG(pending_basic), 0)) {
          TIMER_REG(ack) = 1;                          // reconhece a
interrupção
        tick ++;
        if(tick > 5) {
            //mini_uart_puts("Tick_interrupt\n");
            update_interface();
            tick = 0;
        }
        return;
    }

    /*
    * Interrupção da mini UART
    */
    if(bit_is_set(IRQ_REG(pending_1), 29)) { // verifica interrução AUX
         if(bit_is_set(AUX_REG(irq), 0)) { // verifica interrupção mini
UART
            mini_uart_puts("        Uart_interrupt\n");
```

```
            uint8_t reg_io = MU_REG(io) & 0xFF;
            check_action(reg_io);
        };
        // ... demais interrupcoes sao dos spi_1 e spi_2
    }
}
```

Função de tratamento das interrupções do projeto

Desenvolvimento do projeto

Configurações iniciais

Primeiramente, fez-se o código do funcionamento básico do sistema, o qual deve fazer as seguintes funções básicas:

- Zerar as variáveis não inicializadas presentes no segmento BSS (feito em C dentro da função “kernel_main”, e não na função “_start”);
- Implementar o vetor de interrupções e colocá-lo nos endereços iniciais da memória. Um detalhe importante acerca deste passo é que foi feito tratamento apenas para 2 casos:
    - Reset: Neste caso, faz-se funções iniciais do sistema, como a configuração dos *stack pointers* do modo de Supervisor e do modo IRQ; inicialização do vetor de interrupções; e salto para a função “kernel_main”;
    - IRQ: Fez-se o tratamento para interrupções normais, salvando o endereço de retorno no código na pilha, saltando para a função de tratamento de interrupções trata_irq, e retornando para o ponto do código interrompido.
- Funções básicas para habilitar ou desabilitar as interrupções no sistema;
- Função de *delay,* para atrasar o código pela quantidade de ciclos de relógio especificada como parâmetro.

Assim, chegou-se ao seguinte código básico em *assembler* do sistema, coloca-se também o arquivo de *linker* utilizado:

```
/*****************************************************************************
*********
 * A simple ARM assembly entry point that sets the stack pointer to the
stack
 * top, defined in the linker script, and then jumps to the C entry
point,
 * the kernel main function.

*****************************************************************************
*******/

.section .init
```

```
.globl _start, delay

_start:
  /*
    * Vetor de interrupções
    * Deve ser copiado no enderço 0x0000
    */
    ldr pc, _reset
    ldr pc, _undef
    ldr pc, _swi
    ldr pc, _iabort
    ldr pc, _dabort
    nop
    ldr pc, _irq
    ldr pc, _fiq

    _reset:     .word   reset
    _undef:     .word   panic
    _swi:       .word   panic
    _iabort:    .word   panic
    _dabort:    .word   panic
    _irq:       .word   irq
    _fiq:       .word   panic

/*
* Vetor de reset: início do programa aqui.
*/
reset:
    /*
    * configura os stacks pointers do sistema
    */
    mov r0, #0xd2           // Modo IRQ
    msr cpsr, r0
    ldr sp, =_stack_irq

    mov r0, #0xd3           // Modo SVC, interrupções mascaradas
    msr cpsr, r0
    ldr sp,=_stack_top_

      // Continua executando no modo supervisor (SVC), interrupções
desabilitadas

    /*
```

```
    * Move o vetor de interrupções para o endereço 0
    */
    ldr r0, =_start
    mov r1, #0x0000
    ldmia r0!, {r2,r3,r4,r5,r6,r7,r8,r9}
    stmia r1!, {r2,r3,r4,r5,r6,r7,r8,r9}
    ldmia r0!, {r2,r3,r4,r5,r6,r7,r8,r9}
    stmia r1!, {r2,r3,r4,r5,r6,r7,r8,r9}

    b kernel_main

/**
* Trava o processador (panic)
*/
panic:
   wfe
   b panic

/*
 * Habilita interrupcoes
 */
.global enable_irq
enable_irq:
   mrs r0, cpsr
   bic r0, r0, #(1 << 7)
   msr cpsr_c, r0
   mov pc, lr
/*
 * Desabilita interrupcoes
 */
.global disable_irq
disable_irq:
   mrs r0, cpsr
   orr r0, r0, #(1 << 7)
   msr cpsr_c, r0
   mov pc, lr

/*
 * Tratamento das interrupções.
 * Essa interrupcao e chamada pelo relogio
 * Quando entra nessa rotina o sistema já está no modo SVR
 */
irq:
```

```
    sub lr, lr, #4
    // Salva o endereco de retorno
    push {lr}
    bl trata_irq  // função em C
    pop {lr}
    movs pc, lr

delay:
    subs r0, r0, #1
    bne delay
    mov pc, lr
```

Código em *assembler* com funções básicas do sistema

```
SECTIONS {
  .init 0x8000 : {
      *(.init)
  }

  .text : {
      *(.text)
      . = ALIGN(8);
      *(.rodata)
  }

  .data : {
      *(.data)
      . = ALIGN(8);
      _edata = .;
  }

  _bss_start_ = .;
  .bss : {
      *(.bss)
      . = ALIGN(8);
  }
  _bss_end_ = .;

  . = ALIGN(8);
  heap_addr = .;
  . = . + 32K;
  _stack_top_ = .;
  . = . + 4K;
  _stack_irq = .;
```

```
  _end_ = .;
}
```

Linker script usado para o projeto

Outro ponto importante a ser ressaltado é quanto ao esquema de montagem preciso para execução e teste do projeto. Com isso, fez-se um diagrama com as conexões a serem feitas para a montagem do projeto:

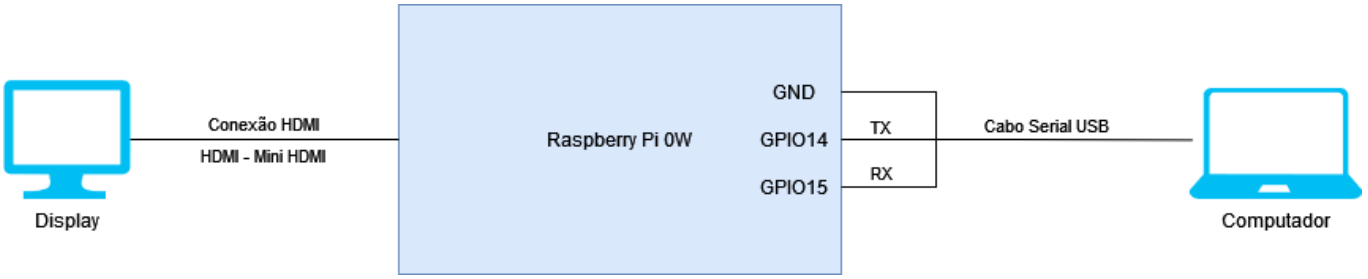

Imagem 2: Esquema de montagem do projeto

Feito isso, pode-se definir a função “kernel_main” do projeto, a qual deve fazer a inicialização da configuração dos periféricos e do *framebuffer*. Além disso, inicializa-se a interface com os elementos do jogo, a bola, as barras e os delimitadores da tela. Como parâmetros escolhidos para a alocação do *framebuffer*, tem-se:

- Altura (HEIGHT): 1920 pixels;
- Largura (WIDTH): 1080 pixels;
- Profundidade (*depth)*: 32 bits.

No fim, tem-se um *loop* eterno, visto que o funcionamento do jogo se dá pelo tratamento de interrupções. Portanto, segue a implementação da função:

```
extern int _bss_start_;
extern int _bss_end_;



/*
 *  Initializse the c environment (partially for now).
 *  Initialize static variables to zero
 */
void init ()
{
  int * bss = &_bss_start_;
  int * bssEnd = &_bss_end_;

  while (bss < bssEnd) {
    *bss = 0;
    bss++;
```

```
  }
}



/*
 * kernel main function, entry point for C code after the ARM Assembly
init
 */
void kernel_main ()
{
  init();

  mini_uart_init();

  fb_info_t fbInfo;
  initializeFrameBuffer(&fbInfo, WIDTH, HEIGHT, 32);
  interface_init(&fbInfo);

  irq_init();

  while (1) {};
}
```

Função kernel_main do projeto

Além disso, vale destacar a função utilizada para alterar um pixel específico na tela, uma vez que ela será a base para todas as funções desenvolvidas para a interface do jogo. Tal função recebe como parâmetros a posição X e Y do pixel na tela; um ponteiro para o *framebuffer* utilizado; e a cor do pixel com 32 bits, pois essa foi a profundidade escolhida para o jogo. Assim, segue a implementação da função:

```
/*
 *  Draw an individual pixel to a framebuffer
 *
 *   Params: fbInfo   - pointer to the framebuffer info struct defining
the
 *                      framebuffer to write to
 *           x       - x coordinate of the pixel
 *           y       - y coordinate of the pixel
 *           color   - the color to write
 *
 */
void fbPutPixel (fb_info_t * fbInfo, uint32_t x, uint32_t y, uint32_t
color)
{
  /* get the byte offset of the pixel and write in the color */
```

```
  uint32_t offset = (y * fbInfo->pitch) + (x << 2);
  uint32_t * pixel = (uint32_t *) (fbInfo->fb + offset);
  *pixel = color;
}

uint32_t pack_color(uint8_t r, uint8_t g, uint8_t b, uint8_t a) {
   return (r << 24) | (g << 16) | (b << 8) | a;
}
```

Função para alterar o estado de um pixel na tela

Implementação do jogo

Pode-se destacar, primeiramente, a ideia básica para viabilizar a implementação do Pong, a qual consiste em como será feito o movimento da bola e das barras, além de implementar a física do jogo de maneira adequada. Para atualizar a interface, utilizar-se-á a interrupção de relógio, cuja função já foi mostrada anteriormente. Analisando esse código, percebe-se que a cada 5 interrupções de relógio, frequência de 200 Hz, executa-se a função denominada “update_interface”, cuja função é atualizar o estado da bola e da posição das barras. Além disso, a cada interrupção de Mini UART, pega-se o valor recebido do *buffer* de recepção e faz-se uma validação de qual ação deve ser tomada a partir do caractere recebido. Ademais, é importante explicitar as constantes usadas para a interface do projeto:

- Tamanho do lado da bola: 20 pixels;
- Velocidade máxima da bola para uma direção (X ou Y): 5 pixels/atualização de interface - lógica será explicada posteriormente ;
- Largura da barra: 28 pixels;
- Altura da barra: 216 pixels;
- Deslocamento da barra quando é apertada uma tecla de movimento: 50 pixels;
- Tecla para movimentar a barra esquerda para cima: w;
- Tecla para movimentar a barra esquerda para baixo: s;
- Tecla para movimentar a barra direita para cima: o;
- Tecla para movimentar a barra direita para baixo: l.

Com a ideia geral de como o funcionamento do jogo se dará, viu-se a necessidade de criar estruturas de dados que pudessem armazenar o estado atual do jogo, o que é de extrema importância para a lógica de atualização da interface, bem como para as validações da física do Pong, a serem mostradas adiante. Dessa forma:

```
typedef struct {
    uint32_t x_position;
    uint32_t y_position;
    int      delta_x;
    int      delta_y;
} ball_state_t;

typedef struct {
```

```
    uint32_t y_position;
    int      delta_y;
} bar_state_t;

typedef struct {
    ball_state_t ball_state;
    bar_state_t left_bar_state;
    bar_state_t right_bar_state;
} states_t;
```

Estruturas de dados para gerenciamento do jogo

Para a bola, armazena-se a posição X e Y dela no momento atual do jogo, além disso, é guardado o delta X e o delta Y a serem somados com a posição atual na próxima atualização de interface. Tendo isso em vista, percebe-se que o movimento e a velocidade da bola são determinados por esses deltas, uma vez que eles ditam como a posição da bola irá variar no espaço com o decorrer do tempo. Para a barra, tem-se a mesma ideia, com a eliminação da posição X, uma vez que essa é constante para as barras.

Outro ponto importante acerca da interface, é quanto ao ponto de referência utilizado para o posicionamento da bola e da barra. Como se trabalha apenas com pixels, e tanto as barras quanto as bolas possuem são compostas por diversos deles, nota-se que existem diversas possibilidades para fazer o referenciamento dos objetos. Para que se tenha uma convenção, adotou-se como ponto de referência a ser usado no gerenciamento de estados o ponto superior esquerdo dos elementos, ou seja, o pixel mais à esquerda e mais acima do elemento visual.

Feitas as definições iniciais do jogo, tem-se as primeiras funções desenvolvidas, as quais têm como objetivo inicializar as estruturas de dados mostradas (funções com prefixo “init”); desenhar na saída HDMI os elementos visuais do jogo (funções com prefixo “draw” e “create”; e funções para apagar elementos visuais do jogo (funções com prefixo “delete”):

```
fb_info_t *fb;
states_t states = {0};

unsigned long long int seed = 1;  // Semente inicial (pode ser qualquer
número)
#define A 1664525                 // Constante multiplicativa
#define C 1013904223              // Constante de incremento
#define M 4294967296              // Modulo (2^32)

void debug_states(void);
void debug_ball_state(void);

unsigned long long int simple_rand() {
    // LCG: Novo valor de seed é (A * seed + C) % M
    seed = (A * seed + C) % M;
    return seed;
}
```

```
int rand_range(int lower, int upper) {
    return (simple_rand() % (upper - lower + 1)) + lower;
}

void draw_delimiters() {
    uint32_t white_color = pack_color(255, 255, 255, 255);
    for(int i = 0; i < HEIGHT; i++) {
        fbPutPixel(fb, 0, i, white_color);
        fbPutPixel(fb, WIDTH-1, i, white_color);
    }
    for(int j = 0; j < WIDTH; j ++) {
        fbPutPixel(fb, j, 0, white_color);
        fbPutPixel(fb, j, HEIGHT-1, white_color);
    }
}

void create_ball() {
    uint32_t white_color = pack_color(255, 255, 255, 255);
    for(int i = 0; i < BALL_SIDE; i++) {
        for(int j = 0; j < BALL_SIDE; j++) {
                    fbPutPixel(fb,  states.ball_state.x_position  +  j,
states.ball_state.y_position + i, white_color);
        }
    }
}

void delete_ball() {
    uint32_t black_color = pack_color(0, 0, 0, 0);
    for(int i = 0; i < BALL_SIDE; i++) {
        for(int j = 0; j < BALL_SIDE; j++) {
                    fbPutPixel(fb,  states.ball_state.x_position  +  j,
states.ball_state.y_position + i, black_color);
        }
    }
}

void init_ball(void) {
    states.ball_state.x_position = WIDTH/2 - BALL_SIDE/2;
    states.ball_state.y_position = HEIGHT/2 - BALL_SIDE/2;
    int x_value = 0;
    do {
        x_value = rand_range(-MAX_BALL_SPEED, MAX_BALL_SPEED);
```

```
    } while (x_value == 0);
    states.ball_state.delta_x = x_value;
            states.ball_state.delta_y   =   rand_range(-MAX_BALL_SPEED,
MAX_BALL_SPEED);
    create_ball();
}

void create_left_bar(void) {
    uint32_t white_color = pack_color(255, 255, 255, 255);
    for(int i = 0; i < BAR_HEIGHT; i++) {
        for(int j = 0; j < BAR_WIDTH; j ++) {
                fbPutPixel(fb, j, states.left_bar_state.y_position + i,
white_color);
        }
    }
}

void delete_left_bar(void) {
    uint32_t black_color = pack_color(0, 0, 0, 0);
    for(int i = 0; i < BAR_HEIGHT; i++) {
        for(int j = 0; j < BAR_WIDTH; j ++) {
                fbPutPixel(fb, j, states.left_bar_state.y_position + i,
black_color);
        }
    }
}

void create_right_bar(void) {
    uint32_t white_color = pack_color(255, 255, 255, 255);
    for(int i = 0; i < BAR_HEIGHT; i++) {
        for(int j = 0; j < BAR_WIDTH; j ++) {
            fbPutPixel(fb, WIDTH - j, states.right_bar_state.y_position
+ i, white_color);
        }
    }
}

void delete_right_bar(void) {
    uint32_t black_color = pack_color(0, 0, 0, 0);
    for(int i = 0; i < BAR_HEIGHT; i++) {
        for(int j = 0; j < BAR_WIDTH; j ++) {
            fbPutPixel(fb, WIDTH - j, states.right_bar_state.y_position
+ i, black_color);
```

```
        }
    }
}

void init_left_bar() {
    states.left_bar_state.y_position = HEIGHT/2 - BAR_HEIGHT/2;
    states.left_bar_state.delta_y = 0;
    create_left_bar();
};

void init_right_bar() {
    states.right_bar_state.y_position = HEIGHT/2 - BAR_HEIGHT/2;
    states.right_bar_state.delta_y = 0;
    create_right_bar();
}

void interface_init(fb_info_t *fbInfo) {
    fb = fbInfo;
    draw_delimiters();
    init_ball();
    init_left_bar();
    init_right_bar();
}
```

Funções iniciais para o Pong

Um ponto interessante a ser visto é quanto à função “init_ball”, uma vez que ela inicializa as velocidades X e Y da bola com valores aleatórios, a fim de tornar o jogo mais imprevisível no começo de rodadas.

Com a inicialização do jogo realizada, deve-se preocupar com a lógica de atualização da interface. Uma questão importantíssima a ser considerada para a atualização, é quanto à lógica de detecção de colisão entre a bola e outros elementos do jogo. Tem-se 3 casos básicos a serem analisados, em que cada condição gera um comportamento diferente para o jogo:

- Bola colide com limite inferior ou superior: Nesse caso, a bola deve apenas sofrer um reflexão, ou seja, é preciso inverter a variável delta_y da bolinha apenas;
- Bola colide com alguma barra: É necessário calcular o ponto de contato com a barra e, a partir disso, definir quais serão as velocidades X e Y da bola, controladas pelos atributos delta_x e delta_y;
- Bola passa da barra e faz-se um ponto no adversário: O efeito desse cenário é reiniciar a interface visual.

Outro ponto importante é como se dará a atualização da barra quando um jogador clicar em alguma tecla de movimento. Nesse caso, ao tratar a interrupção de Mini UART, detecta-se qual movimento deve ser feito no jogo e tal informação é armazenada na variável

delta_y da barra da esquerda ou da direita, dependendo da tecla pressionada. Com isso, quando houver a atualização da interface, atualizar-se-á a posição da barra de acordo com o valor presente no delta_y, zerando tal valor após o seu uso.

Então, segue o resto do código com a implementação dos cenários levantados, bem como a atualização da interface visual e do estado do jogo:

```
void restart_interface() {
    delete_ball();
    delete_left_bar();
    delete_right_bar();
    draw_delimiters();
    init_ball();
    init_left_bar();
    init_right_bar();
};

void update_ball(ball_state_t* ball_state) {
    delete_ball();
    ball_state->x_position += ball_state->delta_x;
    ball_state->y_position += ball_state->delta_y;
    create_ball();
}

void update_bar(bar_state_t* left_bar_state, bar_state_t*
right_bar_state) {
    if (left_bar_state->delta_y != 0) {
        if (left_bar_state->delta_y < 0) {
            if (left_bar_state->y_position < (BAR_MOVEMENT_SPEED + 1))
{
                delete_left_bar();
                left_bar_state->y_position = 0;
                create_left_bar();
                //debug_states();
                left_bar_state->delta_y = 0;
            } else {
                delete_left_bar();
                left_bar_state->y_position += left_bar_state->delta_y;
                create_left_bar();
                //debug_states();
                left_bar_state->delta_y = 0;
            }
        }

        if (left_bar_state ->delta_y > 0) {
```

```
            if (left_bar_state->y_position > HEIGHT - BAR_HEIGHT -
(BAR_MOVEMENT_SPEED + 1)) {
                delete_left_bar();
                left_bar_state->y_position = HEIGHT - BAR_HEIGHT;
                create_left_bar();
                //debug_states();
                left_bar_state->delta_y = 0;
            } else {
                delete_left_bar();
                left_bar_state->y_position += left_bar_state->delta_y;
                create_left_bar();
                //debug_states();
                left_bar_state->delta_y = 0;
            }
        }
    }

    if (right_bar_state->delta_y != 0) {
        if (right_bar_state->delta_y < 0) {
            if (right_bar_state->y_position < (BAR_MOVEMENT_SPEED + 1))
{
                delete_right_bar();
                right_bar_state->y_position = 0;
                create_right_bar();
                //debug_states();
                right_bar_state->delta_y = 0;
            } else {
                delete_right_bar();
                right_bar_state->y_position +=
right_bar_state->delta_y;
                create_right_bar();
                //debug_states();
                right_bar_state->delta_y = 0;
            }
        }

        if (right_bar_state ->delta_y > 0) {
            if (right_bar_state->y_position > HEIGHT - BAR_HEIGHT -
(BAR_MOVEMENT_SPEED + 1)) {
                delete_right_bar();
                right_bar_state->y_position = HEIGHT - BAR_HEIGHT;
                create_right_bar();
                //debug_states();
```

```
                right_bar_state->delta_y = 0;
            } else {
                delete_right_bar();
                right_bar_state->y_position +=
right_bar_state->delta_y;
                create_right_bar();
                //debug_states();
                right_bar_state->delta_y = 0;
            }
        }
    }

    draw_delimiters();
}

void check_colision(ball_state_t *ball_state, bar_state_t
*left_bar_state, bar_state_t *right_bar_state) {
    if (ball_state->x_position<BAR_WIDTH/2 ||
ball_state->x_position>(WIDTH-BAR_WIDTH/2-BALL_SIDE)) { // Colisao nos
delimitadores laterais
        restart_interface();
    }
    else if (ball_state->y_position<(MAX_BALL_SPEED + 1) ||
ball_state->y_position>(HEIGHT-BALL_SIDE-(MAX_BALL_SPEED + 1))) { //
Colisao no delimitador superior ou inferior
        ball_state->delta_y = -ball_state->delta_y;
    }
    else if (ball_state->x_position<(BAR_WIDTH+(MAX_BALL_SPEED + 1)) &&
(ball_state->y_position >= left_bar_state->y_position &&
ball_state->y_position <= (left_bar_state->y_position + BAR_HEIGHT))) {
// Colisao na barra esquerda
        mini_uart_puts("Colisão na barra da esquerda:\n");
        debug_ball_state();
        int colision_height_normalized = ((ball_state->y_position -
left_bar_state->y_position) / (BAR_HEIGHT/(2*MAX_BALL_SPEED))) -
MAX_BALL_SPEED;
        ball_state->delta_y = colision_height_normalized;
        if(colision_height_normalized < 0) colision_height_normalized =
-colision_height_normalized;
        ball_state->delta_x = (MAX_BALL_SPEED + 1) -
colision_height_normalized;
        debug_ball_state();
    }
```

```
    else if (ball_state->x_position>(WIDTH - BAR_WIDTH -
(MAX_BALL_SPEED + 1) - BALL_SIDE) && (ball_state->y_position >=
right_bar_state->y_position && ball_state->y_position <=
(right_bar_state->y_position + BAR_HEIGHT))) { // Colisao na barra
direita
        mini_uart_puts("Colisão na barra da direita:\n");
        debug_ball_state();
        int colision_height_normalized = ((ball_state->y_position -
right_bar_state->y_position) / (BAR_HEIGHT/(2*MAX_BALL_SPEED))) -
MAX_BALL_SPEED;
        ball_state->delta_y = colision_height_normalized;
        if(colision_height_normalized < 0) colision_height_normalized =
-colision_height_normalized;
        ball_state->delta_x = -((MAX_BALL_SPEED + 1) -
colision_height_normalized);
        debug_ball_state();
    }
}

void update_interface(void) {
    check_colision(&states.ball_state, &states.left_bar_state,
&states.right_bar_state);
    update_ball(&states.ball_state);
    update_bar(&states.left_bar_state, &states.right_bar_state);
}

void check_action(char action) {
    if (action == 'w') {
        states.left_bar_state.delta_y = -BAR_MOVEMENT_SPEED;
    }
    else if (action == 's') {
        states.left_bar_state.delta_y = BAR_MOVEMENT_SPEED;
    }
    else if (action == 'o') {
        states.right_bar_state.delta_y = -BAR_MOVEMENT_SPEED;
    }
    else if (action == 'l') {
        states.right_bar_state.delta_y = BAR_MOVEMENT_SPEED;
    }
}

void debug_bar_states(void) {
```

```
    mini_uart_debug_puts("Left bar y position: ",
states.left_bar_state.y_position);
    mini_uart_debug_puts("Left bar delta y: ",
states.left_bar_state.delta_y);
    mini_uart_debug_puts("Right bar state y position: ",
states.right_bar_state.y_position);
    mini_uart_debug_puts("Right bar state delta y: ",
states.right_bar_state.delta_y);
    mini_uart_puts("\n");
}

void debug_ball_state(void) {
    mini_uart_debug_puts("Ball y position: ",
states.ball_state.y_position);
    mini_uart_debug_puts("Ball delta y: ", states.ball_state.delta_y);
    mini_uart_debug_puts("Ball x position: ",
states.ball_state.x_position);
    mini_uart_debug_puts("Ball delta x: ", states.ball_state.delta_x);
    mini_uart_puts("\n");
}
```

Funções relativas à lógica de atualização do jogo

As últimas funções declaradas foram usadas para depuração ao longo do desenvolvimento, o qual foi feito através da Mini UART.

Conclusão

Com isso, com a realização do projeto foi possível abordar e aprofundar diversos pontos vistos durante a disciplina, como a utilização de periféricos da Raspberry Pi, *Core Timer*, Mini UART e *Mailbox 0*. Além disso, destaca-se o uso do canal 1 da *Mailbox 0* para a alocação do *framebuffer*, ponto não abordado em aula e que o grupo teve que utilizar dadas as especificações da placa usada.

Pode-se implementar tais conceitos de forma prática em um projeto *bare metal*, em que o resultado final chegado pelo grupo foi satisfatório, com funcionamento adequado de todas as principais funcionalidades existentes no Pong original.

Referências

[1]: https://www.ponggame.org
[2]: https://github.com/raspberrypi/firmware/wiki/Mailboxes
[3]: https://github.com/raspberrypi/firmware/wiki/Mailbox-framebuffer-interface
[4]: https://datasheets.raspberrypi.com/bcm2835/bcm2835-peripherals.pdf - Seção 2.2
[5]: https://www.wise-ware.com.br/poli/wiki/books/raspberry-pi/page/gpio
[6]: https://www.wise-ware.com.br/poli/wiki/books/raspberry-pi/page/o-core-timer

[7]: https://datasheets.raspberrypi.com/bcm2835/bcm2835-peripherals.pdf - Seção 14