

RS5 - Periférico e tratamento de interrupção



Fernando Gehm Moraes -fernando.moraes@pucrs.br

Willian AnaldoNunes -willian.nunes@edu.pucrs.br

Angelo Dal Zotto -angelo.dalzotto@edu.pucrs.br



ORGANIZAÇÃO

O que veremos sobre “periférico e tratamento de interrupção”:

1. **Hardware:** RS5 + PLIC + memória + periférico.

2. **Software:**

- Tratamento de interrupção
- “driver”
- Aplicação

3. **Validação;**

4. **Exercício.**

0 - Tratamento de Interrupções

Baseado em proposta da SiFive - Dois tipos de controladores de interrupção:

- **Controlador Local de Interrupção do Núcleo (CLIC)**
 - Gerencia interrupções de múltiplos threads de hardware no mesmo núcleo
- **Controlador de Interrupção em Nível de Plataforma (PLIC)**
 - Gerencia interrupções globais ou externas geradas por dispositivos periféricos
 - Permite atribuir prioridades configuráveis para cada interrupção

Suporte a interrupções no RS5

- **Interrupções de timer**
 - Através de um Relógio de Tempo Real (RTC)
- **Interrupções externas**
 - **Gerenciadas pelo PLIC**

O **PLIC é configurável** e oferece interface para Registradores Mapeados em Memória (MMR)

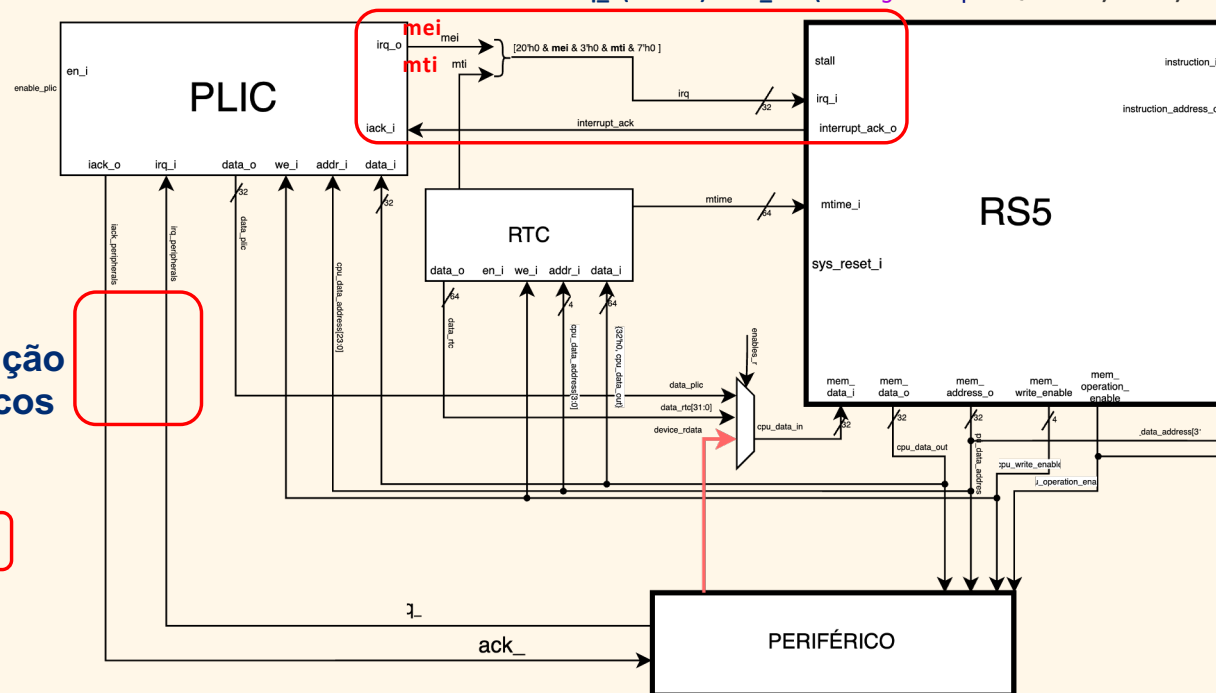
Visão Geral do Top

1 - Hardware: top + periférico

TOP:

- implementado no testbench.sv (para síntese ver a pasta *proto*, que possui uma **UART** e **BRAM**)
- 4 módulos: RS5, memória, PLIC, RTC (antes de colocarmos o periférico)
- Lógica de conexão entre os módulos

```
irq_i (32 bits) e int_ack (assign irq = {20'h0, mei, 3'h0, mti, 7'h0};)
```



Vetor de interrupção parametrizável em função do número de periféricos

```
localparam int i_cnt = 1;
```

Logica de decodificação de endereços mapeados em memória

Top - passo-a-passo para inclusão do periférico ^(1/5)

(1) Arquivo RS5/sim/testbench.sv → é no nosso top

(2) Sinais a serem inseridos no *testbench.sv* (área indicada pelo comentário **TB SIGNALS**)

- **enable_periph, enable_periph_r**: sinais que habilitam o processador a ler um dado (**_r**, de registrado, pois a CPU pode estar tratando outra interrupção)
- **data_periph**: dado gerado pelo periférico

```
////////////////////////////////////  
// PERIPH  
////////////////////////////////////
```

```
logic enable_periph, enable_periph_r;  
logic [31:0] data_periph;
```

➤ Estes sinais se aplicam a outros periféricos (o método é **geral**)

Top - passo-a-passo para inclusão do periférico (2/5)

(2) Sinais a serem inseridos no *testbench.sv* (abaixo da área indicada pelo comentário **PLIC**)

- **irq_periph**, **iack_periph**: sinais de interrupção (observar o uso do **i_cnt** → este será o periférico 1)
 - Caso a variável **iack_periph** esteja declarada para uso do **PLIC**, somente adicione **irq_periph**.

```
logic [i_cnt:1] iack_periph, irq_periph;
```

- **irq_periph** é o sinal de interrupção que sai do periférico, este deve ser enviado ao **PLIC**, que é responsável por tratar interrupções.

Portanto, na importação do **PLIC**, deve-se alterar conexão **.irq_i('0)**, para que receba **irq_periph**.

```
256 //////////////////////////////////////////////////
257 // PLIC
258 //////////////////////////////////////////////////
259
260
261 /* Bits depending on connected peripherals */
262
263 logic [i_cnt:1] iack_periph, irq_periph;
264
265 plic #(
266     .i_cnt(i_cnt)
267 ) plic1 (
268     .clk      (clk),
269     .reset_n  (reset_n),
270     .en_i     (enable_plic),
271     .we_i     (mem_write_enable),
272     .addr_i   (mem_address[23:0]),
273     .data_i   (mem_data_write),
274     .data_o   (data_plic),
275     .irq_i    (irq_periph),
276     .iack_i   (interrupt_ack),
277     .iack_o   (iack_periph),
278     .irq_o    (mei)
279 );
```


Top - passo-a-passo para inclusão do periférico (3/5)

```
always_comb begin
    if (mem_operation_enable) begin
        if (mem_address[31:28] <
4'h2) begin
            enable_ram = 1'b1;
            demais enable em 0;
        end
        else if (mem_address[31:28] <
4'h3) begin
            enable_rtc = 1'b1;
            demais enable em 0;
        end
        else if (mem_address[31:28] <
4'h8) begin
            enable_plic = 1'b1;
            demais enable em 0;
        end
        else if (mem_address[31:28] <
4'h9) begin
            enable_tb = 1'b1;
            demais enable em 0;
        end
        else begin
            enable_periph = 1'b1;
            demais enable em 0;
            demais enable em 0;
        end
    end
end
```

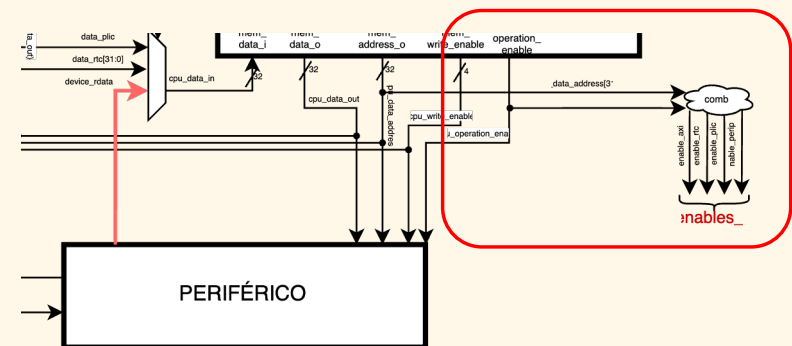
(2) Decodificação do endereço de leitura da CPU (na área indicada pelo comentário **Control**)

- Permite que a CPU receba dados da memória, rtc, plic e periféricos (4 bits mais significativos do endereço usados no controle do mux)

(256MB para RAM)

enable_periph = 1'b0;

Adicione uma instância de “else if”, para **enable_tb**, e utilize “else begin”, para **enable_periph**



Top - passo-a-passo para inclusão do periférico (4/5)

```
always_ff @(posedge clk) begin    // registra os enable - dado
```

é recebido no ciclo seguinte

```
    enable_tb_r <= enable_tb;  
    enable_rtc_r <= enable_rtc;  
    enable_plic_r <= enable_plic;  
    enable_periph_r <= enable_periph;
```

```
end
```

```
always_comb begin
```

```
    if (enable_tb_r) begin  
        mem_data_read = data_tb;
```

```
    end
```

```
    else if (enable_rtc_r) begin  
        mem_data_read = data_rtc[31:0];
```

```
    end
```

```
    else if (enable_plic_r) begin  
        mem_data_read = data_plic;
```

```
    end
```

```
    else if (enable_periph_r) begin  
        mem_data_read = data_periph;
```

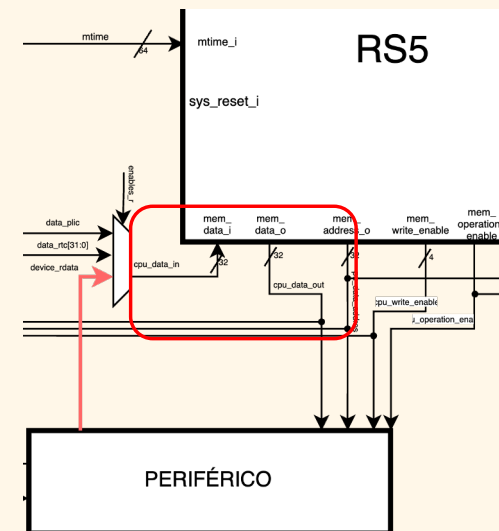
```
    end
```

```
    else begin  
        mem_data_read = data_ram;
```

```
    end
```

```
end
```

(3) Envio dos dados para a CPU



(na área indicada por comentário como **Control -> 2º e 3º Always_comb**)

Top - passo-a-passo para inclusão do periférico (5/5)

(4) Instanciar o periférico – adicionar código (abaixo dos sinais criados para código PERIPH)

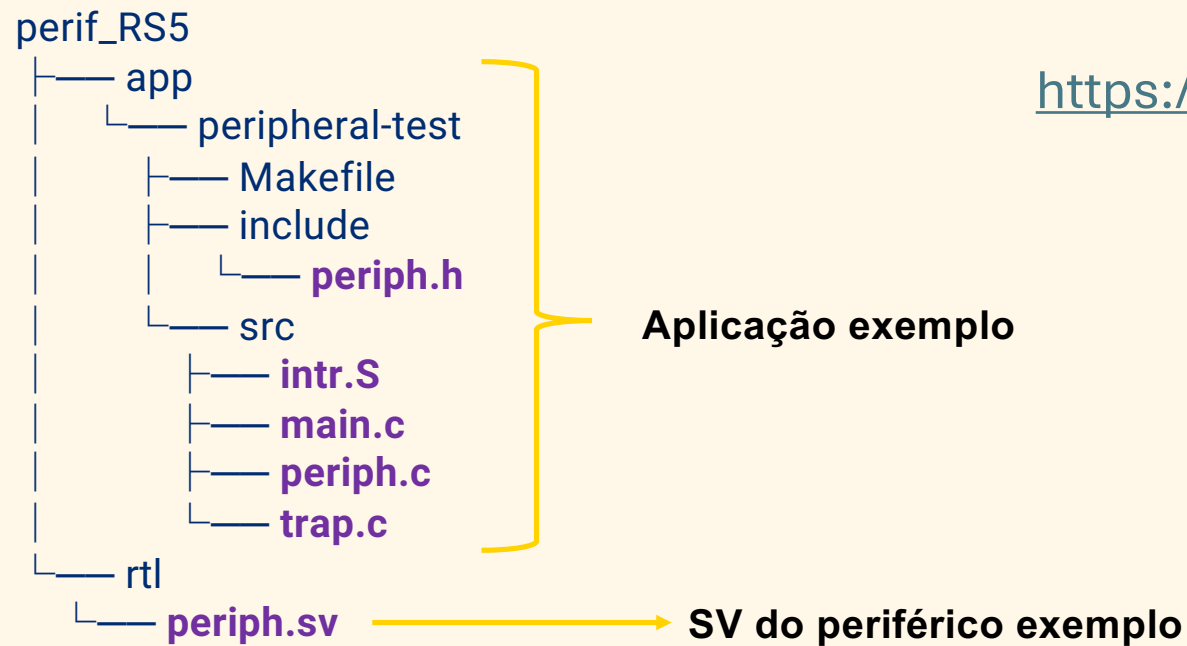
- Barramento de endereços (12 bits)
- Recepção de dados: barramento de dados da memória (mem_data_write) e write enable
- Envio de dados : quando enable_periph ativo o dado é colocado em data_periph
- Controle de interrupção (periférico 1)

```
periph periph(  
    .clk (clk),  
    .reset_n (reset_n),  
    .en_i (enable_periph),  
    .addr_i (mem_address[11:0]),  
    .we_i (mem_write_enable), // ou – reduz we_i de 4 para 1 bit  
    .data_i (mem_data_write),  
    .data_o (data_periph),  
    .irq (irq_periph[1]),  
    .iack (iack_periph[1])  
);
```

Material de Referência

Baixar de:

https://fgmoraes.github.io/perif_RS5.zip



(1) Instanciar o periférico (periph.sc)

- Interface externa

```

module periph (
    input logic          clk,
    input logic          reset_n,

    input logic          en_i,
    input logic [11:0]   we_i,

    input logic [31:0]   addr_i,
    input logic          data_i,
    output logic [31:0]  data_o,

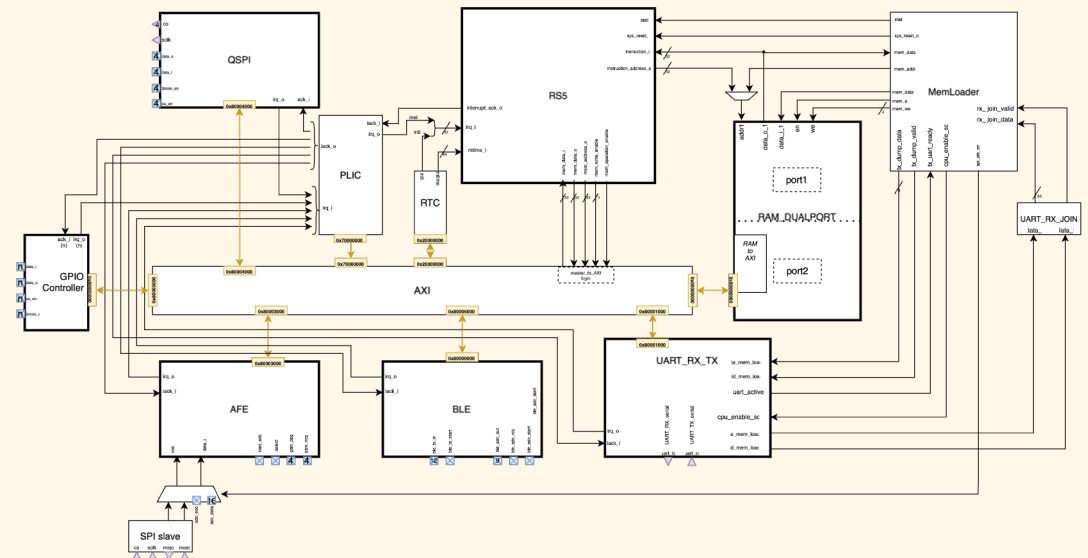
    output logic         irq,
    input logic          iack
);

```

Detalhamento do periférico (1/5)

Exemplo prático

- CPU conectada a um controlador AXI
- Periféricos conectados ao AXI e ao mundo externo

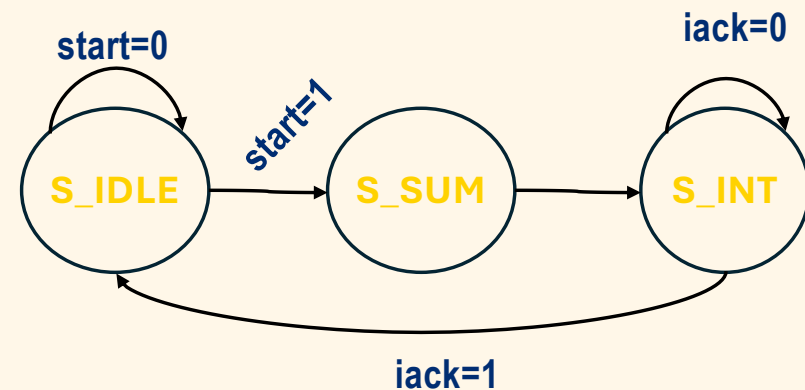


Detalhamento do periférico (2/5)

(2) Interrupção enviada ao PLIC

- Controlado por uma máquina de estados

```
always_ff @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        irq <= 1'b0;
    end
    else begin
        if (current_state == S_INT)
            begin
                irq <= 1'b1;
            end
        else begin
            irq <= 1'b0;
        end
    end
end
```



- Software** ativa o start para iniciar o processamento
- Periférico executa o processamento no estado S_Sum (em periféricos reais aqui haverá muitos estados)
- Ao final do processamento o **periférico** ativa a interrupção (S_Int)
- O **PLIC** responde com IACK para o periférico voltar ao estado S_IDLE

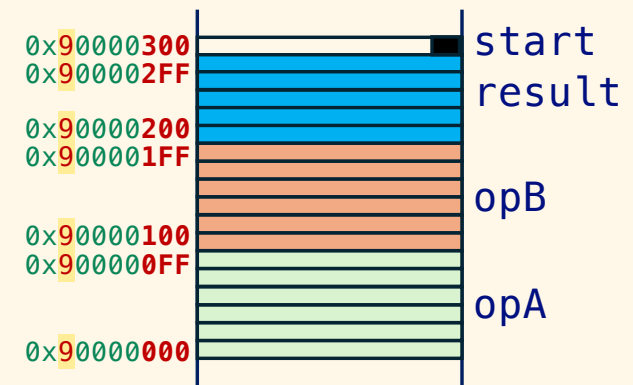
Detalhamento do periférico (3/5)

(3) Dados lidos pelo processador

- Decodificação da parte alta ($[31:28] \geq 9$) dos endereços é realizada no top
- O periférico tem 12 bits como espaço de endereçamento
- Endereços devem corresponder ao software

```
always_ff @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        data_o <= '0;
    end
    else if (en_i && we_i == '0) begin
        unique case (addr_i[11:0])
            12'h000: data_o <= opA;
            12'h100: data_o <= opB;
            12'h200: data_o <= result;
            //
            12'h300: data_o <= {31'b0, start};
            default: data_o <= '0;
        endcase
    end
end
```

na prática só lerá result



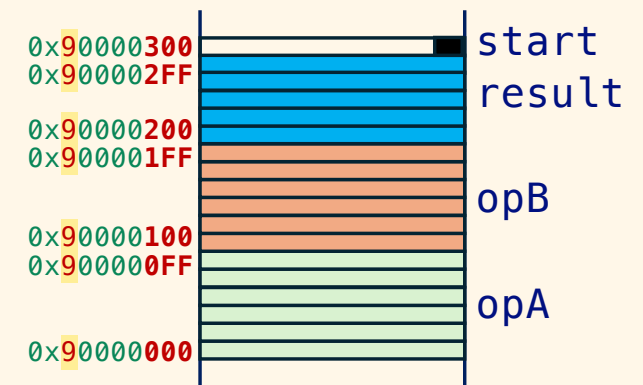
Quantos opA, opB e result o periférico pode usar?

Detalhamento do periférico (4/5)

(4) Dados escritos pelo processador

- Decodificação da parte alta ($[31:28] \geq 9$) dos endereços é realizada no top
- O periférico tem 12 bits como espaço de endereçamento
- Endereços devem corresponder ao software

```
always_ff @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        opA <= '0;
        opB <= '0;
        start <= '0;
    end
    else begin
        if (en_i && we_i != '0) begin
            unique case (addr_i[11:0])
                12'h000: opA <= data_i;
                12'h100: opB <= data_i;
                12'h300: start <= data_i[0];
                default: ;
            endcase
        end
    end
end
```



Aqui não tem **result**: porque?

Detalhamento do periférico (5/5)

(5) Tarefa executada pelo periférico

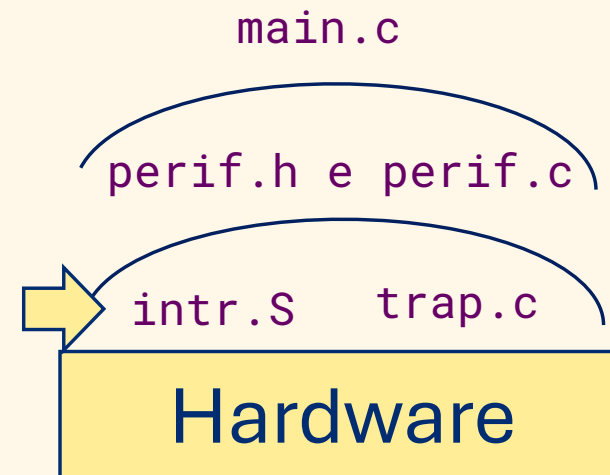
- Soma (só para demonstração de funcionalidade)

```
always_ff@ (posedge clk, negedge reset_n) begin
    if (!reset_n) begin
        result <= '0;
    end
    else begin
        if (current_state == S_SUM)
            begin
                result <= opA + opB;
            end
    end
end
```

Software para periférico e interrupção

Usaremos C + newlib → **crt0.S** da *newlib*, arquivo chamado pelo Makefile

```
peripheral-test
|-- include
|   |-- periph.h
|-- Makefile
|-- src
|   |-- intr.S
|   |-- main.c
|   |-- periph.c
|   |-- trap.c
```



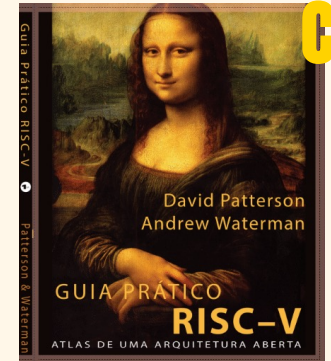
intr.S

- software em **assembly** que faz a interface com o hardware
 - config_intr**: configura o processador para realizar o tratamento da interrupção
 - trap_handler**: chamado na interrupção, e por sua vez chama o **irq_dispatcher**

Registradores CSR

Oito CSRs tratam exceção no modo de máquina:

- 1) **mstatus**, Machine Status, permissão global de interrupção
- 2) **mip**, Machine Interruption Pending, lista as interrupções atualmente pendentes
- 3) **mie**, Machine Interrupt Enable, define as interrupções o processador pode tomar e quais deve ignorar
- 4) **mcause**, Machine Exception Cause, indica qual exceção ocorreu
- 5) **mtvec**, Machine Trap Vector, endereço para qual o processador salta quando ocorre uma exceção
- 6) **mtval**, Machine Trap Value, informações adicionais sobre exceção
- 7) **mepc**, Machine Exception PC, aponta para a instrução onde a exceção ocorreu
- 8) **mscratch**, Machine Scratch, armazenamento temporário para tratadores de *traps* (pode substituir o *sp*)



pág. 107 em diante

RS5/app/peripheral-test/intr.S

intr.S

config_intr: configura o processador para realizar o tratamento da interrupção

trap_handler: chamado na interrupção, e por sua vez chama o **irq_dispatcher**

config_intr:

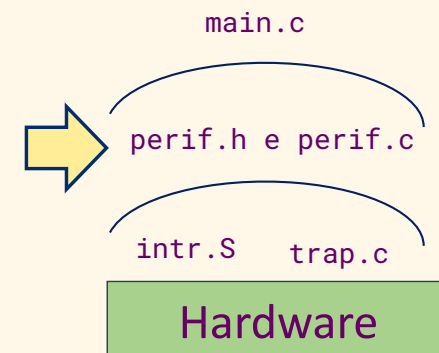
```
# Configura o tratador de interrupções
la t0, trap_handler          # Grava o endereço do rótulo trap_handler
csw mtvec, t0                # no registrador mtvec

li t0, PLIC_BASE             # Habilita Interrupções no PLIC
li t1, -1
sw t1, 0(t0)

li t2, 0x800                 # habilita Interrupções Externas Seta os
bits 11 (MEIE)               # do registrador mie
csw mie, t2

li t1, 0x08                  # # Habilita Interrupções Global Seta o
bit 3 (MIE) # pag 108        # do registrador mstatus
csw mstatus, t1
```

ret



RS5/app/peripheral-test/intr.S

intr.S

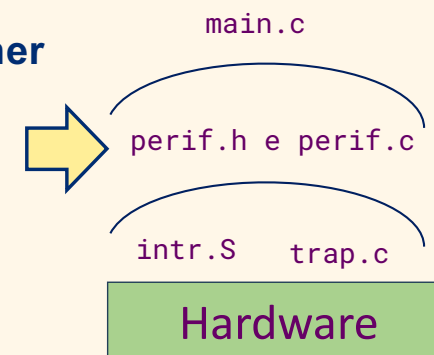
config_intr: configura o processador para realizar o tratamento da interrupção

trap_handler: chamado na interrupção, e por sua vez chama o **irq_dispatcher**

```
trap_handler:
    addi sp, sp, -64
    sw ra, 60(sp)
    ....
    sw t6, 0(sp)           // salva registradores na pilha (contexto)

    csrr a0, mcause
    bgez a0,
    except_handler         // except_handler não implementado neste exemplo
                           // verifica a causa da interrupção
    andi a0, a0, 0x3F
                           // chama dispatcher tendo em a0 o valor do mcause
    jal
    irq_dispatcher         // recupera context

    lw t6, 0(sp)
    ...
    lw ra, 60(sp)          // retornar para o mepc (PC+4 de onde a interrupção foi atendida)
    addi sp, sp, 64
    mret
```



RS5/app/peripheral-test/trap.c: implementa o irq_dispatcher

trap.c

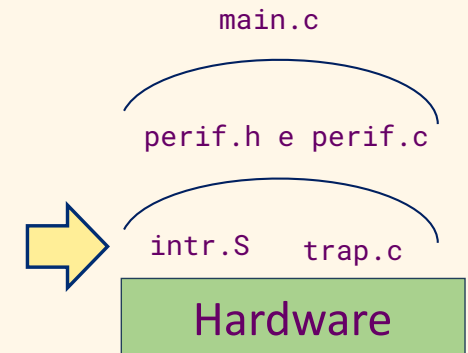
```
#define PLIC_IRQ_ID (*((volatile unsigned int*)0x00000000))
#define PLIC_ACK (*((volatile unsigned int*)0x00000004))
#define MEI 11
#define PERIPH 1
```

```
void irq_mei_dispatcher()
{
    switch(PLIC_IRQ_ID){
        case PERIPH:
            periph_handler();
            PLIC_ACK = PERIPH; // envia iack - fim da interrupção
            break;
        default:
            break;
    }
}
```

```
void irq_dispatcher(int cause)
{
    switch(cause){
        case MEI:
            irq_mei_dispatcher();
            break;
        default: // código para exceção
            break;
    }
}
```

Código de Exceção mcause[XLEN-2:0]	Descrição
1	Interrupção de software de supervisor
3	Interrupção de software da máquina
5	Interrupção de temporizador de supervisor
7	Interrupção do temporizador da máquina
9	Interrupção externa do supervisor
11	Interrupção externa da máquina

p.106



1: interrupção: trap handler

2: irq_dispatcher

3: irq_mei_dispatcher

4: periph_handler (próxima camada)

RS5/app/peripheral-test/include/periph.h

- Mapa de memória
- Protótipo das funções que comunicam-se com o periférico (driver)

```
#define OPA (*((volatile unsigned int*)0x90000000))
#define OPB (*((volatile unsigned int*)0x90000100))
#define RESULT (*((volatile unsigned int*)0x90000200))
#define START (*((volatile unsigned int*)0x90000300))
```

```
void periph_handler();
bool get_periph_ready();
void clear_periph_ready();
void set_operand_a(int val);
void set_operand_b(int val);
void periph_start();
void periph_stop();
int get_periph_data();
int read_periph_result();
```

API do
periférico

```
always_comb begin
    if (mem_operation_enable) begin
        if (mem_address[31:28] < 4'h2) begin
            enable_ram = 1'b1;
            demais enable em 0;
        end
        else if (mem_address[31:28] < 4'h3) begin
            enable_rtc = 1'b0;
            demais enable em 0;
        end
        else if (mem_address[31:28] < 4'h8) begin
            enable_plic = 1'b1;
            demais enable em 0;
        end
        else if (mem_address[31:28] < 4'h9) begin
            enable_tb = 1'b1;
            demais enable em 0;
        end
        else begin
            enable_periph = 1'b1;
            demais enable em 0;
            demais enable em 0;
        end
    end
end
```

perif.h e
perif.c

main.c

perif.h e perif.c

intr.S trap.c

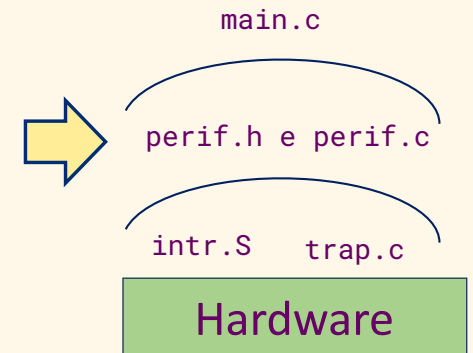
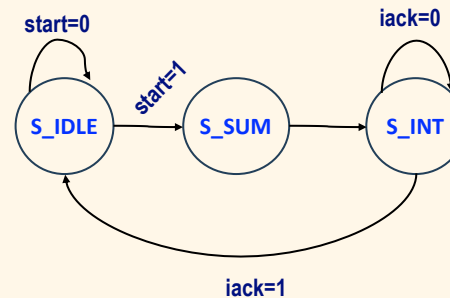
Hardware

periph.h e periph.c

- Recepção de dados quando a interrupção é ativada - **periph_handler**
- Funções de semáforo
- Leitura e escrita de registradores mapeados em memória

```
// Semaphore variables
uint32_t periph_data;
bool periph_ready = false;
```

```
void periph_handler(){
    periph_data = RESULT; // resultado
    periph_stop();        // Baixar o start
    periph_ready = true;   // Ativa semáforo
}
```



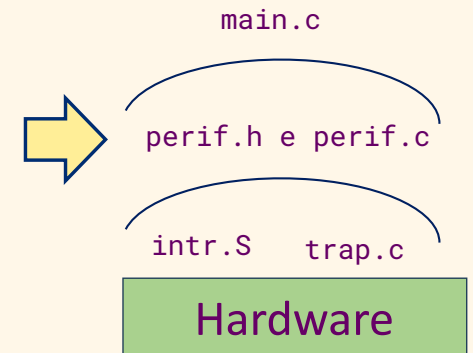
- 1: interrupção: trap handler
- 2: irq_dispatcher
- 3: irq_mei_dispatcher
- 4: periph_handler

periph.h e periph.c

- Recepção de dados quando a interrupção é ativada - `periph_handler`
- Funções de semáforo
 - o software (`main.c`) após enviar dados ao periférico, ficará aguardando que a interrupção seja tratada.
- Leitura e escrita de registradores mapeados em memória

```
// lê o estado do semáforo
bool get_periph_ready(){
    return periph_ready;
}
```

```
// desativa o semáforo
void clear_periph_ready(){
    periph_ready = 0;
}
```



- 1: interrupção: trap handler
- 2: `irq_dispatcher`
- 3: `irq_mei_dispatcher`
- 4: `periph_handler`

RS5/app/peripheral-test/periph.c

- Recepção de dados quando a interrupção é ativada - `periph_handler`
- Funções de semáforo
- Leitura e escrita de registradores mapeados em memória

```
void set_operand_a(int val){  
    OPA = val; }
```

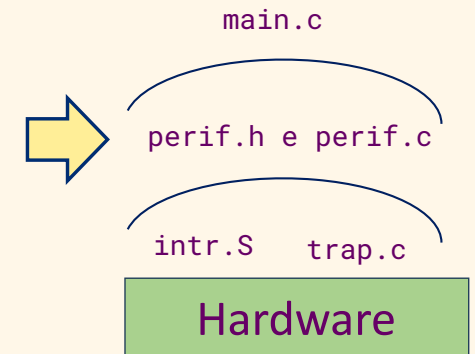
```
void set_operand_b(int val){  
    OPB = val; }
```

```
void periph_start(){  
    START = 1; }
```

```
void periph_stop(){  
    START = 0; }
```

```
int get_periph_data(){  
    return periph_data; }
```

periph.h e periph.c



1: interrupção: trap handler

2: irq_dispatcher

3: irq_mei_dispatcher

4: periph_handler

```

#include <stdio.h>
#include "../include/periph.h"

void extern config_intr(void);

int main()
{
    int periph_data = 0;
    int a, b = 0;

    config_intr(); // Habilitar interrupção

    for (int i = 0; i < 10; i++)
    {
        a = i*3;
        b = i*5;
        set_operand_a(a);
        set_operand_b(b);
        clear_periph_ready(); //Limpa o semáforo (false)
        periph_start();       // Start no periférico

        while(!get_periph_ready()); //Aguarda o semáforo ser verdadeiro (true)

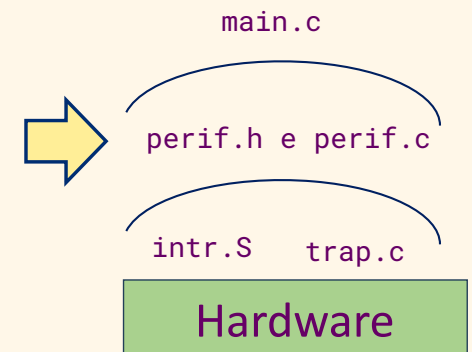
        periph_data = get_periph_data(); //Acessa o dado lido pela interrupção

        printf("a = %d , b = %d, Result: %d\n", a, b, periph_data);
    }

    return 0;
}

```

main.c



Validação

Após organizar os arquivos e pastas obtidos de https://fgmoraes.github.io/perif_RS5.zip, nos locais corretos.

(1) Compile a aplicação do periférico para obter o *bin*:

```
cd app/peripheral-test/
```

```
make clean
```

```
Cleaning up
```

```
make
```

```
Compiling src/trap.c...
```

```
Compiling src/periph.c...
```

```
Compiling src/main.c...
```

```
Compiling ../common/newlib.c...
```

```
Assembling src/intr.S...
```

```
Assembling ../common/crt0.S...
```

```
Linking peripheral-test.elf...
```

```
/soft64/util/.....with RWX permissions
```

```
Generating peripheral-test.bin...
```

```
Generating peripheral-test.lst...
```


(2) Completar o test bench conforme slides 5-8

- Lembre-se de referenciar o sinal BIN_FILE(slide aula1)

-- RUN -----

a = 0 , b = 0, Result: 0

a = 3 , b = 5, Result: 8

a = 6 , b = 10, Result: 16

a = 9 , b = 15, Result: 24

a = 12 , b = 20, Result: 32

a = 15 , b = 25, Result: 40

a = 18 , b = 30, Result: 48

a = 21 , b = 35, Result: 56

a = 24 , b = 40, Result: 64

a = 27 , b = 45, Result: 72

(3) Simulação inicial com verilator

10 iterações de soma de $(i*3)+(i*5)$

474470 END OF SIMULATION

- testbench.sv:325: Verilog \$finish

- Simulation Report: Verilator 5.024 2024-04-05

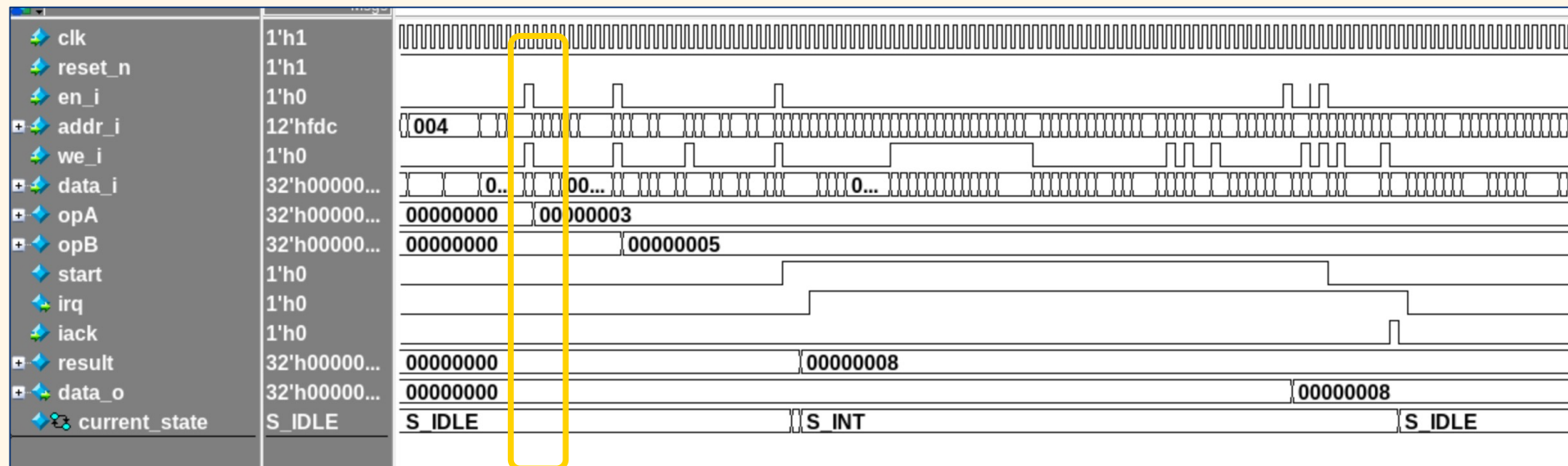
- Verilator: \$finish at 474us; walltime 0.047 s; speed 0.000 s/s

- Verilator: cpu 0.000 s on 1 threads; allocated 217 MB

-- DONE -----

Visualizando a simulação da interrupção

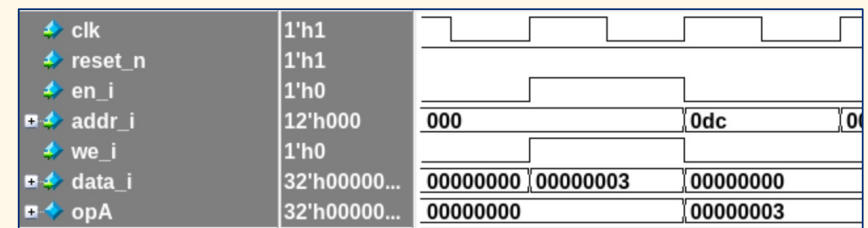
Simular no Questa e inserir os sinais do periférico na forma de onda, como abaixo:



Evento 1:
escreve o
primeiro
operando

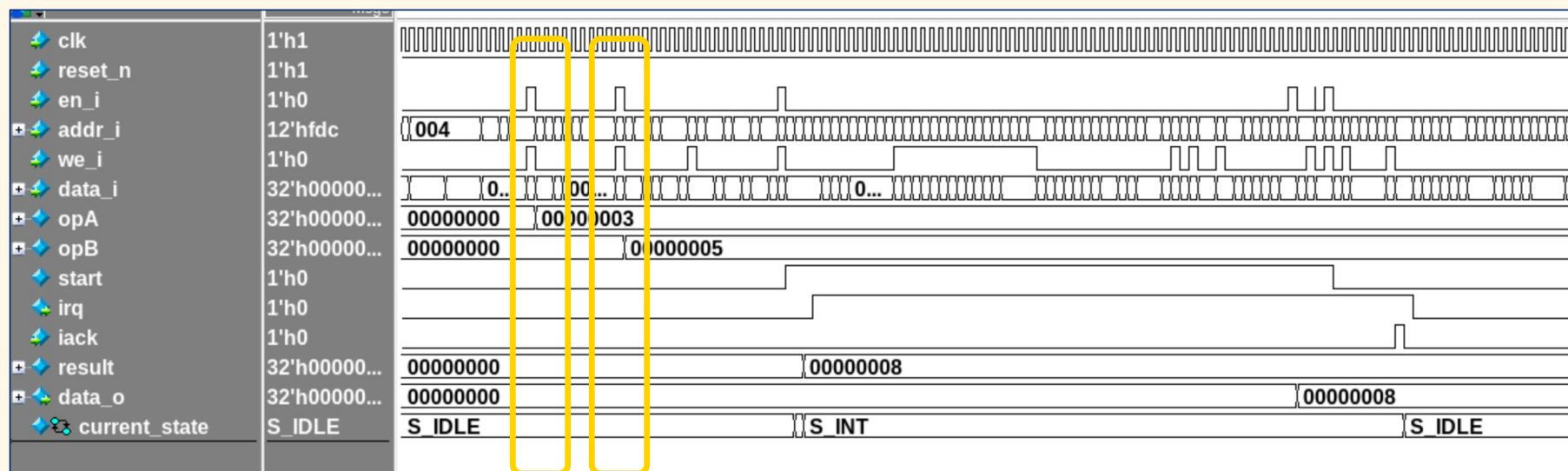
set_operand_a(a)

- o *en_i* e *we_i* sobem
- o valor de *data_i* é armazenado em *opA*



Visualizando a simulação da interrupção

Evento 2: escreve o segundo operando:



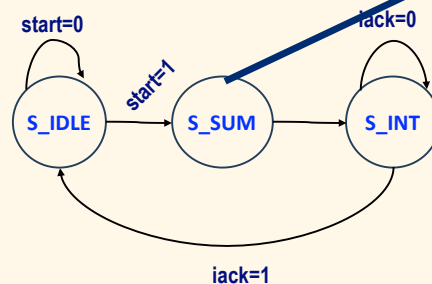
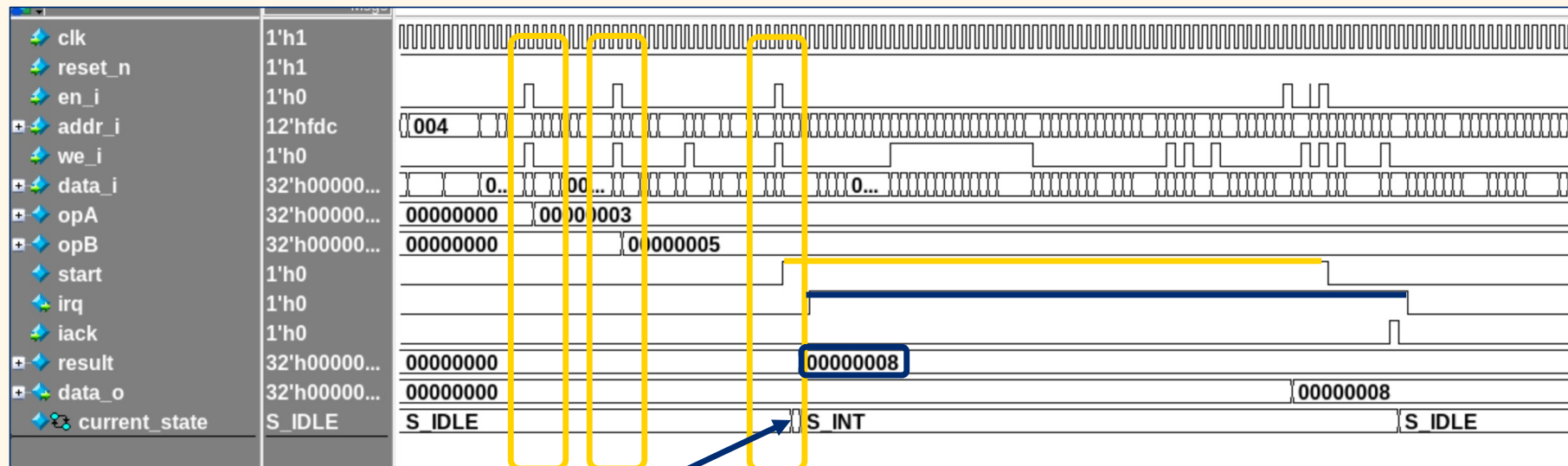
```
set_operand_b(b)
```

→ o en_i e we_i sobem

→ o valor de *data_i* armazenado em opB

Visualizando a simulação da interrupção

Evento 3: inicia o periférico, sobe start, calcula o resultado, e sobe o pedido de int

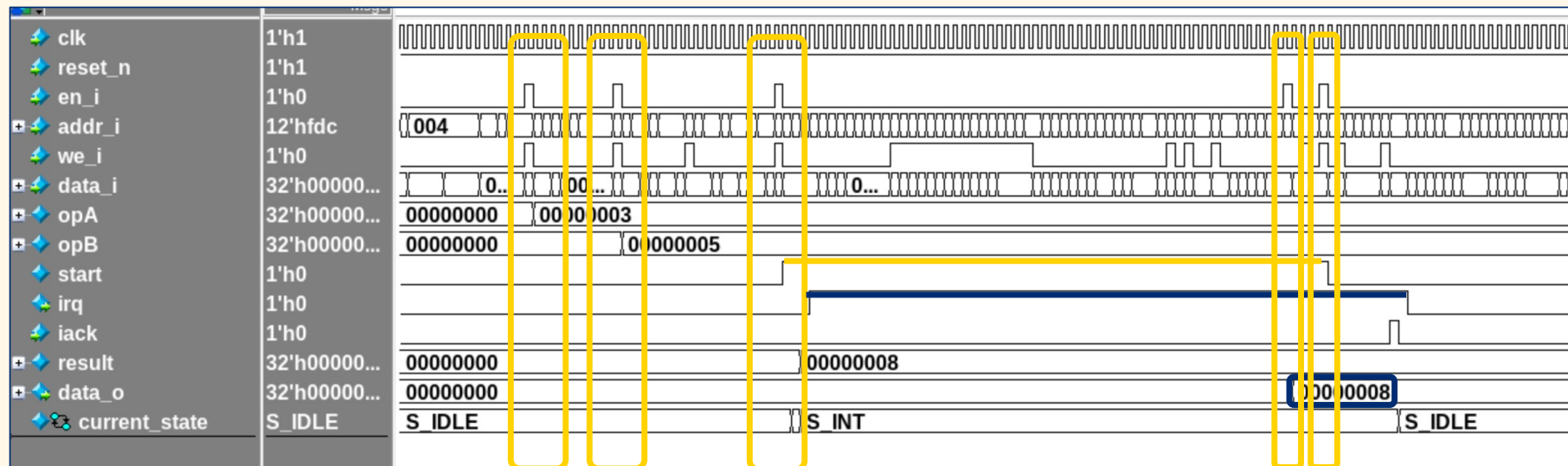


periph_start(); // Start no periférico

- sobe o start
- o periférico calcula a soma
- sobe a interrupção

Visualizando a simulação da interrupção

Eventos 4 e 5: CPU lê o resultado e faz com que o **start** vá a zero(periph_stop());



Observem o tempo para processar a interrupção:

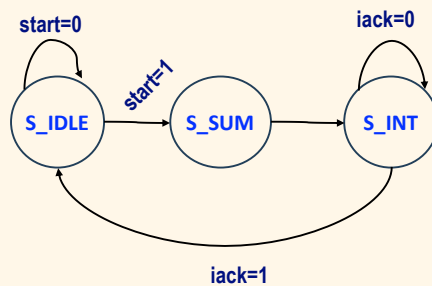
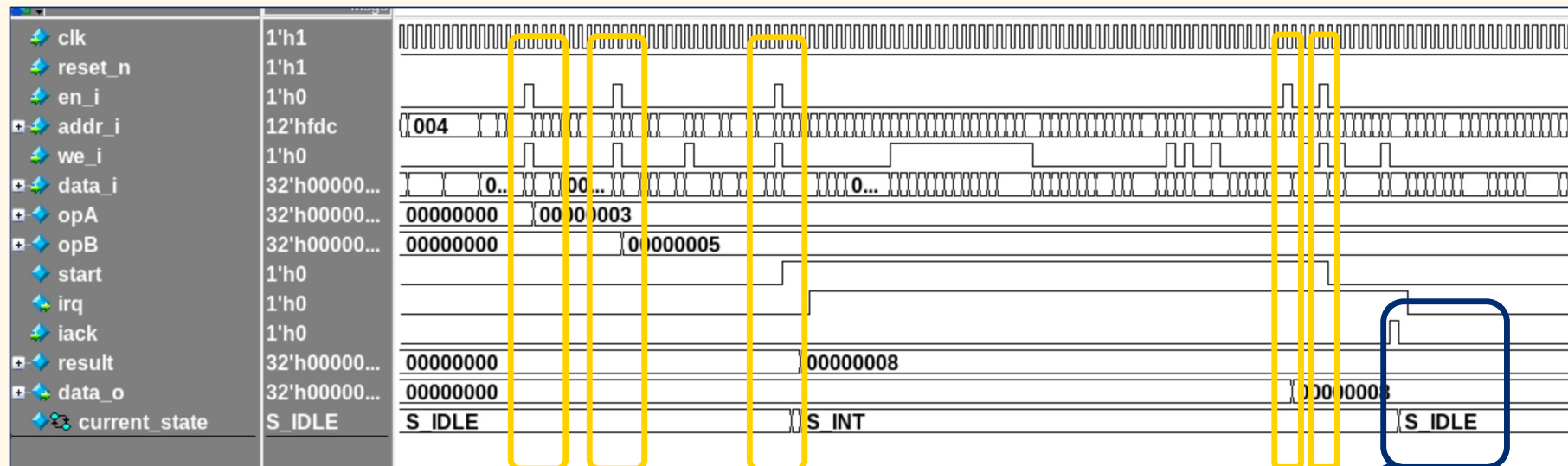
- Uma das causas é a necessidade do **trap_handler** salvar o contexto.

periph_handler

- periph_data = RESULT;
- periph_stop();
- periph_ready = true; // semáforo

Visualizando a simulação da interrupção

Evento 6: Recepção do ack



irq_mei_dispatcher

```
case PERIPH:
    periph_handler();
    PLIC_ACK = PERIPH; //
    envia iack
```


Visualizando a simulação da Interrupção

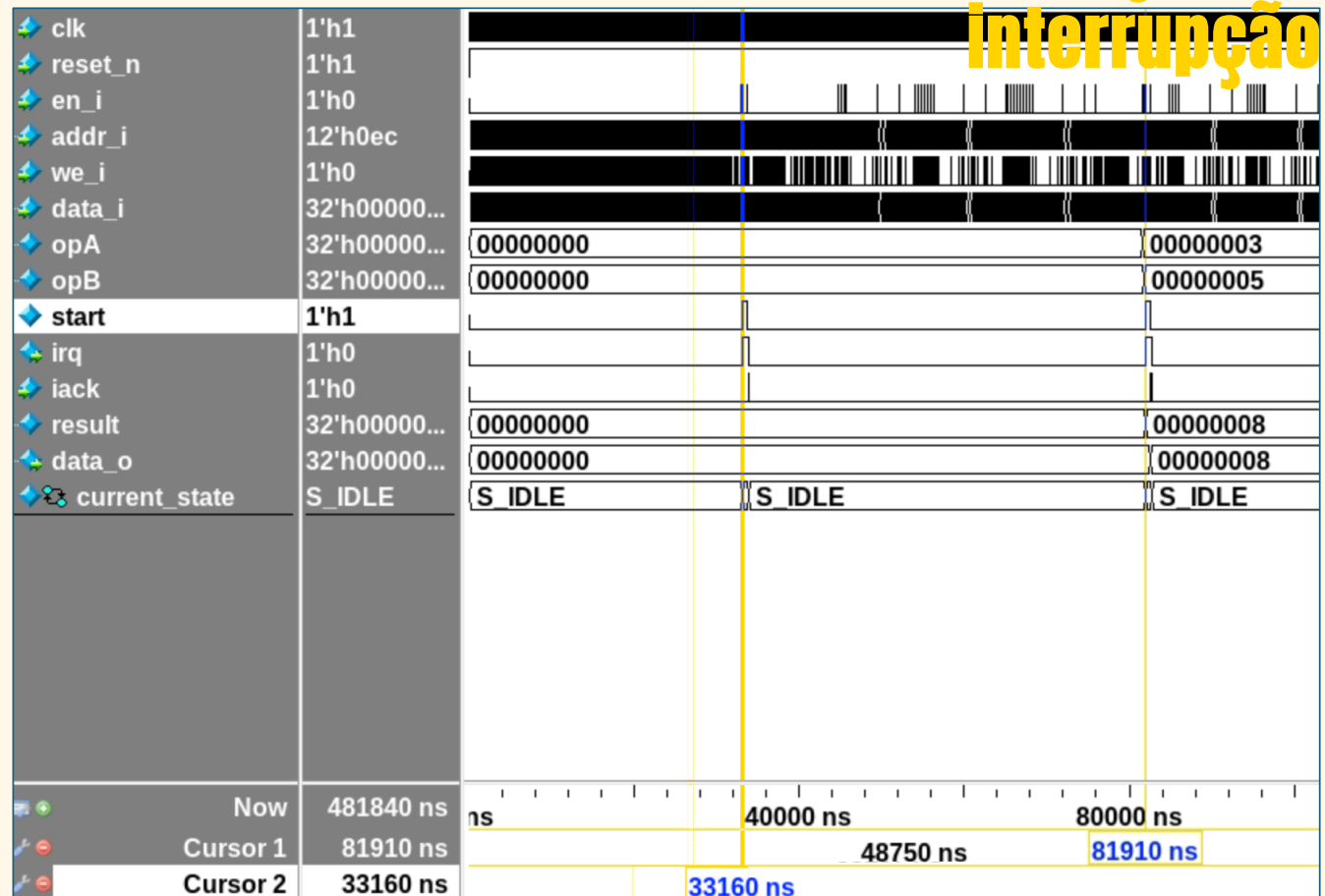
Observar o tempo entre os eventos do sinal.

➤ *start (+-48us)*

➤ *Software consome tempo!*

- Verilator: \$finish at **474us**;

Confere com o Verilator!
(10 iterações)



Exercício

Periférico com um maior número de endereços mapeados em memória (usando offsets)

No lugar de enviarmos: `set_operand_a(a);`

Iremos trabalhar com: `set_operand_a(a, offset);`

Produto Escalar (Dot Product)

O produto escalar (*dot product*) é uma operação matemática que relaciona dois vetores. Seu resultado é um número escalar, calculado pela soma do produto dos elementos correspondentes dos vetores.

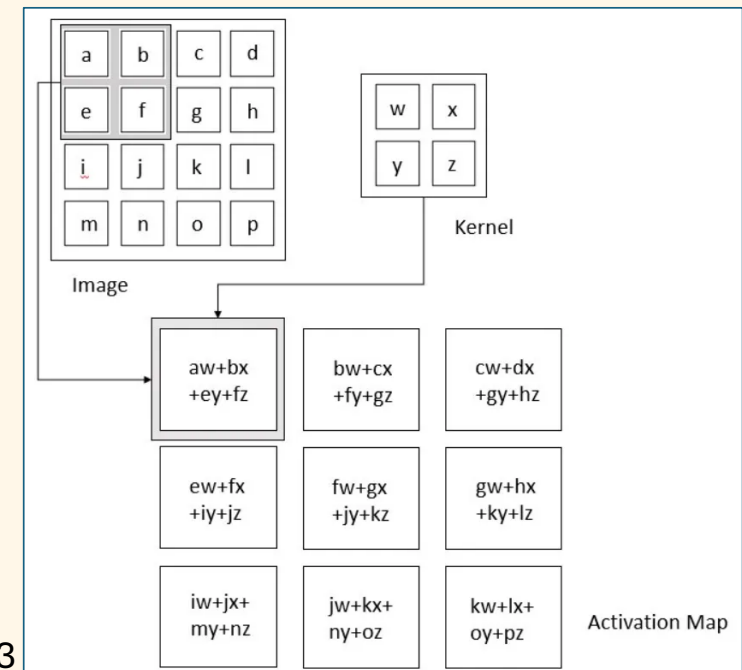
Fórmula Geral:

Seja $u = (u_1, u_2, u_3)$ e $v = (v_1, v_2, v_3)$, então:

$$u \cdot v = u_1 \times v_1 + u_2 \times v_2 + u_3 \times v_3$$

- Vetores: $u = (1, 2, 3)$, $v = (4, 5, 6) \rightarrow u \cdot v = 1 \times 4 + 2 \times 5 + 3 \times 6 = 4 + 10 + 18 = 32$
- Vetores: $u = (0, 1, 2)$, $v = (2, 0, 1) \rightarrow u \cdot v = 0 \times 2 + 1 \times 0 + 2 \times 1 = 0 + 0 + 2 = 2$
- Vetores: $u = (-1, 0, 3)$, $v = (3, -2, 1) \rightarrow u \cdot v = -1 \times 3 + 0 \times (-2) + 3 \times 1 = -3 + 0 + 3 = 0$
- Vetores: $u = (2, 2, 2)$, $v = (1, 1, 1) \rightarrow u \cdot v = 2 \times 1 + 2 \times 1 + 2 \times 1 = 2 + 2 + 2 = 6$
- Vetores: $u = (1, -1, 0)$, $v = (0, 3, -2) \rightarrow u \cdot v = 1 \times 0 + (-1) \times 3 + 0 \times (-2) = 0 - 3 + 0 = -3$

Operação de Convolução (CNN)



Software_(1/4)

Criar uma nova aplicação a partir da aplicação peripheral-test:

- cd app
- cp -r peripheral-test/ perif2

perif2

```
|-- include
|   |--Periph.h
|-- Makefile
|-- src
|   |--intr.S
|   |--main.c
|   |--periph.c
|   |--trap.c
```

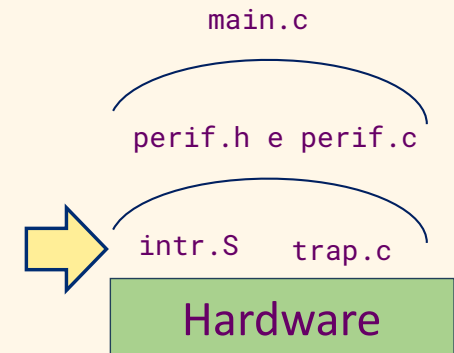
opA e opB tem faixa de endereçamento de 255 posições (OK).

Alterar os ponteiros OPA e os protótipos das funções de escrita:

```
#define OPA(offset) (*(volatile uint32_t *) (0x90000000 + offset*4))
#define OPB(offset) (*(volatile uint32_t *) (0x90000100 + offset*4))
void set_operand_a(int val, int offset);
void set_operand_b(int val, int offset);
```

→ não requer alteração – funções de interface com o processador

→ não requer alteração – funções de configuração do processador



Perif.c: deve agora tratar offsets dos operandos na escrita (lembrar que não implementamos a leitura)

```
perif2
|-- include
|   |--Periph.h
|-- Makefile
|-- src
|   |--intr.S
|   |--main.c
|   |--Periph.c → modificar
|   |--trap.c
```

```
////////////////////////////////////
// Read and write in the peripheral
////////////////////////////////////
void set_operand_a(int val, int offset){
    OPA(offset) = val;
}

void set_operand_b(int val, int offset){
    OPB(offset) = val;
}
```

Main.c: deve escrever os valores no periférico e ler o valor do dot product.

```
perif2
|-- include
|   `-- periph.h
|-- Makefile
`-- src
    |-- intr.S
    |-- main.c
    |-- periph.c
    `-- trap.c
```

```
int main()
{
    int periph_data = 0;

    int N = 6;
    int u[][3] = {{1,2,3},{0,1,2},{-1,0,3},{2,2,2},{1,-1,0},{-150,75,200}};
    int v[][3] = {{4,5,6},{2,0,1},{3,-2,1},{1,1,1},{0,3,-2},{100,-75}};

    config_intr(); // Habilitar interrupção RS5

    for (int i = 0; i < N; i++)
    {
        for(int j=0; j<3; j++){
            printf("(%3d %3d) ", u[i][j], v[i][j] );
            set_operand_a(u[i][j], j); // j é o índice do
            set_operand_b(v[i][j], j);
        }
        clear_periph_ready(); // Limpa o semáforo (false)
        periph_start();       // Start no periférico
    }
}
```

... continua (só precisei alterar o printf no final)

Testar compilação:

```
perif2
|-- include
|   |--Periph.h
|-- Makefile
|-- src
|   |--intr.S
|   |--main.c
|   |--Periph.c
|   |--trap.c
```

make

```
Compiling src/trap.c...
Compiling src/Periph.c...
Compiling src/main.c...
Compiling ../common/newlib.c...
Assembling src/intr.S...
Assembling ../common/crt0.S...
Linking perif2.elf...
/soft64/util/.....has a LOAD segment with RWX permissions
Generating perif2.bin...
Generating perif2.lst...
```

1. Criar um novo periférico a partir do fornecido (pasta RTL):

```
cp periph.sv periph2.sv
```

2. Mudar o nome do módulo de periph para periph2

```
module periph2 (
```

No Testbench:

```
localparam string BIN_FILE = "../app/perif2/perif2.bin";
```

```
periph2 periph( // alteração a instanciação para usar o novo periférico
```

No perif2.sv:

- Usar vetores para opA e opB:

```
logic [31:0] opA [0:2];  
logic [31:0] opB [0:2];
```
- Ler os valores de opA e opB de acordo com o offset. **Exemplo:**

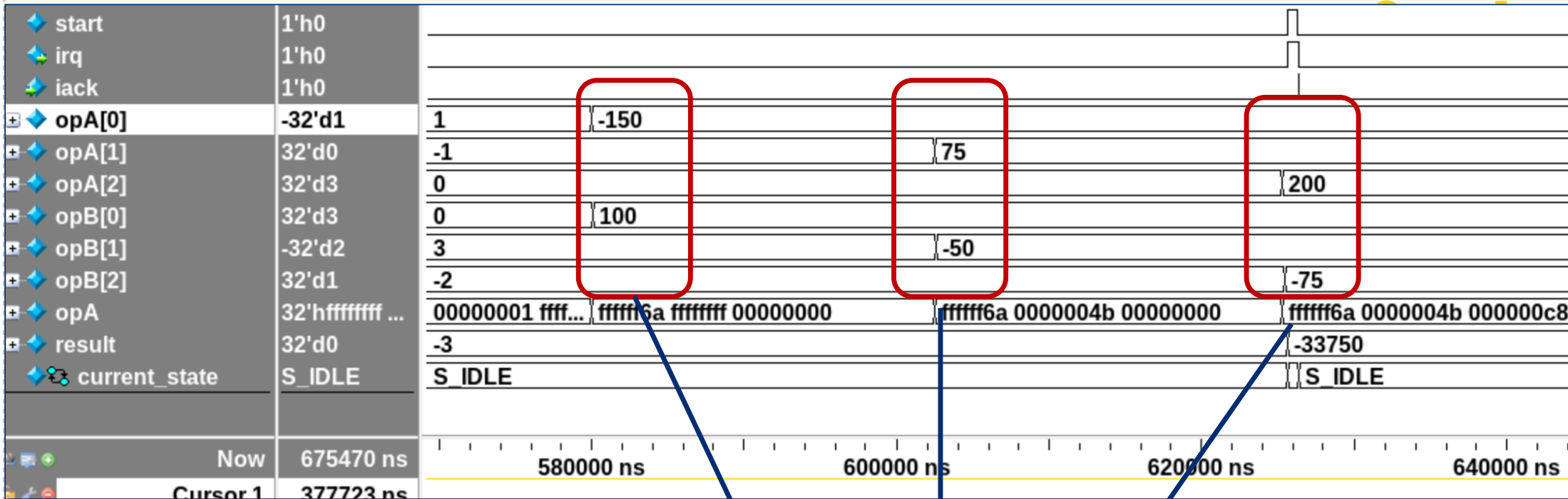
```
12'h004: opA[1] <= data_i;
```
- Implementar o *dot product* em result

Testando o hardware com a aplicação:

```
-- RUN -----  
( 1 4) ( 2 5) ( 3 6) Dot Product 0: 32  
( 0 2) ( 1 0) ( 2 1) Dot Product 1: 2  
( -1 3) ( 0 -2) ( 3 1) Dot Product 2: 0  
( 2 1) ( 2 1) ( 2 1) Dot Product 3: 6  
( 1 0) ( -1 3) ( 0 -2) Dot Product 4: -3  
(-150 100) ( 75 -50) (200 -75) Dot Product 5: -33750
```

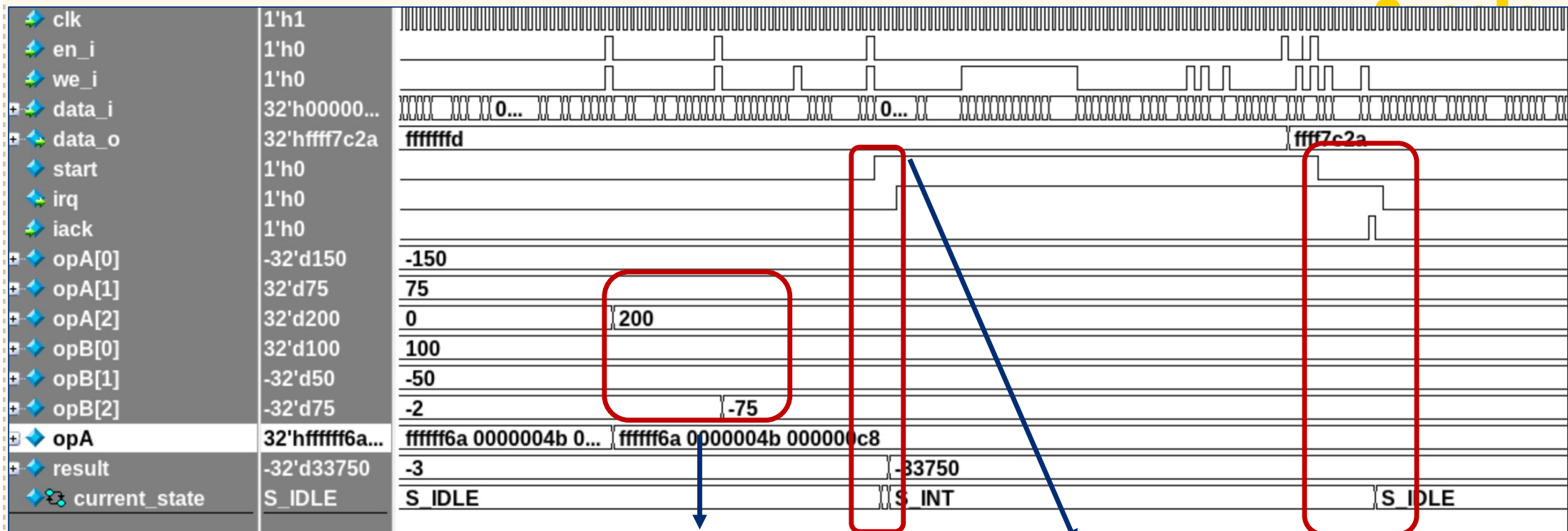
```
# 675470 END OF SIMULATION  
- testbench.sv:326: Verilog $finish  
- Simulation Report: Verilator 5.024 2024-04-05  
- Verilator: $finish at 675us;
```

Simulação do



→ escritas (último dot product)

Simulação do



Terceira escrita

Start e
execução do
dot product

Pedido de INT

Desce start
Int ACK
Desce IRQ (no IDLE)

start	1'h0	
result	-32'd33750	-3, -33750
current_state	S_IDLE	S_IDLE, S_SUM, S_INT

Fim tutorial 3

RS5

Fernando Gehm Moraes - fernando.moraes@pucrs.br

Willian Analdo Nunes - willian.nunes@edu.pucrs.br

Angelo Dal Zotto - angelo.dalzotto@edu.pucrs.br