UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Projeto Final - RTC

Aluno: Pedro Augusto Tortola Pereira - 1561464
Matheus Augusto Burda - 1661736
Disciplina: Lógica Reconfiguravel - ELEW33
Turma: S71

Curso: Engenharia de Computação
Professora: Luiz Fernando Copetti
Data: 5 de julho de 2024

Introdução

Este relatório apresenta o desenvolvimento e resultados obtidos do projeto de um RTC (Real Time Clock), para a disciplina Lógica Reconfigurável. O RTC foi construído em VHDL a partir do software Quartus II, utilizando a placa Cyclone II, e integrado a um servidor desenvolvido em Python, possuindo uma comunicação via Ethernet entre ambos.

Desenvolvimento

O projeto pode ser dividido em 3 grandes etapas, sendo elas: Hardware, Servidor, Conexão. A primeira a ser debatida será a de Hardware

Hardware

Para a construção do RTC em VHDL, tomou-se necessário a utilização de diversos componentes, feitos em VHDL,que serão descritos a seguir.

- **Contador 4 bits:** O componente base da aplicação, responsável pela forma que foi feita a contagem dos demais contadores a seguir. Ele possui os sinais de **RST**, para reset em nível alto, **LD** que é um pino de sinalização para carregar o conteúdo do barramento **LOAD**, setando assim o valor do barramento no contador. Além disso, funciona como um contador de 4 bits comum.

```
architecture count4_arch of count4  is
    signal count: std_logic_vector (3 downto 0) := "0000";
begin
    process (CLK, RST)
        begin
        if (RST = '1') then
            count <= "0000";
        elsif (LD = '1') then
            count <= LOAD;
        elsif (CLK'event and CLK = '1') then
            if (CLR = '1') then
                count <= "0000";
            elsif (EN = '1') then
                    count <= std_logic_vector(unsigned(count)+1);
            end if;
        end if;
    end process;

    Q <= count;

end architecture;
```

- **Contador 60:** Utilizando de dois contadores 4 bits, é o responsável tanto pela contagem dos minutos e segundos. Cada um dos contadores 4 bits é utilizado para manter o controle da unidade e da dezena. A parte mais importante deste código é a forma que é feito o clock enable das dezenas, quando a unidade chega em 9, e os sinais de clear tanto da unidade quanto da dezena.

```
tens_en <= '1' when uni_out_s = "1001" and EN = '1' else '0';

uni_clr  <= '1' when uni_out_s = "1001" and EN = '1' else '0';
tens_clr <= '1' when uni_out_s = "1001" and tens_out_s = "0101" and EN = '1'
            else '0';

TENS_OUT <= tens_out_s;
UNI_OUT  <= uni_out_s;
```

- **Contador 24:** O ultimo contador, responsável pelo manejamento das horas. Também utiliza de dois contadores 4 bits, igual descrito acima, porém com os valores necessários de reset.

```
tens_en <= '1' when uni_out_s = "1001" and EN = '1' else '0';

uni_clr  <= '1' when (uni_out_s = "1001"
                or (uni_out_s = "0011" and tens_out_s = "0010"))
                and EN = '1'
                else '0';
tens_clr <= '1' when uni_out_s = "0011" and tens_out_s = "0010"
                     and EN = '1' else '0';

TENS_OUT <= tens_out_s;
UNI_OUT  <= uni_out_s;
```

- **Clock 1s:** Como a placa utilizada para implementação possui dois clocks disponíveis, e o utilizado é de 50MHz foi necessário uma adaptação para que a sincronia na conta do RTC. O componente Clock 1s é quem faz isso, transformando o clock da placa em um clock de 1s que por sua vez era utilizado nos contadores.

```
architecture a_clk_1s of clk_1s is

signal count         : integer := 1;
signal tmp                 : std_logic := '0';

begin

    process(CLK)
    begin
        if(CLK'event and CLK='1') then
            count <= count + 1;
            if (count = 50000000/2) then
                tmp <= NOT tmp;
                count <= 0;
            end if;
        end if;
    end process;

    CLK_OUT <= tmp;
```

```
end architecture;
```

- **Reg 32:** Componente que foi utilizado para realizar a recepção e passagem de dados do servidor para o cliente. Utilizando de 24 bits, dos seus 32 totais, para a segmentação de unidades e dezenas das horas, minutos, e segundos, cada um com 4 bits.

```
ARCHITECTURE Behavior OF reg32 IS
BEGIN
PROCESS(resetn, clock)
BEGIN
    IF resetn = '0' THEN
        Q <= x"00000000";
    elsif clock'EVENT AND clock = '1' then
        if WE = '1' then
            Q <= D;
        end if;
    end if;
END PROCESS;
END Behavior;
```

- **Rtc:** Responsável pela integração dos contadores com o clock. Componente que recebia o valor qual deveria começar a contagem, a partir do Reg 32, e o enviava para o top level.

```
hour_en <= '1' when
                q_min_tens_s = "0101"
                and q_min_uni_s = "1001"
                and min_en = '1'
                and EN = '1'
            else '0';
min_en  <= '1' when
              (q_seg_tens_s = "0101" and q_seg_uni_s = "1001" and EN = '1')
            else '0';

Q_MIN_TENS <= q_min_tens_s;
Q_MIN_UNI  <= q_min_uni_s;
Q_SEC_TENS <= q_seg_tens_s;
Q_SEC_UNI  <= q_seg_uni_s;
```

- **Rtc Top Level:** Por último, e mais importante, o componente que realiza a integração entre o Rtc e intercepta os dados vindo do servidor. É importante ressaltar o conteúdo de entrada / saída deste componente. Dos 32 bits disponíveis no barramento do processador, somente são utilizados 24, distribuídos nesta sequência:

```
-- reg 0: INPUT FROM NIOS (32 bits)
-- XXXX  XXXX  HHHH hhhh  MMMM mmmm SSSS ssss
-- XXXX -> [24-31]: não utilizado
-- HHHH -> [20-23]: dezenas das horas
-- hhhh -> [16-19]: unidades das horas
-- MMMM -> [12-15]: dezenas dos minutos
-- mmmm -> [08-11]: unidades dos minutos
-- SSSS -> [04-07]: dezenas dos segundos
-- ssss -> [00-03]: unidades dos segundos

rst                  <= not resetn;
write_enable <= write_en and chipselect and (not add);
read_enable  <= read_en  and chipselect and (not add);
rtc_load          <= write_en and chipselect and add and writedata(0);

ld_hour_t <= r32_0_out(23 downto 20);
ld_hour_u <= r32_0_out(19 downto 16);
ld_min_t  <= r32_0_out(15 downto 12);
ld_min_u  <= r32_0_out(11 downto 8);
ld_sec_t  <= r32_0_out(7 downto 4);
ld_sec_u  <= r32_0_out(3 downto 0);

rtc_out(3 downto 0)   <= q_sec_uni;
rtc_out(7 downto 4)   <= q_sec_tens;
rtc_out(11 downto 8)  <= q_min_uni;
rtc_out(15 downto 12) <= q_min_tens;
rtc_out(19 downto 16) <= q_hour_uni;
rtc_out(23 downto 20) <= q_hour_tens;
rtc_out(31 downto 24) <= (others => '0');

readdata <= rtc_out when read_enable = '1' else (others => 'Z');
```

Servidor

O servidor foi confeccionado em Python, utilizando das bibliotecas socket, threading, time e datetime. Com isso em mente, a primeira parte do código se da seguinte forma

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('0.0.0.0', 7777))
server_socket.listen(5)
```

onde um objeto do tipo server_socket é criado e em seguida é associado com a porta 7777, e é configurado para receber até pedidos de conexão.
E então o socket espera a uma conexão na porta especificada, que é feita pela parte do cliente.

```
client_socket, addr = server_socket.accept()
```

Uma vez que a conexão foi estabelecida, a captação do horário é realizada a partir do código

```
timezone_offset = int(input("Enter timezone offset (in hours, positive or negative):
↪  "))
current_time = datetime.now() + timedelta(hours=timezone_offset)
complete_reg_content = ((current_time.hour // 10) << 20) | ((current_time.hour % 10)
↪  << 16) | ((current_time.minute // 10) << 12) | ((current_time.minute % 10) << 8) |
↪  ((current_time.second // 10) << 4) | (current_time.second % 10)
```

O usuário digita o fuso horário desejado, que será atribuído em relação ao Tempo Universal Coordenado (UTC). Em seguida é feita a separação, com operações binarias, para as dezenas e unidades da hora, minuto e segundo. Para que seja possível a mudança de fuso horário no meio da comunicação, foi implementado um processo multi thread da forma

```
input_thread = threading.Thread(target=get_input)
input_thread.daemon = True
input_thread.start()
```

onde o fuso horário é captado da mesma forma, a partir de um input do usuário, e então atualizado na variável global

```
def get_input():
global timezone_offset, client_socket, complete_reg_content
print('Starting input thread')
while True:
    input_str = input('')
    timezone_offset = int(input_str)
    print(f"New timezone: {timezone_offset}")
    print(f"TH: Timezone offset to: {timezone_offset} hours")
    current_time = datetime.now() + timedelta(hours=timezone_offset)
    print(f"TH: New current time: {current_time}")
    complete_reg_content = ((current_time.hour // 10) << 20) | ((current_time.hour %
    ↪  10) << 16) | ((current_time.minute // 10) << 12) | ((current_time.minute % 10)
    ↪  << 8) | ((current_time.second // 10) << 4) | (current_time.second % 10)
```

Por fim, o processo fica em um loop, onde o horário segmentado captado em complete_reg_content é enviado para o cliente, caso tenha sido atualizado. O servidor então recebe o horário do RTC calculado e é exibido a partir da ultima linha do código

```
while True:

    if complete_reg_content is None:
        client_socket.sendall(('ok').encode('utf-8'))
    else:
        print(f'sending to client {complete_reg_content}\n')
        client_socket.sendall(str(complete_reg_content).encode('utf-8'))
        time.sleep(0.5)
        complete_reg_content = None

    response = client_socket.recv(1024).decode('utf-8')

    print(f"RTC Client time: {response}")
```

Conexão

A conexão feita pelo cliente no eclipse começa pela criação do socket, descrita abaixo, conectando no ip do servidor (especificado pela diretiva "inet_addr") e pela porta (7777). Em sequência, é descrito o resto da conexão do cliente.

```
struct sockaddr_in sa;
int SocketFD;

char buffer[1024];

SocketFD = socket(AF_INET, SOCK_STREAM, 6);
printf("Socket created.\n");
sa.sin_family = AF_INET;
sa.sin_port = htons(7777);
sa.sin_addr.s_addr = inet_addr("192.168.0.176");
if (connect(SocketFD, (struct sockaddr *)&sa, sizeof sa) == -1) {
    perror("Connection failed.\n");
    printf("%s\n", strerror(errno));
    close(SocketFD);
    exit(EXIT_FAILURE);
}
msleep(1000);

IOWR(TOP_RTC_0_BASE, 0, 0);
IOWR(TOP_RTC_0_BASE, 1, 1);

char text[16] = "";

int MAX_VALUE = 16777215;
int read_data;
int complete_reg_content;

while(1) {
    memset(buffer,0,strlen(buffer));

    if(recv(SocketFD, buffer, sizeof(buffer), 0) < 0)
```

```
        break;

    if (strcmp(buffer, "ok")) {
        complete_reg_content = atoi(buffer);
        printf("reg content%d\n", complete_reg_content);
        printf("changing time to %d\n", complete_reg_content);
        IOWR(TOP_RTC_0_BASE, 0, complete_reg_content);
        IOWR(TOP_RTC_0_BASE, 1, 1);
    }

   read_data = IORD(TOP_RTC_0_BASE, 0);

   sprintf(text, "%02d:%02d:%02d",
          ((read_data >> 20) & 0xF) * 10 + ((read_data >> 16) & 0xF),
          ((read_data >> 12) & 0xF) * 10 + ((read_data >> 8) & 0xF),
          ((read_data >> 4) & 0xF) * 10 + (read_data & 0xF));

   LCD_Clear();
   LCD_Show_Text(text);

   printf("current time -> %02d:%02d:%02d\n",
              ((read_data >> 20) & 0xF) * 10 + ((read_data >> 16) & 0xF),
              ((read_data >> 12) & 0xF) * 10 + ((read_data >> 8) & 0xF),
              ((read_data >> 4) & 0xF) * 10 + (read_data & 0xF));

    if (send(SocketFD, text, sizeof(text), 0) < 0){
        perror("ABORTED: Socket timed out!");
        exit(EXIT_FAILURE);
    }

   usleep(1000000);
}
```

Conclusão

Com base nos desenvolvimentos e resultados alcançados no projeto do RTC para a disciplina de Lógica Reconfigurável, pode-se concluir que houve um avanço significativo na integração de hardware e software. A implementação em VHDL no Quartus II, utilizando a placa Cyclone II, proporcionou uma base sólida para a construção do Real Time Clock. A integração bem-sucedida com um servidor Python através de comunicação Ethernet, com um cliente em C, se provou complicada por motivos inesperados, como a falha da captação do IP por DHCP do roteador para a placa e afins.