Arquitectura de Computadoras 1 - Trabajo Practico para Promoción

Integrantes:

- Schilliro Agustin
- Dal Bello Juan Cruz

Contador de programa:

Se implementa el contador de programa por medio de un proceso explicito que se activa con el cambio de valor de la señal clk (asignada al puerto Clk del procesador). Este proceso reinicia el valor del registro de instruccion a 0 de manera asincrona si `reset` esta activado. Sino, asigna a `reg_pc` el valor de la direccion en memoria de la siguiente instruccion.

```
process (clk, reset)
    begin
        if reset= '1' then
        reg_pc <= (others =>'0');
    elsif (rising_edge(clk)) then
        if (rising_edge(clk)) then
            reg_pc <= next_reg_pc;
        end if;
    end if;
end process;
```

Para el calulculo de la posicion en memoria de la siguiente instruccion utilizamos procesos implicitos que simulan un MUX.

```
# Siguiente direccion de memoria (direccionamiento normal)
pc_4 <= reg_pc + 4;
```

```
# Direccionamiento por beq
direccion_salto_condicional <= pc_4 + inm_extended_shifted;
```

```
# Direccionamiento pseudo-inmediato por jump
direccion_salto_incondicional <= pc_4(31 downto 28) & target_address_extended;
```

```
# Eleccion entre las tres posibles direcciones de acuerdo a las flags activas
next_reg_pc <= (direccion_salto_condicional) when (Branch and ALU_zero) else
```

```
direccion_salto_incondicional when (Jump) else pc_4;
```

Para el caso de la señal `inm_extended_shifted` se concatenaron los 30 bits menos significativos de la señal `inm_extended` con dos bits:

```
inm_extended_shifted <= inm_extended(29 downto 0) & "00";
```

Y para `target_address_extended` se concatena toda la señal `target_address` con dos bits adicionales para que llegue a los 32 al concatenar esta señal con los 4 bits mas significativos de la direccion actual.

```
target_address_extended <= target_address(25 downto 0) & "00";
```

Banco de registros

Para el banco de regitros, se asignaron sus dos entradas con las direcciones `rs` y `rt` de la instruccion. La flag que habilita la escritura proviene de la unidad de control, la cual se explica mas adelante. La salida 1 `data1_rd` se asignó a la entrada `a` de la ALU, y la salida 2 `data2_rd` entra en un MUX que define el valor `b` de la ALU. La entrada `data_write` se define por medio de un MUX.

```
Port map (
    clk => clk,
    reset => reset,
    wr => RegWrite,
    reg1_rd => I_DataIn(25 downto 21),
    reg2_rd => I_DataIn(20 downto 16),
    reg_wr => reg_wr,
    data_wr => data_Write,
    data1_rd => data1_RegRead,
    data2_rd => data2_RegRead
);
```

Se usa un MUX para asignar el valor que se va a escribir en el banco de registros, eligiendo entre la salida de la memoria de datos o de la ALU.

```
data_Write <= D_DataIn when MemtoReg else ALU_result;
```

ALU

En la ALU, las entrada `a` se asignó a la salida 1 del banco de registros, y la entrada `b` se define usando un MUX junto con el `offset` extendido de las instrucciones `tipo-I` y la flag correspondiente. La salida `zero` se usa como flag al momento de definir los saltos condicionales. El codigo de control se asigna por medio de un proceso explicito. La salida `result` se asigna a la entrada `Addr` de la memoria de datos, para el caso de las

instrucciones `lw` y `sw`, y a la entrada de un MUX que define si se va a escribir desde la memoria de datos o desde la ALU.

```
E_ALU: ALU port map(
    a => data1_RegRead ,
    b => ALU_oper_b ,
    control => ALU_control ,
    zero => ALU_zero ,
    result => ALU_result);
```

La entrada `b` de la ALU va a ser el valor correspondiente a la extension a 32 bits del offset de las instrucciones lw y sw para calcular la direccion de memoria de la cual se va a leer o en cual se va a escribir, si la flag `ALUSrc` esta activa, o la salida 2 `data2_rd` en el caso contrario.

```
ALU_oper_b <= inm_extended when ALUSrc else data2_RegRead;
```

El proceso explicito que define el codigo de operacion de la ALU usa la flag `ALUOp` y los bits del campo `funct` de la instruccion. En caso de una instruccion de `tipo-R`, el campo `funct` indica que operacion logica debe llevar a cambo el componente, y para el resto de operaciones se asigna el codigo correspondiente (de suma para `lw` y `sw` o de resta para `beq`).

```
process (ALUOp, funct)
begin
    case ALUOp is
        when "00" => -- lw o sw
            ALU_control <= "010";

        when "01" => -- branch
            ALU_control <= "110";

        when "10" => -- tipo-R
            case funct is
                when "100000" => -- add
                    ALU_control <= "010";
                when "100010" => -- sub
                    ALU_control <= "110";
                when "100100" => -- and
                    ALU_control <= "000";
                when "100101" => -- or
                    ALU_control <= "001";
                when "101010" => -- slt
                    ALU_control <= "111";
                when others => -- codigo sin usar
                    ALU_control <= "011";
            end case;

        when others => -- ALUOp incorrecto
```

```
            ALU_control <= "011";
    end case;
end process;
```

Unidad de control

Se generan las señales `RegWrite`, `RegDst`, `Branch`, `MemRead`, `MemtoReg`, `MemWrite`, `ALUSrc`, `Jump` y `ALUOp` por medio de un proceso explicito que toma los 6 bits mas significativos de la instruccion.

`RegWrite`: habilita la escritura en el banco de registros.

`RegDst`: habilita el registro de escritura `rt` o `rd`.

`Branch`: habilita el salto condicional.

`MemRead`: habilita la lectura de la memoria de datos.

`MemtoReg`: habilita la escritura en el banco de registros de la salida de la memoria de datos (`lw`) o la salida de la ALU (`tipo-R`).

`MemWrite`: habilita la escritura en la memoria de datos.

`ALUSrc`: habilita la salida 2 del banco de registros o el offset extendido (`lw` y `sw`) para la entrada `b` de la ALU.

`Jump`: habilita el salto incondicional.

`ALUOp`: define (junto con el campo `funct` en el caso de las instrucciones de `tipo-R`) la operacion que va a ejecutar la ALU.

```
process (op)
begin
    if op = "000000" then -- instrucciones de tipo-R
        RegWrite <= '1';
        RegDst   <= '1';
        Branch   <= '0';
        MemRead  <= '0';
        MemtoReg <= '0';
        MemWrite <= '0';
        ALUSrc   <= '0';
        Jump     <= '0';
        ALUOp    <= "10";

    elsif op = "100011" then -- lw
        RegWrite <= '1';
        RegDst   <= '0';
        Branch   <= '0';
        MemRead  <= '1';
        MemtoReg <= '1';
        MemWrite <= '0';
        ALUSrc   <= '1';
        Jump     <= '0';
        ALUOp    <= "00";
```

```
    elsif op = "101011" then -- sw
        RegWrite <= '0';
        -- RegDst no importa
        Branch   <= '0';
        MemRead  <= '0';
        -- MemtoReg no importa
        MemWrite <= '1';
        ALUSrc   <= '1';
        Jump     <= '0';
        ALUOp    <= "00";

    elsif op = "000100" then -- beq
        RegWrite <= '0';
        -- RegDst no importa
        Branch   <= '1';
        MemRead  <= '0';
        -- MemtoReg no importa
        MemWrite <= '0';
        ALUSrc   <= '0';
        Jump     <= '0';
        ALUOp    <= "01";

    elsif op = "000010" then -- jump
        RegWrite <= '0';
        --RegDst no importa, no se va a guardar en el banco de regs
        Branch   <= '0';
        MemRead  <= '0';
        --MemtoReg no importa
        MemWrite <= '0';
        --ALUSrc no importa
        Jump     <= '1';
        --ALUOp no importa

    else -- codigo de instruccion incorrecto
        RegWrite <= '0';
        RegDst   <= '0';
        Branch   <= '0';
        MemRead  <= '0';
        MemtoReg <= '0';
        MemWrite <= '0';
        ALUSrc   <= '0';
        Jump     <= '0';
        ALUOp    <= "00";
    end if;
end process;
```

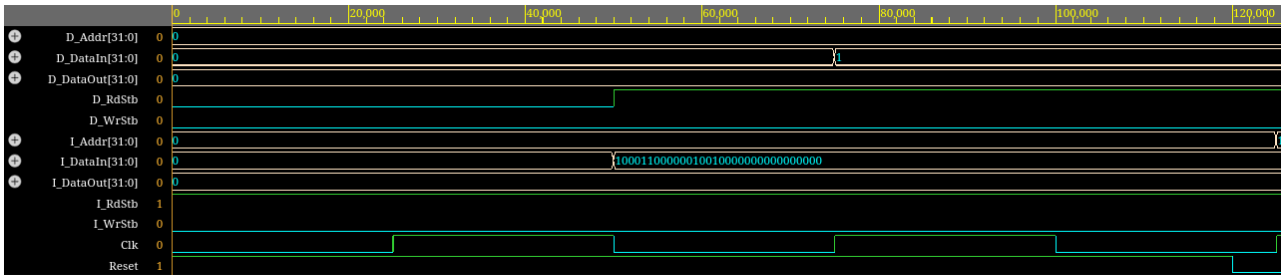
Flanco ascendente y descendente:

La memoria del programa trabaja en flanco descendente asegurándose que el procesador tenga tiempo suficiente para capturar correctamente la siguiente instrucción. Esto permite que el ciclo de lectura y la decodificación estén alineados con el clk. El banco de registros trabaja en flanco descendente para poder

actualizar su contenido de manera que los datos estén disponibles antes del siguiente ciclo de instrucción. Con la memoria de datos ocurre algo similar con el banco de registros, esta sincronizada con el procesamiento de la instrucción, y trabajando en flanco ascendente se asegura que la escritura este alineada con el ciclo de escritura posterior al procesamiento de la próxima instrucción. El contador del programa trabaja en flanco ascendente porque le permite sincronizar el cambio de la dirección de la instrucción con el inicio del ciclo de instrucción. En el flanco ascendente el procesador hace el cálculo para que la memoria de instrucción en flanco descendente la reciba.

Observaciones finales

- La primera instruccion tiene un deley inicial para poder ejecutarse debido a que necesita dos flancos para poder ejecutarse. En el primer flanco descendente se obtiene la primera instruccion en la posicion 0x0 de la memoria de programa debido al `reset` que pone al contador de programa en ese valor, y recien en el proximo flanco ascendente se obtiene la informacion de la memoria de datos. En el descendente de este mismo ciclo se guardan los datos en el banco de registros, pero esto no se puede visualizar en EPWave. En el proximo ascendente luego de desactivarse el `reset` se va a cargar la proxima instruccion del contador de programa.



- Al simular el procesador en el entorno Eda Playground usando EPWave, hay señales asociadas a componentes que trabajan en flanco descendente que se actualizan en ascendente, y viceversa. Sin embargo, esto no significa que los resultados se guarden en ese flanco. Ej: la señal `data_wr` asociada al dato que se va a guardar en el banco de registros se actualiza en flanco ascendente, pero se guarda en flanco descendente, lo cual no se puede visualizar en EPWave.