Labo Complex Digitaal Ontwerp: ALU

Cas Truyers, Xander Rasschaert
Klas: S2Elecfs
Datum: 30/09/2024

Inleiding

In deze labo-opdracht hebben we een Arithmetic and Logic Unit (ALU) ontworpen. De ALU voert zowel arithmetische als logische operaties uit op twee ingangsoperanden, zoals beschreven in de opdracht. De ALU is een belangrijk onderdeel van een CPU.

1 Analyse

De opgave bestond uit het ontwerpen van een 8-bit ALU die verschillende operaties uitvoert, zoals NOT, AND, OR, XOR, ADD, SUB, CMP, RR, RL, en SWAP. Elke operatie werd beschreven met behulp van VHDL. Hieronder volgt een stap-voor-stap uitleg van de implementatie van het ontwerp.

1.1 Componenten en structuur

De ALU is opgebouwd rond een n-bit adder-component, die zelf gebruik maakt van een full one-bit adder (FA1B). Het leek ons logisch om te beginnen bij de laagste component in deze hiërarchie, de FA1B, en vervolgens stapsgewijs op te bouwen naar de volledige ALU. Dit werd ons ook aanbevolen door de labobegeleider. De FA1B hebben we beschreven op basis van de logische poorten die de functionaliteit van een 1-bit full adder implementeren. De volgende stap was het beschrijven van een n-bit ripple carry adder in VHDL, door meerdere FA1B-componenten te verbinden. De grootste uitdaging hierbij was het correct instantiëren en koppelen van de verschillende componenten, waarbij we ons gebaseerd hebben op het schema dat in de labo-opgave werd voorzien. Tot slot hebben we de ALU verder opgebouwd. Door deze stapsgewijze aanpak konden we elke component afzonderlijk simuleren voordat we verder gingen naar de volgende laag in de hiërarchie.

1.2 Code Fragmenten

Hieronder volgt een voorbeeld van de VHDL-code die bij de openstaande secties, gemarkeerd als TO-DO, ingevuld werden. De comments dienen als uitleg.

1.3 FA1B.vhd

```
ENTITY FA1B IS
        PORT(
                -- TODO: complete entity declaration
                A : in std_logic;
                B : in std_logic;
                C_in : in std_logic;
                S : out std_logic;
                C_out : out std_logic);
END entity;

-- Architecture describes behaviour of FA1B
-- Implementing the logic gates diagram using xor/and/or gates

ARCHITECTURE LDD1 OF FA1B IS
BEGIN
        S <= (A xor B) xor C_in ; -- resulting output of adder
        C_out <= (A and B) or ((A xor B) and C_in); -- Carry out bit from add operation
END LDD1;
```

1.4 ADD.vhd

```
    architecture LDD1 of ADD is
    -- list of signals and components
     -- Carry signal for connecting the carry signals between
         FA1B
     -- note that we have an extra bit for the carry_out of the
         last adder.
            signal carry : std_logic_vector(C_DATA_WIDTH downto
                0);

    -- components: Declaration of FA1B component for later
        initialization in architecture
    -- same syntax as entity with entity -> component
            component  FA1B IS
            PORT(
                    A : in std_logic;
                    B : in std_logic;
                    C_in : in std_logic;
                    S : out std_logic;
                    C_out : out std_logic);
        END component;


    begin
        carry(0) <= carry_in; -- External carry_in gets put on
            first bit of carry signal
        -- Initializes (C_DATA_WIDTH - 1) times the FA1B
            component to create n-bit full adder
        FA_array: For i in C_DATA_WIDTH-1 downto 0 generate
            Inst_FullAdder: FA1B  PORT MAP(
    -- FA1B input A bit gets linked to the i-th bit from vector
        input A of n-bit full adder
                A => A(i),
    -- FA1B input B bit gets linked to the i-th bit from the
        vector input B of n-bit full adder
                B => B(i),
    -- Carry in (C_in) from FA1B bit gets linked to i-th bit of
        carry signal vector
                C_in => carry(i),
```

```
-- Result (S) from FA1B gets linked to the i-th bit of
    result vector of n-bit adder
            S => Result(i),
-- link the output carry to the input carry of the next FA
            C_out => carry(i+1)
        );
    end generate FA_array;
-- Carry out of n-bit adder components gets linked to the
    last (MSB) of carry signal.
        carry_out <= carry(C_DATA_WIDTH);

end LDD1;
```

1.5 ALU8bit.vhd

```
    -- TODO: complete the following lines to perform logical
         operations
    -- implementation of some operations
    -- We describe the logic for the basic gate operations
    -- simply with the operator of that gate used on input x
         and y

    -- and
    and_result_i <= x and y ;
    -- or
    or_result_i <= x or y ;
    -- xor
    xor_result_i <= x xor y ;
    -- not
    not_result_i <= not x ;

    -- rr shift to the right by slicing and concatinating
    rr_result_i <= '0' & X(C_DATA_WIDTH-1 downto 1);
    -- rl shift to the left by slicing and concatinating
    rl_result_i <= X(C_DATA_WIDTH-2 downto 0) & '0';

    -- swap first half MSB's with second half LSB's, for n-
        bit ALU (only even)
    swap_result_i <= X((C_DATA_WIDTH/2)-1 downto 0) & X(
        C_DATA_WIDTH-1 downto C_DATA_WIDTH/2);

    -- Extra carry signal defined for the rotate operations
        (we chose to make extra carry signal for our
        readability and understanding)
    carry_r <= X(0) when (op = ALU_OP_RR) else
               X(C_DATA_WIDTH -1) when (op = ALU_OP_RL) else
               '0';

    -- TODO: change the adder's secondary input and carry in
        , based on the operation (addition/subtraction)
    -- addition and subtraction

    -- Describing MUX logic at inputs Y and carry_in of n-
        bit adder as described by diagram in assignment
    add_secondary_input_i <= NOT Y when (op = ALU_OP_SUB)
        else
                             Y when (op = ALU_OP_ADD);
    add_carry_in_i <= '1' when (op = ALU_OP_SUB) else
                      '0' when (op = ALU_OP_ADD);
```

```
    -- TODO: set 'result_i' to a specific operation result
        based on the selected operation 'op'
    -- result mux determines which internal signal gets
        linked to the output, determined by the "op" input
        vector of the ALU
    with op select
        result_i <= not_result_i when ALU_OP_NOT,
                    and_result_i when ALU_OP_AND,
                    or_result_i when ALU_OP_OR,
                    xor_result_i when ALU_OP_XOR,
                    add_result_i when ALU_OP_ADD,
                    add_result_i when ALU_OP_SUB,
                    rr_result_i when ALU_OP_RR,
                    rl_result_i when ALU_OP_RL,
                    swap_result_i when ALU_OP_SWAP,
                    (others => '0') when others;

    Z <= result_i;

    -- TODO: control the flags
    -- carry flag: 1 carry flag for SUB, ADD, RR and RL (
        based on op)
    --   don't forget that rotate left/right can also
        produce a carry
    --   you might need some extra signals
    cf <= add_carry_i  when  op = ALU_OP_ADD else   -- Carry
         for ADD or SUB (based on adder carry out)
          not add_carry_i when op = ALU_OP_SUB else
          carry_r when (op = ALU_OP_RR) else
                                          -- Carry for Rotate
              Right (RR) from original LSB (see carry_r
              signal linking above)
          carry_r when (op = ALU_OP_RL) else
                                          -- Carry for Rotate
              Left (RL) from original MSB (see carry_r signal
               linking above)
          '0';

              -- Default case

    -- zero flag
    zf <= '1' when result_i = (result_i'range => '0') else
        '0' ; -- Assign zero-flag with '1' when all bits of
        the signal vector result_i are '0', else assing '0'.
        result_i'range returns the range of result_i

    -- equal, smaller, greater flag -> using the "when ...
        else" syntax to simply assign '1' when the condition
        is true and '0' when false
    ef <= '1' when X = Y else '0' ;
    gf <= '1' when X > Y else '0' ;
    sf <= '1' when X < Y else '0' ;

end Behavioral;
```

2 Simulaties

De verschillende operaties werden gesimuleerd met behulp van de simulator in vivado. Hieronder is de golfvormen te zien die de resultaten van de simulaties weergeven alsook de console output. Onderaan deze sectie staat het RTL schema

van de ALU8bit.

2.1 ADDER.vhd

‘

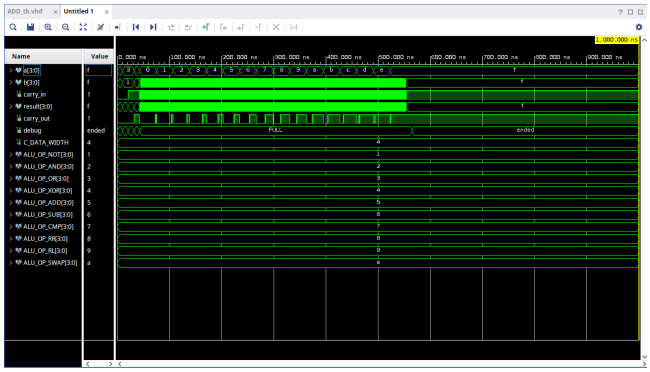

Figure 1: Golfvorm van de simulatie van de ADDER.vhd.

Simulatie Console Output:

run 1000ns
SUCCESS: ADD without carry.
SUCCESS: ADD+carry$_i$*nwithoutcarry.*
$SUCCESS : ADDwithcarry.$
$Note : SIMULATIONENDED$
$Time : 575nsIteration : 0Process : /ADD_tb/STIM_PROC$

Figure 2: Console output van de simulatie van ADD.vhd.

2.2 ALU8bit.vhd

‘

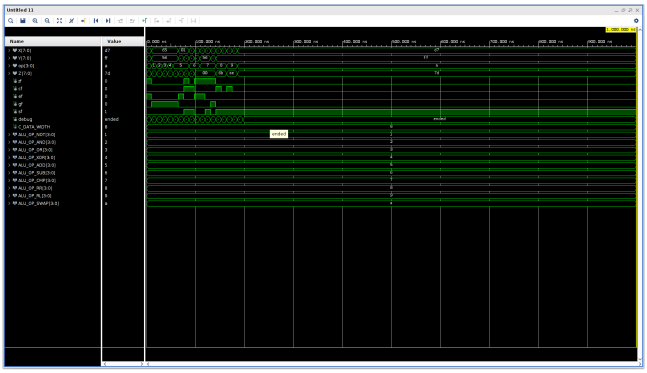

Figure 3: Golfvorm van de simulatie van de ALU8bit.vhd.

Simulatie Console Output:

SUCCESS: NOT.
SUCCESS: AND.
SUCCESS: OR.
SUCCESS: XOR.
SUCCESS: ADD 1.
SUCCESS: ADD 2.
SUCCESS: ADD 3.
SUCCESS: SUB 1.
SUCCESS: SUB 2.
SUCCESS: CMP equal.
SUCCESS: CMP smaller.
SUCCESS: CMP greater.
SUCCESS: RR 1.
SUCCESS: RR 2.
SUCCESS: RL 1.
SUCCESS: RL 2.
SUCCESS: SWAP.
Note: SIMULATION ENDED

Time: 207 ns Iteration: 0 Process: /ALU_tb/STIM_PROC

Figure 4: Console output van de simulatie van de ALU8bit.vhd.

2.3 RTL-Schema

‘

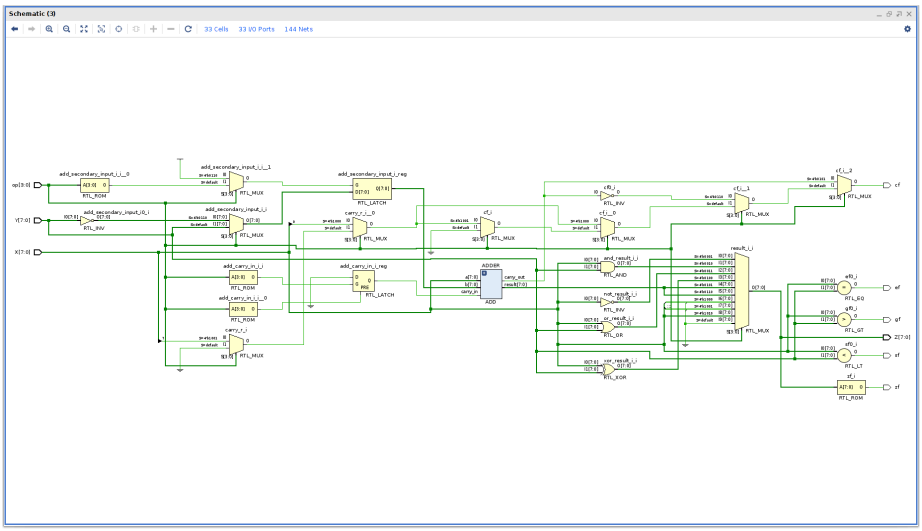

Figure 5: RTL schema van de ALU8bit.

3 Besluit

In dit labo werd een 8-bit ALU en zijn componten ontworpen en succesvol gesimuleerd.

3.1 Wat is gerealiseerd?

We hebben niet alleen een 8-bit ALU ontworpen, maar ook de onderliggende n-bit adder-component en de één-bit adders (FA1B). Deze hiërarchische structuur heeft geleid tot een volledig werkende ALU. Dit werd bevestigd door simulaties die alle doelstellingen van de testbench behaalden.

Daarnaast hebben we gebruik gemaakt van generics, wat de flexibiliteit van het ontwerp vergroot. Dankzij deze aanpak kunnen we, zonder de architectuur aan te passen, eenvoudig een 16-bit ALU implementeren. Hoewel we dit niet expliciet hebben getest, is het ontwerp bewust generiek opgezet om toekomstige uitbreidingen mogelijk te maken.

Ten slotte hebben we met succes de bitstream gegenereerd en geüpload naar een FPGA. De ALU8bit is vervolgens met succes getest op de hardware, wat aantoont dat het ontwerp ook in een praktische setting correct functioneert.

3.2 Wat kan er beter? Wat zijn de beperkingen?

Hoewel het ontwerp van de ALU functioneel is, zijn er enkele aspecten die verder verbeterd kunnen worden. Zo hadden we kunnen testen of de ALU zonder aanpassingen aan de architectuur ook zou werken als een 16-bit versie, puur door gebruik te maken van de generics. Op dit moment zijn sommige delen van het ontwerp echter zo geschreven dat ze alleen werken voor ALU's met een even aantal bits.

Daarnaast zijn er waarschijnlijk optimalisaties mogelijk met betrekking tot de snelheid van de ALU. Omdat snelheid geen prioriteit had binnen de huidige opdracht, is dit aspect niet verder onderzocht, maar daar is waarschijnlijk ruimte voor verbetering. Ook ondersteunt onze ALU enkel integer-operaties. Voor toepassingen die floating-point berekeningen vereisen, zouden extra uitbreidingen nodig zijn.

3.3 Hoe kunnen de problemen opgelost en uitbreidingen uitgevoerd worden?

Wat betreft de snelheid, zouden we de manier waarop de componenten met elkaar communiceren kunnen herzien. Misschien zijn er efficiëntere manieren om de signalen te verbinden, waardoor de ALU sneller kan werken. We kunnen bijvoorbeeld alternatieve manieren onderzoeken om de ripple carry adder te beschrijven. Tot slot, als we de ALU willen uitbreiden naar meer geavanceerde toepassingen, zouden we ondersteuning voor floating-point operaties moeten toe voegen. Dan zou de ALU met decimale getallen kunnen werken.