CSU22022 Computer Architecture I

Eleventh Lecture - RTL

Michael Manzke

2023-2024

Trinity College Dublin

Register Transfer

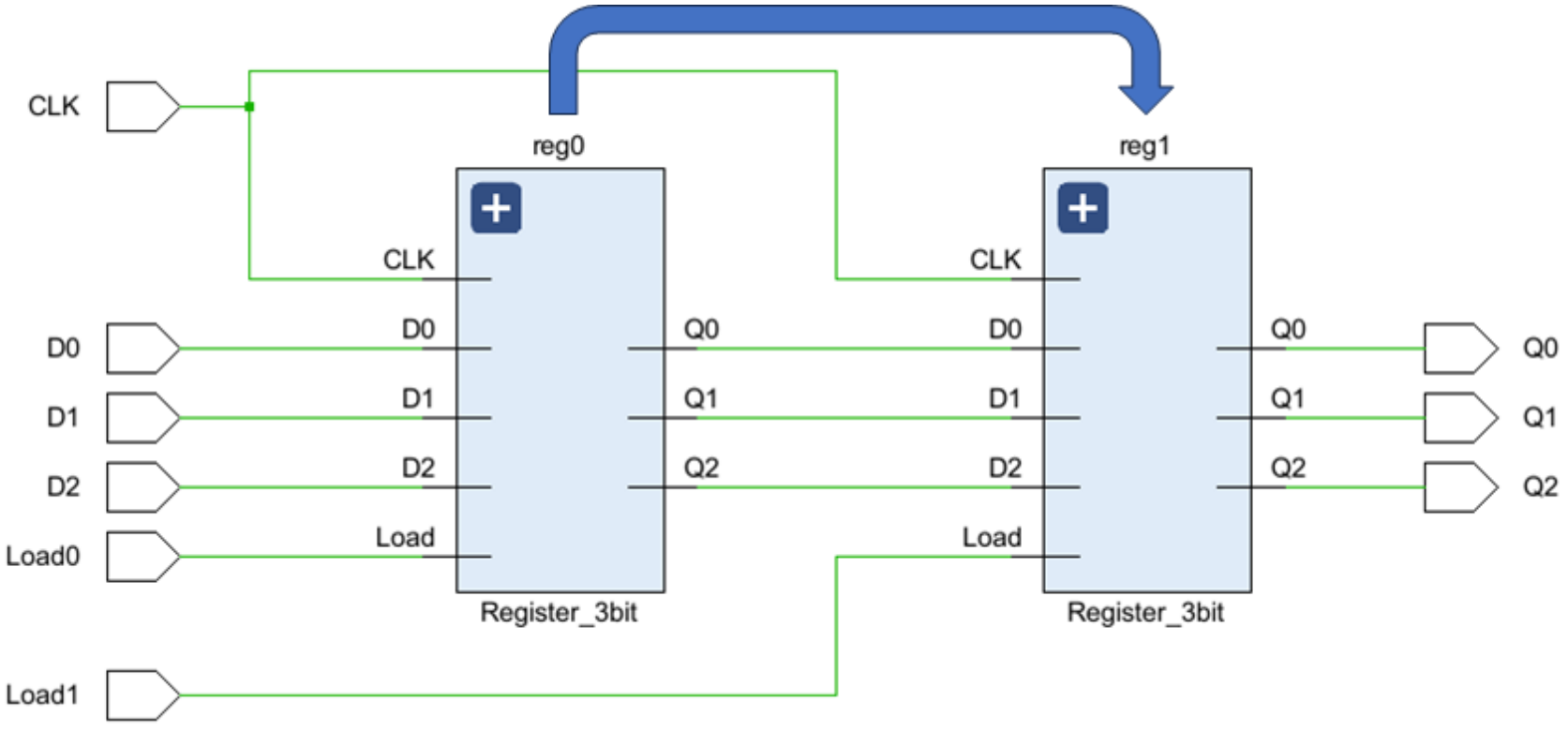


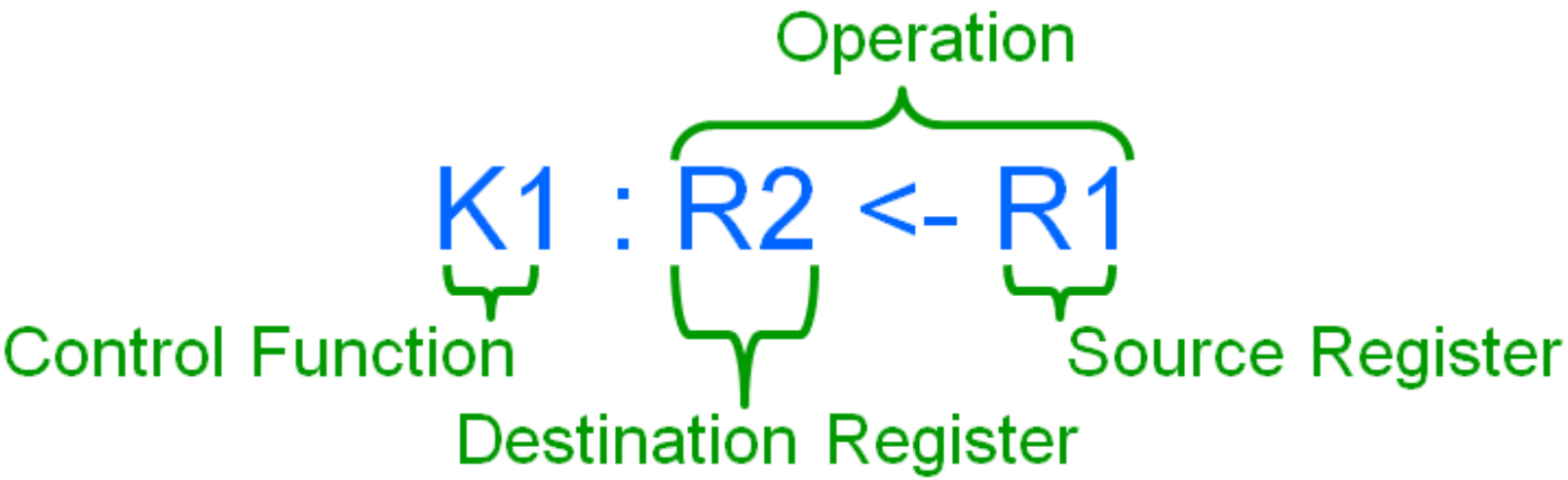
Figure 1: The schematic depicts a register transfer from reg0 to reg1

Register Transfer

- Describing large-scale processor activity
- To discuss digital systems of this scale and level of complexity:
    - We need a number of descriptive tools
- For example:
    - Circuit schematics highlight the circuit components and their connectivity

Register Transfer Specification

- Source Register
- Destination Register
- Operation to be applied
- Condition or control function under which the transfer will occur
    - We assume synchronous operation and omit the clock



Building Register-Transfer Statements

Symbol(s)	Description	Examples
Letters and Numerals	Denote Registers	AR, DR, R2, IR
Parentheses	Denote sections of Registers	R2(9), AR(2), R1(7:0)
Arrow	Denotes data transfer	R1<-R2 IR<-DR
Comma	Separates simultaneous transfers	R1<-R2, R3<-AR
Square bracket	Denote memory addressing	DR<-M[AR] (a read) M[AR]<-DR (a write)

RTL and VHDL

Operation	RTL	VHDL	CSU22022
Combinational Assignment	$=$	<= (CSA)	
Register Transfer	$\leftarrow$	<= (CSA)	
Addition	$+$	+	we don't use it!
Subtraction	-	-	we don't use it!
Bitwise AND	$\wedge$	and	
Bitwise OR	$\vee$	or	
Bitwise XOR	$\oplus$	xor	
Bitwise NOT	$\bar{}$	not	

Table 1: Logical operations

RTL and VHDL

Operation	RTL	VHDL	CSU22022
Shift left (logical)	sl	sll	we don't use it!
Shift right (logical)	sr	srl	we don't use it!
Vector/Register	A(3:0)	A(3 downto 0)	
Concatenation	||	&	

Table 2: Logical operations

Micro-Operation

- A micro-operation is an operation which can be accomplished:
  - Within a small number of gate propagation delays
  - Upon data stored in adjacent registers and memory
- Those commonly encountered in digital systems divide naturally into four groups:
  - Transfer or identity micro-ops copy data
    - e.g., R1 $\leftarrow$ R2, DR $\leftarrow$ M[AR]
  - Arithmetic micro-ops provide the elements of arithmetic
    - e.g. R0 $\leftarrow$ R1 + R2
  - Logic micro-0ps provide per bit operations
    - e.g. R1 $\leftarrow$ R2 or R2
  - Shift micro-ops provide bit rotations
    - e.g. R1 $\leftarrow$ sr R2, R0 $\leftarrow$ rol R1

Arithmetic Micro-ops - Adder

Let R0, R1, and R3 be n-bit Register and consider what can be done with an n-bit Adder:

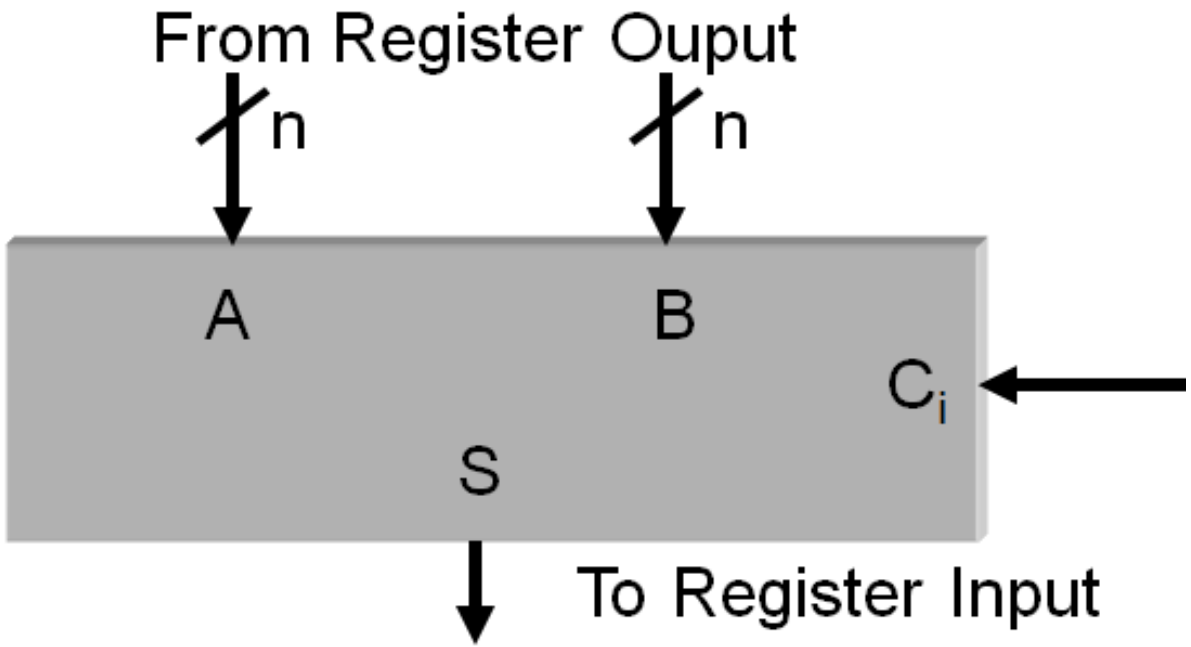


Figure 2: n-bit Adder

Full Adder - See Seventh lecture

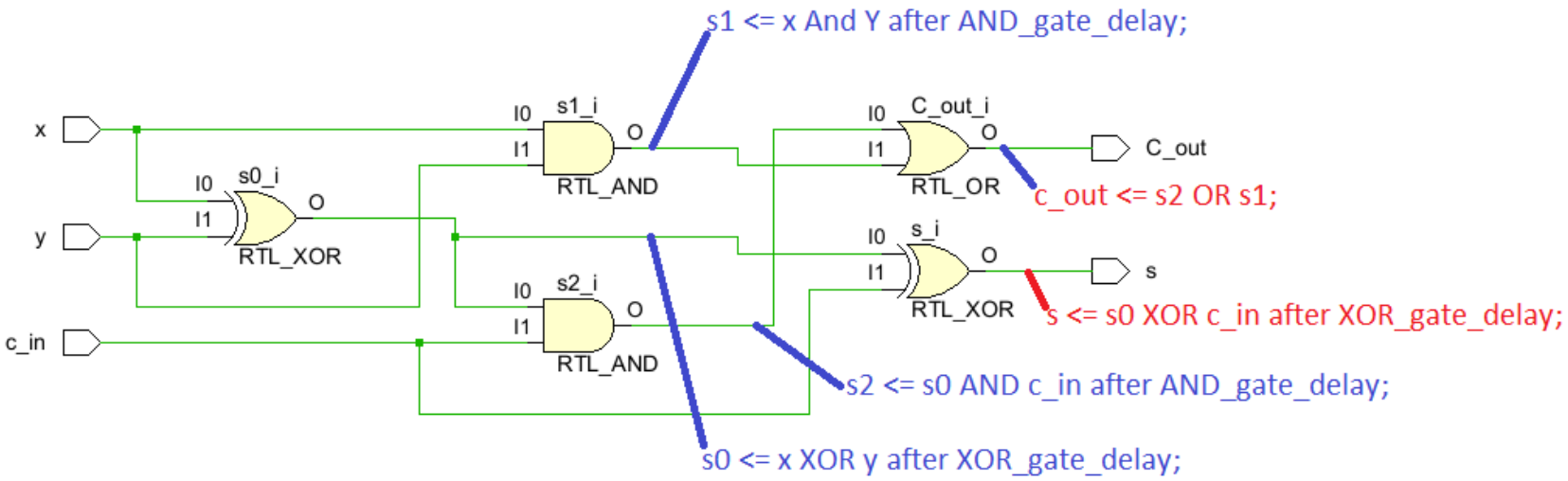


Figure 3: Implementation of a Full Adder with Two Half Adders and an OR Gate

Full Adder VHDL Code - One

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Full_Adder is
    Port ( x : in STD_LOGIC;
           y : in STD_LOGIC;
           c_in : in STD_LOGIC;
           s : out STD_LOGIC;
           C_out : out STD_LOGIC);
end Full_Adder;
```

Listing 1: entity Full_Adder

Full Adder VHDL Code - Two

```
1  architecture Behavioral of Full_Adder is
2      signal s0, s1, s2 : std_logic;
3      --  Propagation Delay according to StdentID e.g. 26 33
       57 25(DEC)
4      constant AND_gate_delay : Time := 6ns; -- least
       significant digit 6=5+1
5      constant NAND_gate_delay : Time := 3ns;-- next more
       significant digit 3=2+1
6      constant OR_gate_delay : Time := 8ns;  -- next more
       significant digit 8=7+1
7      constant NOR_gate_delay : Time := 6ns; -- next more
       significant digit 6=5+1
8      constant XOR_gate_delay : Time := 4ns; -- next more
       significant digit 4=3+1
9      constant XNOR_gate_delay : Time := 4ns;-- next more
       significant digit 4=3+1
10     constant NOT_gate_delay : Time := 7ns; -- next more
       significant digit 7=6+1
```

Listing 2: signals and constants

Full Adder VHDL Code - Three

```
1 begin
2
3     s0 <= x XOR y after XOR_gate_delay;
4     s1 <= x And Y after AND_gate_delay;
5     s2 <= s0 AND c_in after AND_gate_delay;
6     s <= s0 XOR c_in after XOR_gate_delay;
7     c_out <= s2 OR s1;
8
9 end Behavioral;
```

Listing 3: Concurrent signal assignment statements

8-bit Ripple Adder

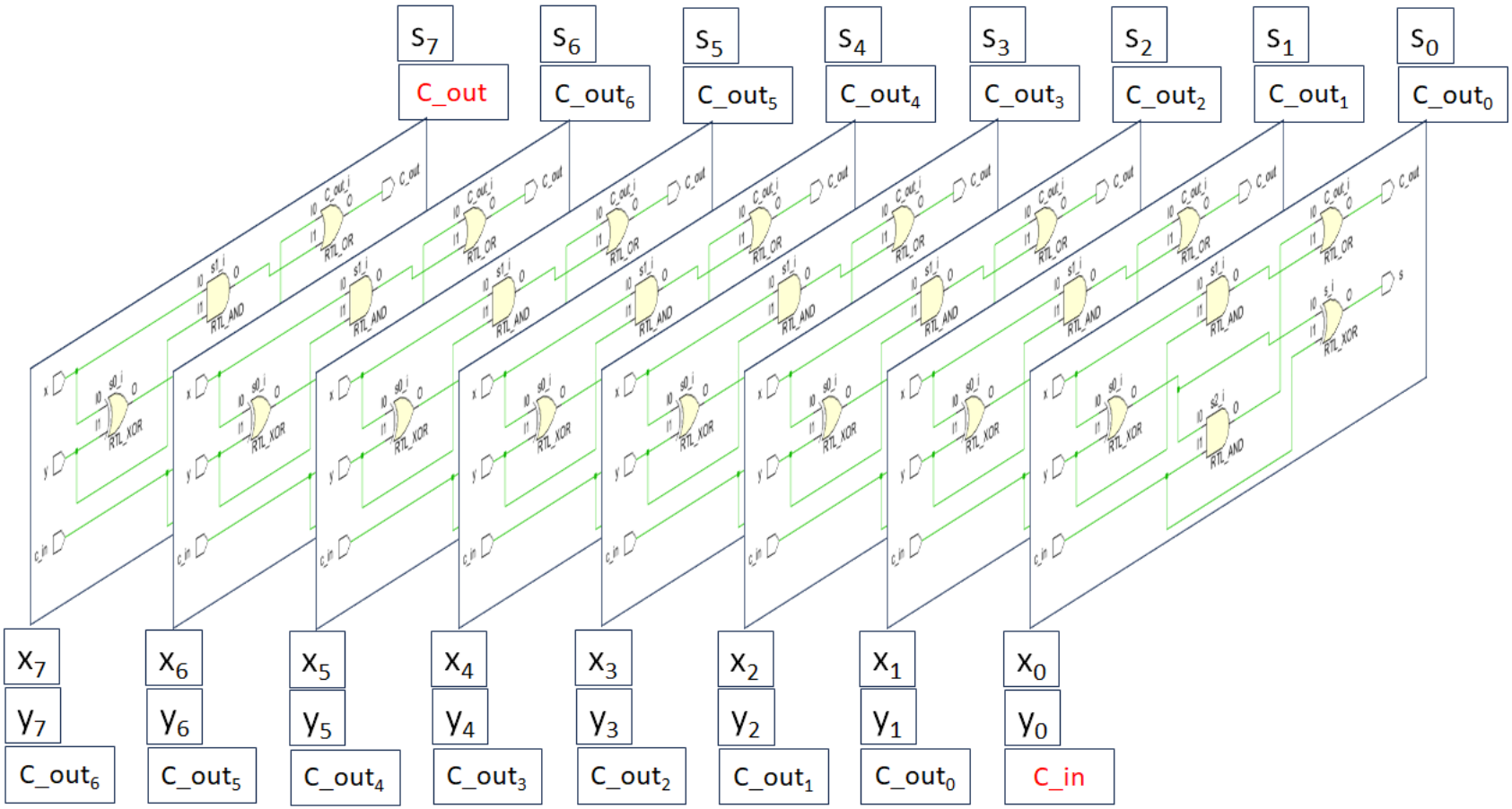


Figure 4: c_out from the less significant bit becomes the c_in for the next more significant bit.

Full 8-bit Ripple Adder VHDL Code - One

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity RippleCarryAdder8_bit is
5     Port ( x, y : in STD_LOGIC_VECTOR (7 downto 0);
6            c_rca_in : in STD_LOGIC;
7            s : out STD_LOGIC_VECTOR (7 downto 0);
8            c_rca_out : out STD_LOGIC);
9 end RippleCarryAdder8_bit;
```

Listing 4: entity Full_Adder

Full 8-bit Ripple Adder VHDL Code - Two

```
1 architecture Behavioral of RippleCarryAdder8_bit is
2
3     COMPONENT Full_Adder
4     Port ( x : in STD_LOGIC;
5          y : in STD_LOGIC;
6          c_in : in STD_LOGIC;
7          s : out STD_LOGIC;
8          c_out : out STD_LOGIC);
9     END COMPONENT;
```

Listing 5: COMPONENT declaration

Full 8-bit Ripple Adder VHDL Code - Three

```
1   signal c_out0, c_out1, c_out2, c_out3 : std_logic;
2   signal c_out4, c_out5, c_out6 : std_logic;
3
4   --  Propagation Delay according to StdentID e.g. 26 33
    57 25(DEC)
5   constant AND_gate_delay : Time := 6ns; -- least
    significant digit 6 =5+1
6   constant NAND_gate_delay : Time := 3ns;-- next more
    significant digit 3=2+1
7   constant OR_gate_delay : Time := 8ns;  -- next more
    significant digit 8=7+1
8   constant NOR_gate_delay : Time := 6ns; -- next more
    significant digit 6=5+1
9   constant XOR_gate_delay : Time := 4ns; -- next more
    significant digit 4=3+1
10  constant XNOR_gate_delay : Time := 4ns;-- next more
    significant digit 4=3+1
11  constant NOT_gate_delay : Time := 7ns; -- next more
    significant digit 7=6+1
```

Listing 6: signals and constants

Full 8-bit Ripple Adder VHDL Code - Four

```
1   -- Instantiate the least significant bit
2   bit0: Full_Adder PORT MAP (
3           x => x(0),
4           y => y(0),
5           c_in => c_rca_in,
6           s => s(0),
7           c_out => c_out0
8         );
9
10  bit1: Full_Adder PORT MAP (
11          x => x(1),
12          y => y(1),
13          c_in => c_out0,
14          s => s(1),
15          c_out => c_out1
16        );
```

Listing 7: Instantiate Full_Adders for bit0 and bit1

Full 8-bit Ripple Adder VHDL Code - Five

```
1   -- Instantiate the least significant bit
2   bit2: Full_Adder PORT MAP (
3           x => x(2),
4           y => y(2),
5           c_in => c_out1,
6           s => s(2),
7           c_out => c_out2
8         );
9
10  bit3: Full_Adder PORT MAP (
11          x => x(3),
12          y => y(3),
13          c_in => c_out2,
14          s => s(3),
15          c_out => c_out3
16        );
```

Listing 8: Instantiate Full_Adders for bit2 and bit3

Full 8-bit Ripple Adder VHDL Code - Six

```
1   -- Instantiate the least significant bit
2   bit4: Full_Adder PORT MAP (
3           x => x(4),
4           y => y(4),
5           c_in => c_out3,
6           s => s(4),
7           c_out => c_out4
8         );
9
10  bit5: Full_Adder PORT MAP (
11          x => x(5),
12          y => y(5),
13          c_in => c_out4,
14          s => s(5),
15          c_out => c_out5
16        );
```

Listing 9: Instantiate Full_Adders for bit4 and bit5

Full 8-bit Ripple Adder VHDL Code - Seven

```
1   -- Instantiate the least significant bit
2   bit5: Full_Adder PORT MAP (
3           x => x(5),
4           y => y(5),
5           c_in => c_out4,
6           s => s(5),
7           c_out => c_out5
8         );
9
10  bit6: Full_Adder PORT MAP (
11          x => x(6),
12          y => y(6),
13          c_in => c_out5,
14          s => s(5),
15          c_out => c_out6
16        );
```

Listing 10: Instantiate Full_Adders for bit6 and bit7

8-bit Ripple Adder Schematic

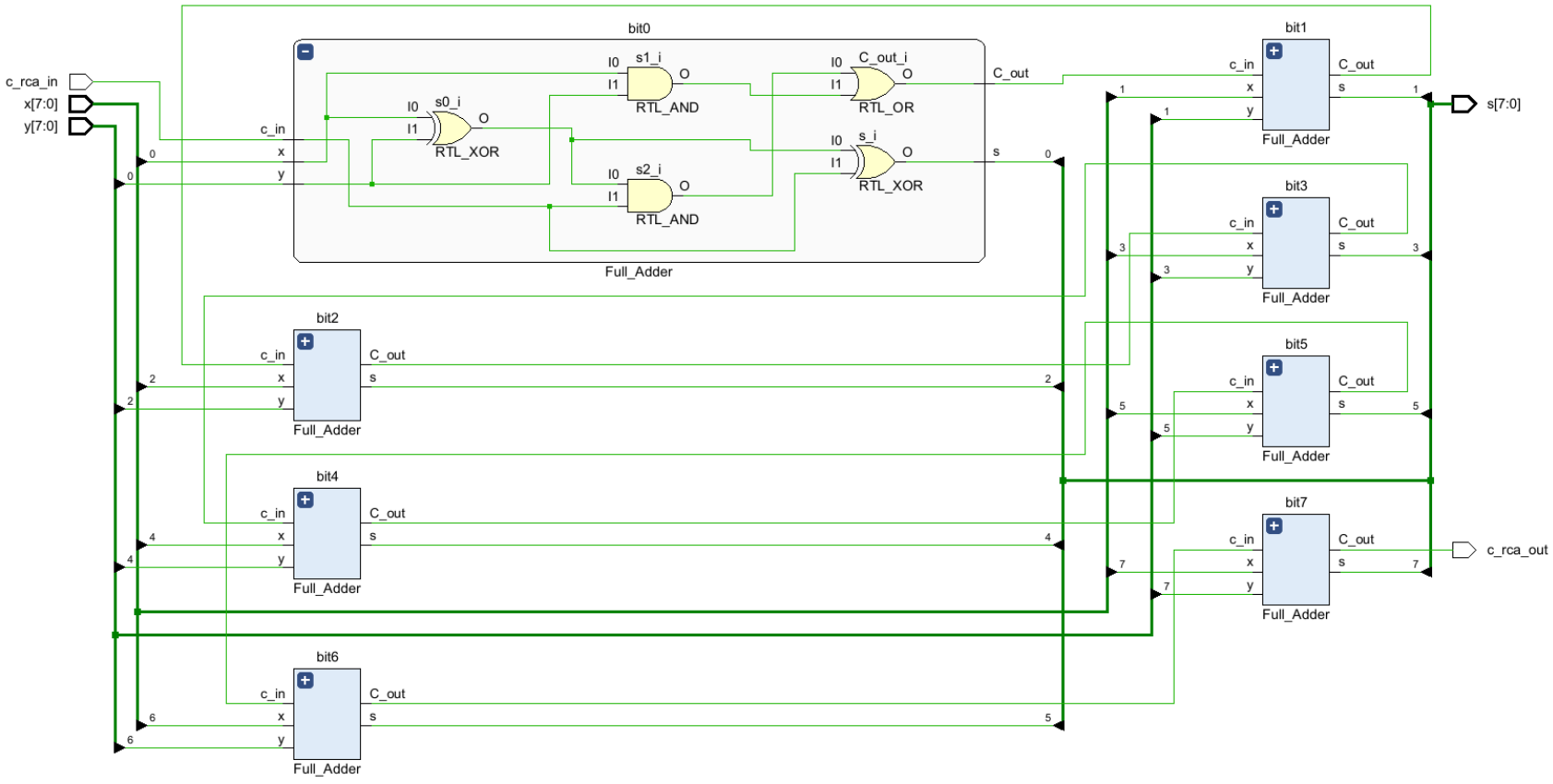


Figure 5: Schematic with detail for the least significant bit.

Conditioned use of an Adder

Operation	A	B	C	Function
R0←R1+R2	R1	R2	0	Addition
R0←R1-R2	R1	$\overline{R2}$	1	Subtraction
R0←R1+1	R1	0...0	1	Increment
R0←R1-1	R1	1...1	0	Decrement
R0←$\overline{R2}$	0...0	$\overline{R2}$	0	1's Complement
R0←$\overline{R2}$+1	0...0	$\overline{R2}$	1	2's Complement

Table 3: By conditioning what arrives at A,B, and C we can achieve the above