Please make sure that the radix for all vectors in your simulation timing diagrams is switched to "Signed Decimal"!

The following VHDL Entities must be tested. Please adapt VHDL-testbenches to the following requirements and run the simulations.

- ***Register_32BitStudentID***

  Load the following sequence of values into the register: 0, StudentID (HEX value), 0, StudentID (HEX value).

- ***Decoder32Bit_5to32StudentID***

  Demonstrate that the decoder generates all 32 correct output signals.

- ***Multiplexer32Bit_32to1StudentID***

  Your simulation should provide the following 32bit values on all 32 inputs of the multiplexer: StudentID (HEX value) onto the first input, StudentID - 1 (HEX value) onto the second input, StudentID - 2 (HEX value onto the third input, … please continue until all 32 multiplexer inputs have a value. Show that you can map these 32 inputs to the output by changing the 5-bit_select_vector.

- ***Multiplexer32Bit_2to1StudentID***

  Your simulation should provide the following 32bit values on the 2 inputs of the multiplexer: StudentID (HEX value) onto the first input, StudentID - 1 (HEX value) onto the second input. Show that you can map these two inputs to the output by changing the select signal.

- ***RegisterFile32Bit_32Plus1StudentID*** (32 x ***Register_32BitStudentID***, 1 x ***Register_32BitStudentID*** (temp register 32), 1 x ***Decoder32Bit_5to32StudentID,*** 2 x ***Multiplexer32Bit_32to1StudentID,*** 32 x *DecoderSignal and RW and not(TD) logic*, 1 x *RW and TD logic*)

  Load your StudentID (HEX value) into the first of the 33 registers. Then load your StudentID - 1 (HEX value) into the 2nd register, your StudentID - 2 (HEX value) into the 3rd register ... please continue until all 33 registers (including temp register 32) have a value.

  Demonstrate that you can not write into the 33 registers when the RW signal is unset.

  Demonstrate that you can make the register content available on the "A Port" and the "B Port" by changing the "A address" and the "B address".

  Demonstrate that the TD, TA, and TB signals overwrite the DR, SA, and SB addresses.

- ***FullAdder_StudentenID***

  Show that your full adder implements the correct truth table for a full adder.

- ***RippleCarryAdder32Bit_StudentID*** (32 x ***FullAdder_StudentenID***, C_V_Logic)

  Simulate the following:

  All numbers are in 2's complement.

  1. positive(StudentID (HEX value) + positive value
  2. positive(StudentID (HEX value) + negative value
  3. negative(StudentID (HEX value)) + positive value
  4. negative(StudentID (HEX value)) + negative value

  5. Demonstrate worst case propagation delay. Also how long is the propagation delay?

  6. If you add a 2's complement number to your StudentID (HEX value), what number would set the C flag and what number would set the V flag?

- ***Multiplexer1Bit_2to1 StudentID***

  Show that the multiplexer works.

- ***BInputLogic_StudentID***

  Simulation your StudentID (HEX value) as input to the ***BInputLogic_StudentID*** and demonstrate that you can output all 0's, your StudentID, 1's complement, and all 1's by changing the select signals S_0 and S_1.

- ***RippleCarryAdderSubtractor32Bit_StudentenID*** (1 x ***BInputLogic_StudentID***, 1 x ***RippleCarryAdder32Bit_StudentID***)

  Your simulation should provide your StudentID (HEX value) on the "A" 32-bit input vector. You should also select a suitable value for the "B" 32-bit input vector.

  Demonstrate that by changing the signals S_0, S_1, and C_{in}, you generate the following outputs:

  A, A + 1, A + B, A + B + 1, A + 1's complement(B), A + 1's complement(B) + 1, A – 1, and A

- ***Multiplexer1Bit_4to1StudentID***

  Show that the multiplexer works.

- ***LogicCircuit1Bit_StudentID*** (AND, OR, XOR, NOT, ***Multiplexer1Bit_4to1StudentID***)

  Demonstrate that our code can perform the following bitwise operations: AND, OR, XOR, NOT

- ***LogicCircuit32Bit_StudentID*** (32 x ***LogicCircuit1Bit_StudentID***)

  Your simulation should provide your StudentID (HEX value) on the "A" 32-bit input vector. You should also select a suitable value for the "B" 32-bit input vector.

  Demonstrate that by changing the signals S_0, and S_1, you generate the following bitwise operations: AND, OR, XOR, NOT.

- ***ALU32Bit_StudentID*** (1 x ***RippleCarryAdderSubtractor32Bit_StudentenID***, 1 x ***LogicCircuit32Bit_StudentID***, 1 x ***Multiplexer32Bit_2to1StudentID***)

  Your simulation should provide your StudentID (HEX value) on the "A" 32-bit input vector. You should also select a suitable value for the "B" 32-bit input vector.

  Demonstrate that by changing the signals S_0, S_1, S_2, and C_{in} you generate the following outputs:

  A, A + 1, A + B, A + B + 1, A + 1's complement(B), A + 1's complement(B) + 1, A – 1, A, A AND B, A OR B, A XOR B, and NOT A.

- ***SR_SL_ShiftUnit_StudentID*** (32 x ***Multiplexer1Bit_4to1StudentID***)

  Your simulation should provide your StudentID (HEX value) as 32-bit input vector.

  Demonstrate that you can shift your StudentID by one bit to the right or one bit to the left or leave it unchanged by changing the signals S_0 and S_1.

- ***Negative_Detect_Logic***

  Demonstrate that the negative-detection logic works.

- ***Zero_Detect_Logic***

  Demonstrate that the zero-detection logic works.

- ***FunctionUnit32Bit_StudentID*** (1 x ***ALU32Bit_StudentID***, 1 x ***SR_SL_ShiftUnit_StudentID***, 1 x ***Multiplexer32Bit_2to1StudentID***, 1 x ***Negative_Detect_Logic***, 1 x ***Zero_Detect_Logic***)

  The order in with you must demonstrate the various operations of your Function-Unit is determined by the last digit of your student number (ID). Please see the following table on the next page for details.

  The simulation timing diagram should show these operations in the correct order and on a single screenshot.

  Furthermore, your testbench should provide your StudentID (HEX value) as 32-bit input vector on the "A" input of your Function Unit and your StudentID (HEX value) plus the last digit of your StudentID on the "B" input of your Function Unit.

		Last Digit of your Student Number (ID)									
		0	1	2	3	4	5	6	7	8	9
Order of micro-operations	1st	F=A (FS=00000)	F=A+B	F=A+B+1	F=A+1's c B	F=A+1's c B+1	F=A-1	F=A AND B	F=A OR B	F=A XOR B	F=1's c A
	2nd	F=sIB	F=A OR B	F=A XOR B	F=A OR B	F=1's c A	F=A (FS=00111)	F=A OR B	F=A XOR B	F=1's c A	F=A+1
	3rd	F=A+1	F=A (FS=00000)	F=A+B	F=A+B+1	F=A+1's c B	F=A+1's c B+1	F=A-1	F=A AND B	F=A OR B	F=A XOR B
	4th	F=A AND B	F=srB	F=A AND B	F=A XOR B	F=A (FS=00111)	F=A XOR B	F=srB	F=B	F=A+1	F=A (FS=00000)
	5th	F=A+B	F=A+1	F=A (FS=00000)	F=A+B	F=A+B+1	F=A+1's c B	F=A+1's c B+1	F=A-1	F=A AND B	F=A OR B
	6th	F=B	F=1's c A	F=B	F=sIB	F=srB	F=B	F=1's c A	F=sIB	F=A (FS=00111)	F=sIB
	7th	F=A+B+1	F=A+1's c B	F=A+1	F=A (FS=00000)	F=A+B	F=A+B+1	F=A+1's c B	F=A+1's c B+1	F=A-1	F=A AND B
	8th	F=srB	F=sIB	F=1's c A	F=B	F=sIB	F=srB	F=B	F=A (FS=00111)	F=srB	F=B
	9th	F=A+1's c B	F=A+B+1	F=A+1's c B+1	F=A+1	F=A (FS=00000)	F=A+B	F=A+B+1	F=A+1's c B	F=A+1's c B+1	F=A-1
	10th	F=1's c A	F=B	F=srB	F=A+1's c B+1	F=B	F=1's c A	F=sIB	F=srB	F=B	F=A (FS=00111)
	11th	F=A (FS=00111)	F=A+1's c B+1	F=A+1's c B	F=A-1	F=A+1	F=A (FS=00000)	F=A+B	F=A+B+1	F=A+1's c B	F=A+1's c B+1
	12th	F=A OR B	F=A XOR B	F=sIB	F=1's c A	F=A XOR B	F=sIB	F=A (FS=00111)	F=1's c A	F=sIB	F=srB
	13th	F=A-1	F=A (FS=00111)	F=A-1	F=A AND B	F=A OR B	F=A+1	F=A (FS=00000)	F=A+B	F=A+B+1	F=A+1's c B
	14th	F=A+1's c B+1	F=A AND B	F=A (FS=00111)	F=srB	F=A-1	F=A AND B	F=A+1	F=A (FS=00000)	F=A+B	F=A+B+1
	15th	F=A XOR B	F=A-1	F=A OR B	F=A (FS=00111)	F=A AND B	F=A OR B	F=A XOR B	F=A+1	F=A (FS=00000)	F=A+B

- ***Datapath32Bit_StudentID*** (1 x ***RegisterFile32Bit_32Plus1StudentID***, 1 x ***FunctionUnit32Bit_StudentID***, 2 x ***Multiplexer32Bit_2to1StudentID*** *(MUX B and MUX D)*)

  Please provide a clock signal for the registers in your register-file. The clock signal must be appropriate for the worst-case propagation delay of your Function Unit.

  Load your StudentID (HEX value) into the first of the 33 registers. Then load your StudentID - 1 (HEX value) into the 2nd register, your StudentID - 2 (HEX value) into the 3rd register ... please continue until all 32 registers have a value.

  The load operation should be via "Data in" port and "MUX D". See Figure 1: Register File and Functional Unit on Project 2 - Microcoded Instruction Set Processor assignment.

  Once all 33 registers (including the temp register 32) are loaded, use the last digit of your student number to select the destination-register (D address).

  Use the (last digit of your student number + 5) to select the source-register A (A address).

  Use the (last digit of your student number + 15) to select the source-register B (B address).

  The order in with you must demonstrate the various operations of your Datapath is determined by the last digit of your student number (ID). Please see the table on the previous page for details. The same order as for the simulation of the Function Unit.

  Maintain the current configuration but switch the "MUX B" to "Constant in". Your simulation should provide your StudentID (HEX value) on the "Constant in" port. Execute the various operations of your Datapath in the order determined by the last digit of your student number (ID) but only those 10 operations that require data on the "B" input of your Function Unit.

- ***Memory32Bit_512StudentID***

  Initialise all 512 memory locations starting with the last two digits of your StudentID (HEX value) at memory address 0 and increment this number by one for all the remaining memory locations e.g. 11, 12, 13 …

  Read all 512 memory addresses.

  Overwrite 32 memory locations starting at the address of the last two digit of your StudentID (HEX value).

  Demonstrate that this doesn't work if the MW signal is unset.

- ***Microprocessor32Bit_StudentID***(1 x ***Datapath32Bit_StudentID***, 1 x ***Memory32Bit_512StudentID***, 1 x ***Multiplexer32Bit_2to1StudentID*** (MUX M))

  Instantiate the datapath, the memory, and MUX M inside the microprocessor entity. Assuming all 512 memory locations are initialised (see ***Memory32Bit_512StudentID***)

  Transfer data from the memory into the 33 registers (including the temp register 32) starting at memory address equal to last two digit of your StudentID (HEX value).

  Calculate the 2's complement for every register and write the values into the same registers. After these operations every register holds the 2's complement of its previous value.

  Transfer the content of all the registers back to its original location in memory.

- ***Control Memory42Bit_256StudentID***

  Initialise all 256 memory locations starting with the last two digit of your StudentID (HEX value) at memory address 0 and increment this number by one for all the remaining memory locations e.g. 11, 12, 13 …

  Read all 256 memory addresses.

- ***Microprocessor32Bit_StudentID***(1 x ***Datapath32Bit_StudentID***, 1 x ***Memory32Bit_512StudentID***, 1 x ***Multiplexer32Bit_2to1StudentID*** (MUX M), 1 x ***Control Memory42Bit_256StudentID***)

  Instantiate the datapath, the memory, the MUX M and control memory inside the microprocessor entity.

  We follow the simulation instructions for the Datapath32Bit_StudentID procedure but this time the control signals (TD, TA, TB, FS, MD, RW, MM, MW) for the datapath come from the control memory and not the testbench.

  Furthermore, the registers in the register-file must be loaded from the memory.

  Assuming all 512 memory locations are initialised (see Memory32Bit_512StudentID).

  Load the first 33 memory locations into the 33 registers.

  Once all 33 registers (including the temp register 32) are loaded, use the last digit of your student number to select the destination-register (D address).

  Use the (last digit of your student number + 5) to select the source-register A (A address).

  Use the (last digit of your student number + 15) to select the source-register B (B address).

  The order in with you must demonstrate the various operations of your Datapath is determined by the last digit of your student number (ID). Please see the table on the

previous page for details. It is the same order as for the simulation of the Function Unit and the datapath.

These 15 operations must be implemented in 15 consecutive memory locations in the control memory by setting the correct control signals (TD, TA, TB, FS, MD, RW, MM, MW).

Your testbench must provide the correct address for the control memory.

VCNZ_FlipFlops_StudentID

Instantiate the datapath, the memory, the MUX M, the control memory, and the VCNZ_FlipFlops inside the microprocessor entity.

Demonstrate that you can set and unset these control flags from the datapath and the control memory.

- ***CAR17Bit_StudentID***

  Demonstrate that you can load, increment and reset the control address register.

- ***IR32Bit_StudentID***

  Demonstrate that you can load this register and generate the correct output signals (Opcode, DR, SA, SB).

- ***PC32Bit_StudentID***

  Demonstrate that the PC increments, adds a 2's complement number to current content of the register, and can be reset to a predefined value.

- ***Multiplexer1Bit_8to1StudentID***

  Demonstrate the functionality of this multiplexer.

- ***ExtendLogic_StudentID***

  Demonstrate the functionality of this logic.

- ***ZeroFillLogic***

  Demonstrate the functionality of this logic.

- ***Microprocessor32Bit_StudentID***(1 x ***Datapath32Bit_StudentID***, 1 x ***Memory32Bit_512StudentID***, 1 x ***Multiplexer32Bit_2to1StudentID*** (MUX M), 1 x ***Control Memory42Bit_256StudentID***, 1 x ***VCNZ_FlipFlops_StudentID***, 1 x ***CAR17Bit_StudentID***, 1 x ***IR32Bit_StudentID***, 1 x ***PC32Bit_StudentID***, 1 x ***Multiplexer1Bit_8to1StudentID*** (MUX S), 1 x ***ExtendLogic_StudentID*** , 1 x ***ZeroFillLogic***)

Main Memory		
Addess base 10		
	-- Machine Code in Memory	
	-- Initialise Address register to 256	
	-- set 0	
LastDigitOfStudentID + 50	**R[DR] <- R[SA] XOR R[SB]**	-- DR=SA=SB= last digit of student ID
	-- set 1	
LastDigitOfStudentID + 51	**R[DR] <- R[SA] + 1**	-- DR=SA= last digit of student ID
	-- set 2	
LastDigitOfStudentID + 52	**R[DR] <- sl R[SB]**	-- DR=SB= last digit of student ID
	-- set 4	
LastDigitOfStudentID + 53	**R[DR] <- sl R[SB]**	-- DR=SB= last digit of student ID
	-- set 8	
LastDigitOfStudentID + 54	**R[DR] <- sl R[SB]**	-- DR=SB= last digit of student ID
	-- set 16	
LastDigitOfStudentID + 55	**R[DR] <- sl R[SB]**	-- DR=SB= last digit of student ID
	-- set 32	
LastDigitOfStudentID + 56	**R[DR] <- sl R[SB]**	-- DR=SB= last digit of student ID
	-- set 64	
LastDigitOfStudentID + 57	**R[DR] <- sl R[SB]**	-- DR=SB= last digit of student ID
	-- set 128	
LastDigitOfStudentID + 58	**R[DR] <- sl R[SB]**	-- DR=SB= last digit of student ID
	-- set 256	
LastDigitOfStudentID + 59	**R[DR] <- sl R[SB]**	-- DR=SB= last digit of student ID
	-- Load Data from MEM into Registerfile	
	-- Load pointer to user data	
LastDigitOfStudentID + 60	**R[DR]←M[R[SA]]**	-- DR=last digit of student ID + 1, SA=last digit of student ID
	-- Load Student ID	
LastDigitOfStudentID + 61	**R[DR]←M[R[SA]]**	-- DR=last digit of student ID + 2, SA=last digit of student ID + 1
	-- Increment Pointer	
LastDigitOfStudentID + 62	**R[DR] <- R[SA] + 1**	-- DR=SA= last digit of student ID + 1
	-- Load Number of shift operation	
LastDigitOfStudentID + 63	**R[DR]←M[R[SA]]**	-- DR=last digit of student ID + 3, SA=last digit of student ID +1
	-- Shift to the right by n bits	
LastDigitOfStudentID + 64	**R[DR] <- n R[SA] sr R[SB]**	-- DR=SB= last digit of student ID + 2, SA=last digit of student ID +3
	-- Increment Pointer	
LastDigitOfStudentID + 65	**R[DR] <- R[SA] + 1**	-- DR=SA= last digit of student ID + 1
	-- Store number of bits in memory	
LastDigitOfStudentID + 66	**M[R[SA]]←R[SB]**	-- SA= last digit of student ID + 1, SB=last digit of student ID + 3
	-- Data in Memory	
256	X"100 + LastDigitOfStudentID"	-- Pointer to user data
256 + LastDigitOfStudentID	X"25123456"	-- your Student ID. This is just an example.
256 + LastDigitOfStudentID + 1	X"00000004"	-- number of bits to shift to right
256 + LastDigitOfStudentID + 2	X"00000000"	-- Result

Control Memory

The following Machine Code Instructions need to be implemented in the Control Memory:

1.	R[DR]←M[R[SA]]	Load	(see lecture notes 16-CSU22022_processor_sixteenth_lecture_2021_2022, page 13/14)
2.	M[R[SA]]←R[SB]	Store	
3.	R[DR] <- R[SA] + 1	Increment	
4.	R[DR] <- R[SA] XOR R[SA]	XOR	
5.	R[DR] <- sl R[SB]	Shift left, 1 bit	
6.	R[DR] <- n R[SA] sr R[SB]	Shift right, n bits	(see lecture notes 17-CSU22022_processor_seventeenth_lecture_2021_2022, page 4)

Last Digit of your Student (ID) **X"???????****?****"** determines the order in which these Machine Instructions must be implemented in the Control Memory

The 2nd Digit of your Student (ID) **X"??????****??****"** in the control memory address of the 1st State of your 1st Instruction of your algorithmic state machine

Therefore, the Opcode for these six machine instructions depends on the last two digits of the Students (ID) and potential implementation differences.

The **IF** and **EXO** microoperations (see lecture notes 16-CSU22022_processor_sixteenth_lecture_2021_2022, page 14) must be implemented after your last instruction.

Therefore, the Opcode for these six machine instructions depends on the last two digits of the Students (ID) and potential implementation differences.

You must provide an Algorithmic State Machine Chart for your implementation (see lecture notes 16-CSU22022_processor_sixteenth_lecture_2021_2022, page 13)

	Order in which these Machine Instructions must be implemented in the Control Memory					
Last Digit of your Student (ID)	1st	2nd	3rd	4th	5th	6th
0	R[DR] <- R[SA] XOR R[SB]	R[DR]←M[R[SA]]	R[DR] <- n R[SA] sr R[SB]	R[DR] <- R[SA] + 1	M[R[SA]]←R[SB]	R[DR] <- sl R[SB]
1	R[DR]←M[R[SA]]	R[DR] <- sl R[SB]	R[DR] <- R[SA] XOR R[SB]	M[R[SA]]←R[SB]	R[DR] <- n R[SA] sr R[SB]	R[DR] <- R[SA] + 1
2	R[DR] <- n R[SA] sr R[SB]	R[DR] <- R[SA] XOR R[SB]	R[DR]←M[R[SA]]	R[DR] <- R[SA] + 1	R[DR] <- sl R[SB]	M[R[SA]]←R[SB]
3	R[DR] <- R[SA] + 1	M[R[SA]]←R[SB]	R[DR] <- sl R[SB]	R[DR] <- n R[SA] sr R[SB]	R[DR]←M[R[SA]]	R[DR] <- R[SA] XOR R[SB]
4	R[DR] <- sl R[SB]	R[DR]←M[R[SA]]	R[DR] <- R[SA] + 1	R[DR] <- R[SA] XOR R[SB]	M[R[SA]]←R[SB]	R[DR] <- n R[SA] sr R[SB]
5	R[DR] <- R[SA] XOR R[SB]	R[DR] <- sl R[SB]	R[DR] <- n R[SA] sr R[SB]	R[DR]←M[R[SA]]	R[DR] <- R[SA] + 1	M[R[SA]]←R[SB]
6	R[DR]←M[R[SA]]	R[DR] <- n R[SA] sr R[SB]	R[DR] <- sl R[SB]	M[R[SA]]←R[SB]	R[DR] <- R[SA] XOR R[SB]	R[DR] <- R[SA] + 1
7	M[R[SA]]←R[SB]	R[DR] <- R[SA] + 1	R[DR] <- R[SA] XOR R[SB]	R[DR] <- sl R[SB]	R[DR] <- n R[SA] sr R[SB]	R[DR]←M[R[SA]]
8	R[DR] <- R[SA] + 1	R[DR] <- R[SA] XOR R[SB]	R[DR]←M[R[SA]]	R[DR] <- n R[SA] sr R[SB]	M[R[SA]]←R[SB]	R[DR] <- sl R[SB]
9	R[DR] <- n R[SA] sr R[SB]	R[DR] <- sl R[SB]	M[R[SA]]←R[SB]	R[DR] <- R[SA] + 1	R[DR]←M[R[SA]]	R[DR] <- R[SA] XOR R[SB]

If your microprocessor can execute the specified machine code and only then you can implement a Carry Look Ahead Adder for an extra 20% mark!

- ***CarryLookAheadAdder32Bit_StudentID***

  Perform the same test as for the ***RippleCarryAdder32Bit_StudentID*** (32 x ***FullAdder_StudentenID***, C_V_Logic) and compare the results.

  Compare the overall propagation delays (all gates in both implementations should have the same propagation delays)