


CSCE 611:
Design of a 32-bit ALU

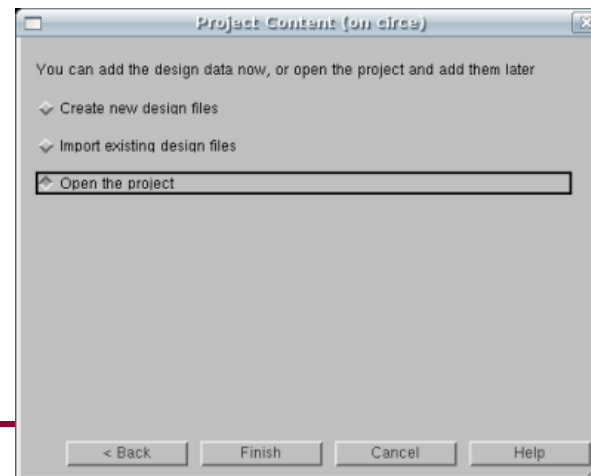
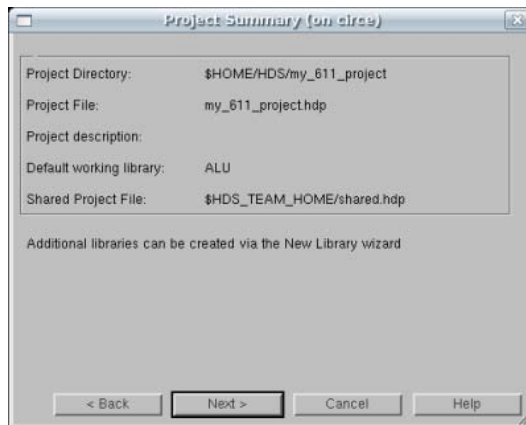
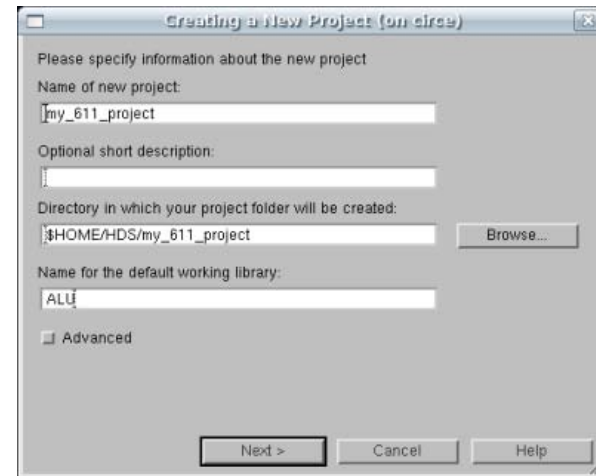
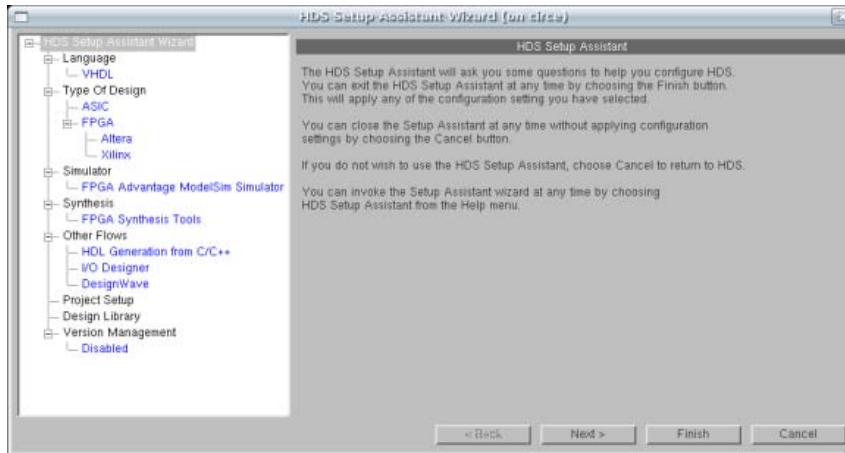


Instructor: Jason D. Bakos



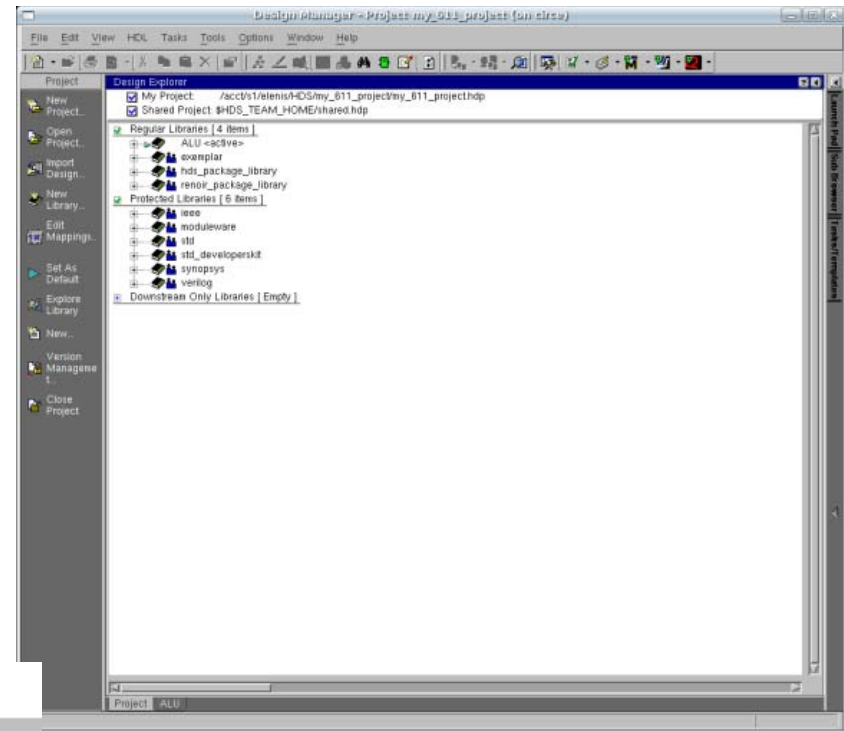
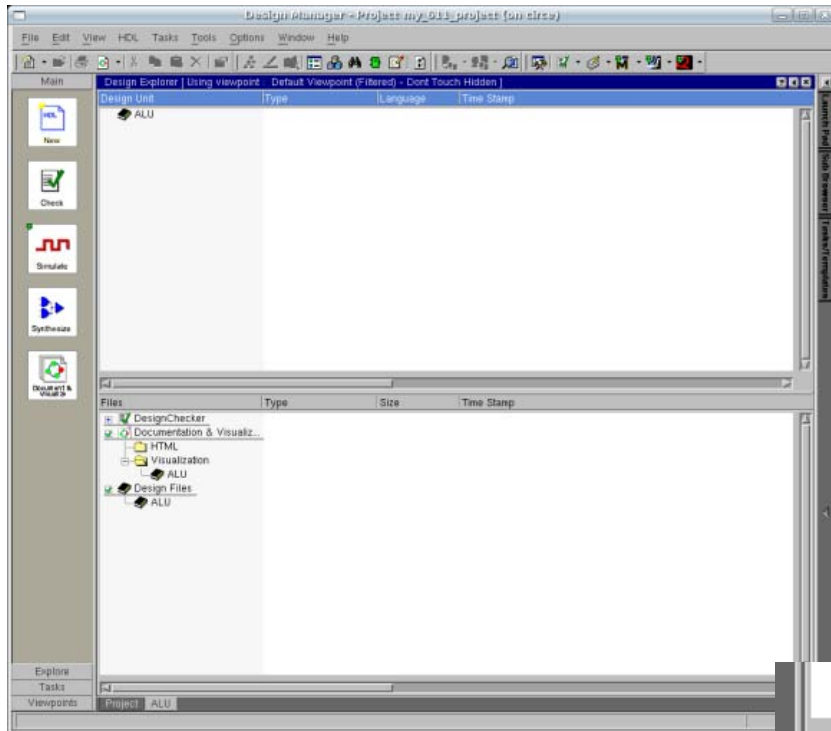
Create Project

- Open FPGA Advantage and create your project

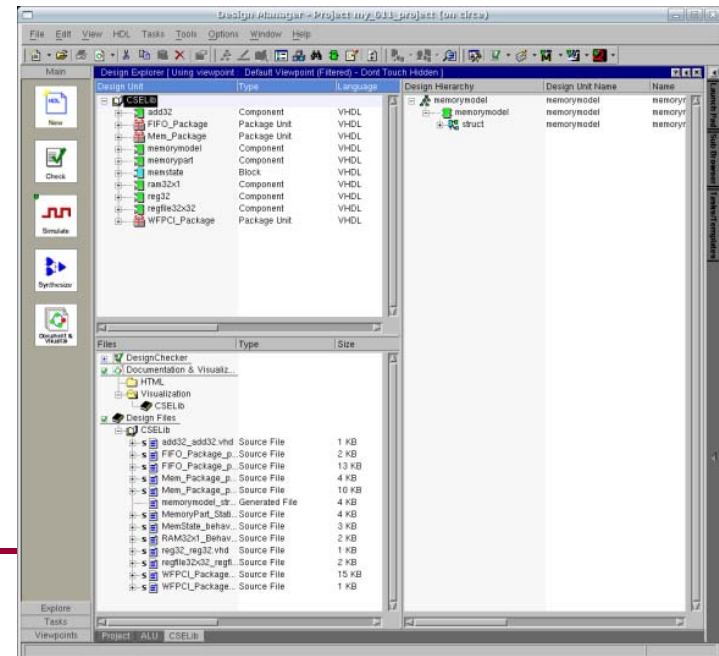
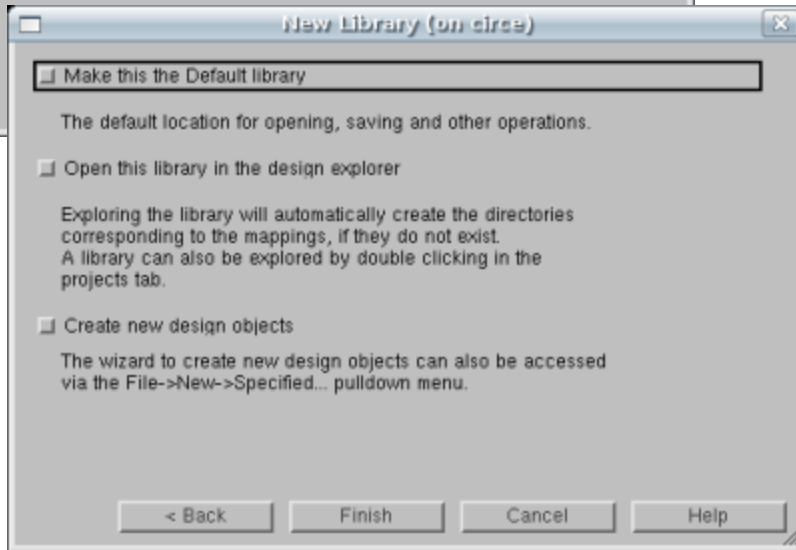
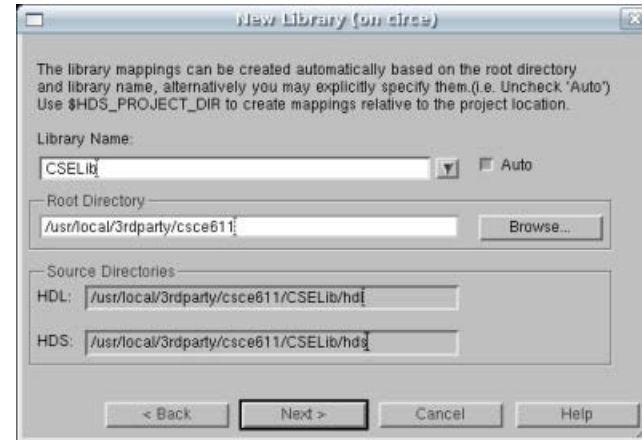
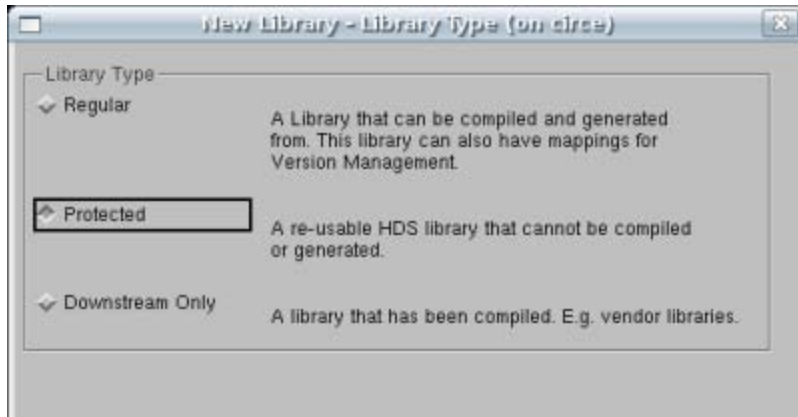


Design Manager

- Library and project views in Design Manager...



Mapping Libraries



ALU Specifications

- Specifications for ALU
 - GOAL: implement all logical, arithmetic, shift, and comparison operations in MIPS instruction set
 - Operations
 - Bit-wise AND, OR, XOR, and NOR
 - Signed and unsigned addition, subtraction
 - » Overflow detection, zero-result detection
 - Signed and unsigned set-on-less-than comparison
 - Logical shift left and right, arithmetic shift right
 - Must accept 2 x 32-bit operands and produce a 32-bit result



ALU Interface

- Inputs
 - A, B (32 bits)
 - SHAMT (how many bits?)
 - ALUOP (how many bits?)
 - 13 total operations
- Outputs
 - C (32 bits)
 - Overflow
 - Zero

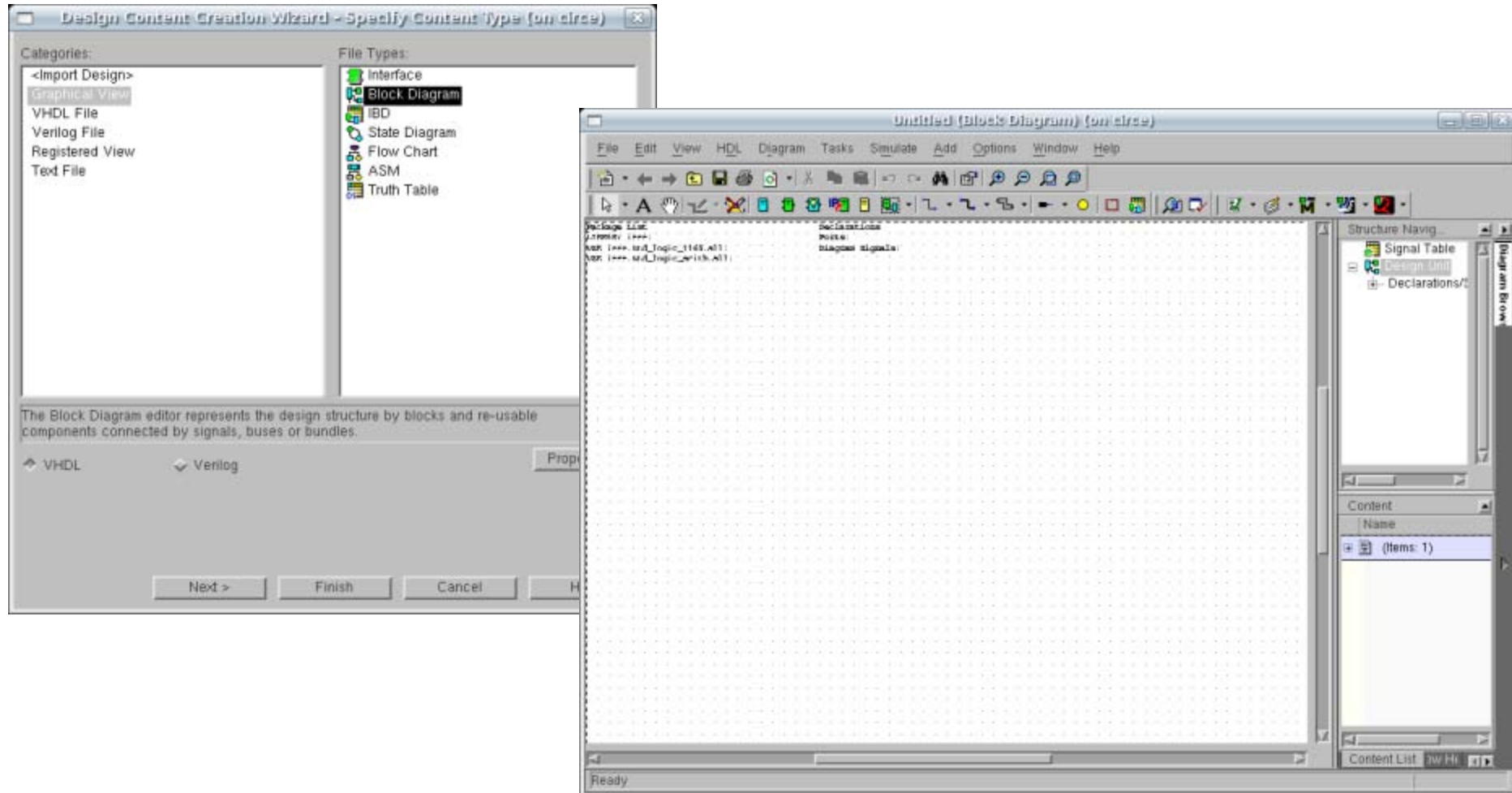


ALU Design Methodology

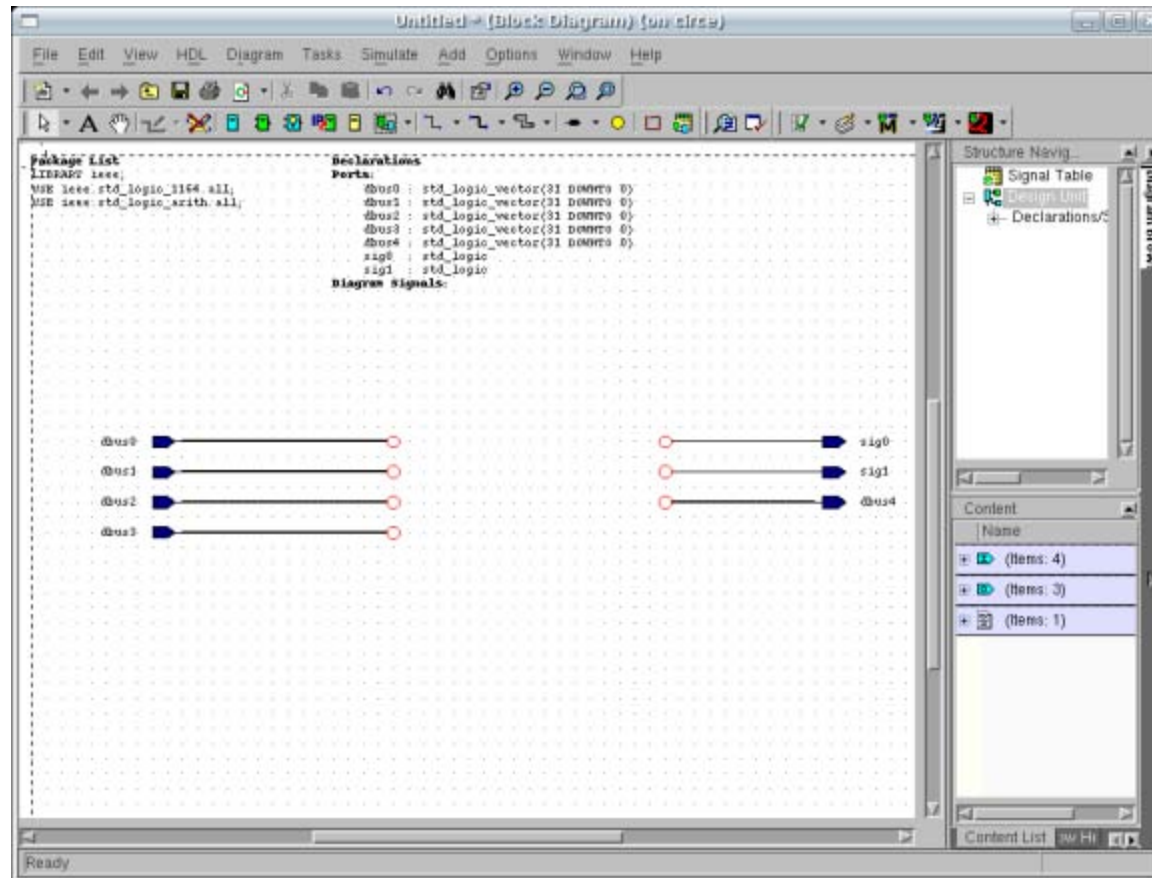
- We will work top-down to design the ALU
 - First step is to create top-level design
 - Need to choose a *view* which will implement a VHDL *architecture*
 - View type: *block diagram*
 - Implements structural VHDL
 - From design browser...
 - File | New | Design Content | Graphical View | Block Diagram



Block Diagram Editor

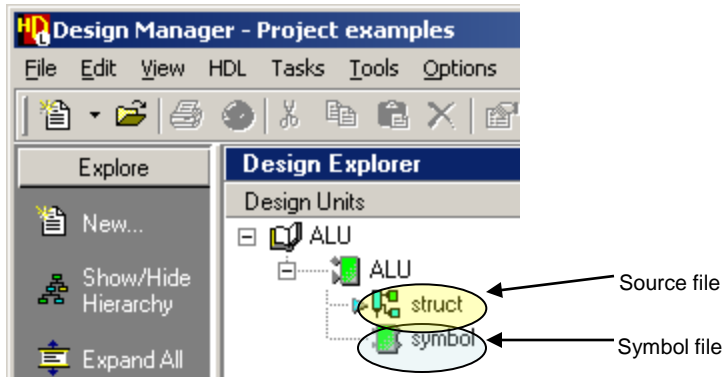
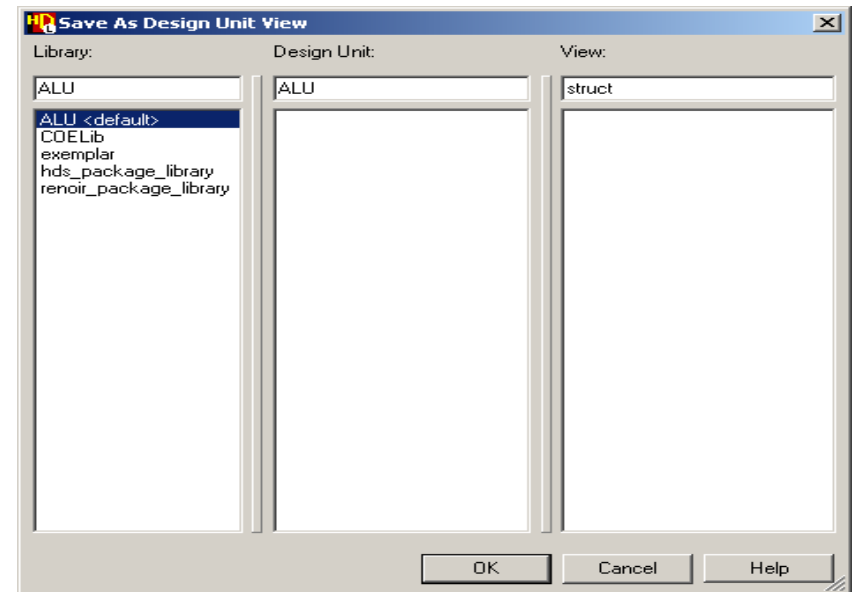


Adding Ports



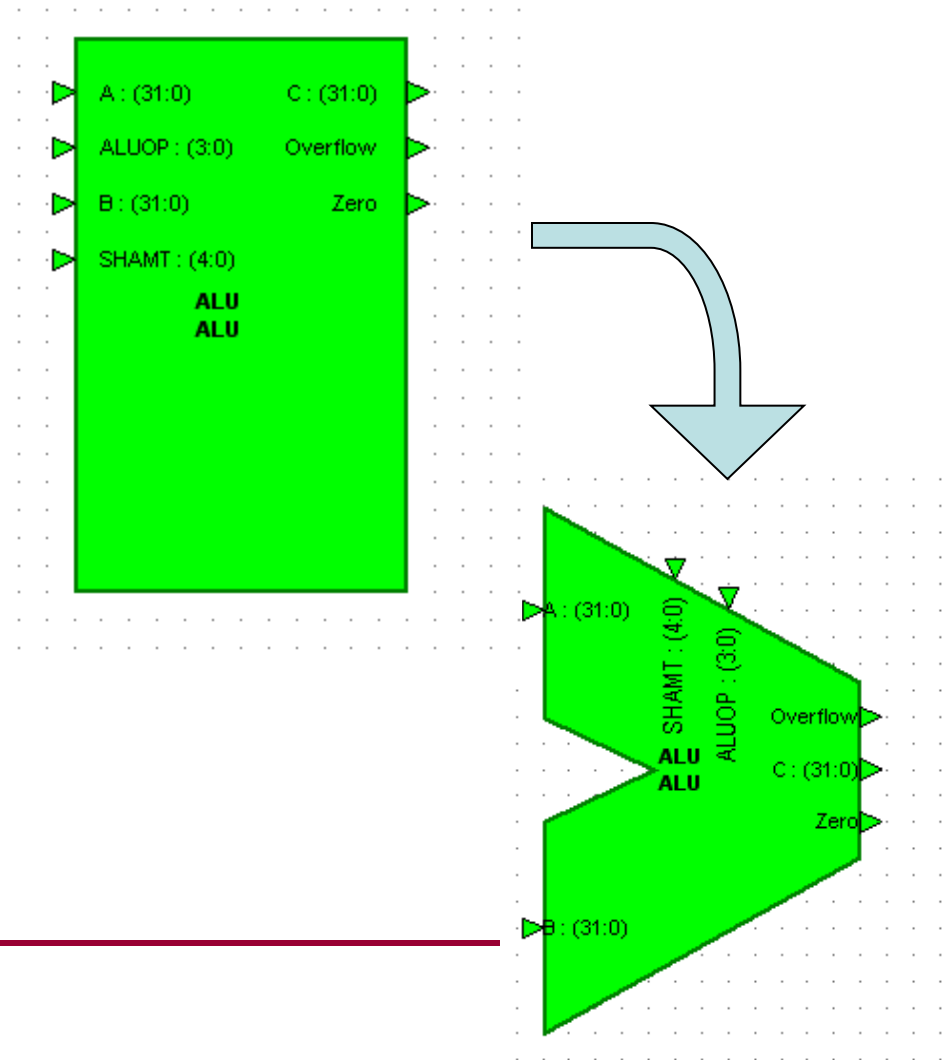
Return to Design Manager

- Save the block diagram into the ALU library
 - The *component* name will be “ALU”
- Let’s look at the ALU symbol...
 - Click “up” in BD, or
 - Use the design browser



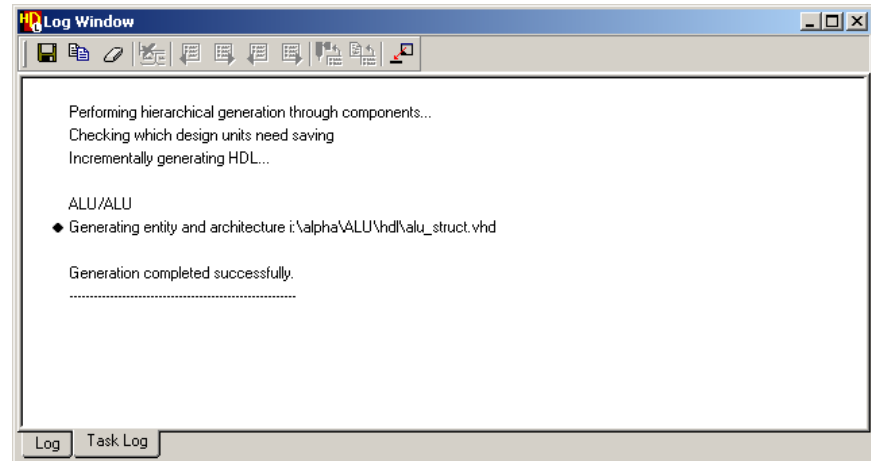
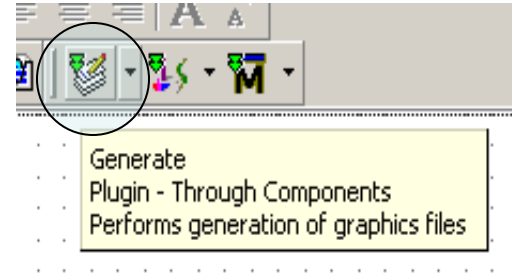
Symbol Editor

- The symbol looks something like this...
 - We can change the shape and pin locations here
 - Right click, then “Autoshapes”
 - Make the symbol look like an **ALU** symbol



VHDL Generation

- Go back to the block diagram window and let's generate VHDL for our design...
- Next, let's take a look at the VHDL that we generated...



Generated VHDL

```
-- hds header_start
--
-- VHDL Entity ALU.ALU.symbol
--
-- Created:
--   by - jbakos.UNKNOWN (TWEETY)
--   at - 11:19:31 01/07/03
--
-- Generated by Mentor Graphics' HDL Designer(TM)
2002.1a (Build 22)
--
-- hds header_end
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY ALU IS
  PORT(
    A      : IN      std_logic_vector (31 DOWNTO 0);
    ALUOP  : IN      std_logic_vector (3 DOWNTO 0);
    B      : IN      std_logic_vector (31 DOWNTO 0);
    SHAMT  : IN      std_logic_vector (4 DOWNTO 0);
    C      : OUT     std_logic_vector (31 DOWNTO 0);
    Overflow : OUT   std_logic;
    Zero   : OUT     std_logic
  );
-- Declarations

END ALU ;
-- hds interface_end

-- VHDL Architecture ALU.ALU.struct
--
-- Created:
--   by - jbakos.UNKNOWN (TWEETY)
--   at - 11:19:31 01/07/03
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2002.1a
(Build 22)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ARCHITECTURE struct OF ALU IS
  -- Architecture declarations
  -- Internal signal declarations

BEGIN
  -- Instance port mappings.

END struct;
```



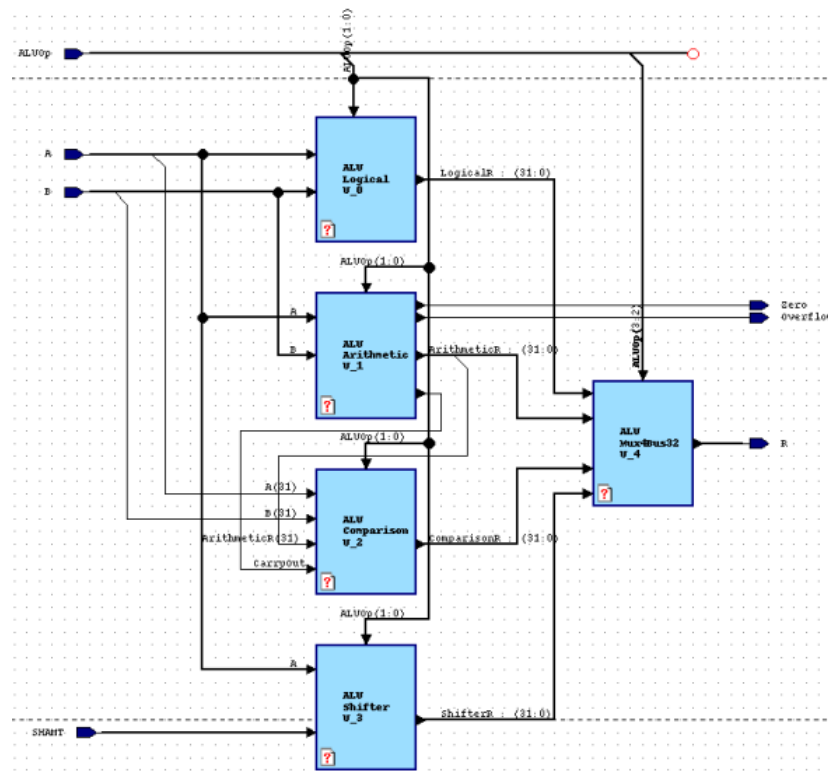
Control Signals

- Recall that the ALUOp is 4 bits
 - High-order two bits used to determine operation class (ALUOp(3:2))
 - Low-order two bits used to determine which operation is performed within each class (ALUOp(1:0))
- Next, let's define "operation classes" and have *subblocks* compute intermediate results in parallel...
 - Logical operations (ALUOp(3:2) == "00")
 - AND, OR, NOR, XOR
 - Arithmetic operations (ALUOp(3:2) == "01")
 - ADD, ADDU, SUB, SUBU
 - Comparison (ALUOp(3:2) == "10")
 - SLT, SLTU
 - Shift (ALUOp(3:2) == "11")
 - SLL, SRL, SRA
- Idea: perform each operation type in parallel and select the appropriate output using the two high-order bits of ALUOp



Subblocks

- Add subblocks, name them, and wire them up...
 - Note that ALUOp needs a bus ripper...



New Generated VHDL 1

- Let's take a look at the generated VHDL for the top-level design...
 - Things to note
 - Same entity statement as before
 - Internal signal declarations
 - Subblocks declared as components (with interfaces)
 - FOR ALL statements
 - Embedded block code
 - Instance port mappings



New Generated VHDL 2

```
LIBRARY ALU;
ARCHITECTURE struct OF ALU IS
  -- Architecture declarations
  -- Internal signal declarations
  SIGNAL ArithmeticR : std_logic_vector(31 DOWNTO 0);
  SIGNAL Asign       : std_logic;
  SIGNAL Bsign       : std_logic;
  SIGNAL CarryOut    : std_logic;
  SIGNAL ComparisonR : std_logic_vector(31 DOWNTO 0);
  SIGNAL LogicalR    : std_logic_vector(31 DOWNTO 0);
  SIGNAL Rsign       : std_logic;
  SIGNAL ShifterR    : std_logic_vector(31 DOWNTO 0);

  -- Component Declarations
  COMPONENT Arithmetic
  PORT (
    A      : IN      std_logic_vector (31 DOWNTO 0);
    ALUOp  : IN      std_logic_vector (1 DOWNTO 0);
    B      : IN      std_logic_vector (31 DOWNTO 0);
    ArithmeticR : OUT  std_logic_vector (31 DOWNTO 0);
    CarryOut : OUT  std_logic ;
    Overflow : OUT  std_logic ;
    Zero    : OUT  std_logic
  );
  END COMPONENT;
  COMPONENT Comparison
  PORT (
    ALUOp      : IN      std_logic_vector (1 DOWNTO 0);
    Asign      : IN      std_logic ;
    Bsign      : IN      std_logic ;
    CarryOut   : IN      std_logic ;
    Rsign      : IN      std_logic ;
    ComparisonR : OUT  std_logic_vector (31 DOWNTO 0)
  );
  END COMPONENT;

  COMPONENT Logical
  PORT (
    A      : IN      std_logic_vector (31 DOWNTO 0);
    ALUOp  : IN      std_logic_vector (1 DOWNTO 0);
    B      : IN      std_logic_vector (31 DOWNTO 0);
    LogicalR : OUT  std_logic_vector (31 DOWNTO 0)
  );
  END COMPONENT;
  COMPONENT Mux4Bus32
  PORT (
    ALUOp      : IN      std_logic_vector (3 DOWNTO 2);
    ArithmeticR : IN      std_logic_vector (31 DOWNTO 0);
    ComparisonR : IN      std_logic_vector (31 DOWNTO 0);
    LogicalR    : IN      std_logic_vector (31 DOWNTO 0);
    ShifterR    : IN      std_logic_vector (31 DOWNTO 0);
    R           : OUT  std_logic_vector (31 DOWNTO 0)
  );
  END COMPONENT;
  COMPONENT Shifter
  PORT (
    A      : IN      std_logic_vector (31 DOWNTO 0);
    ALUOp  : IN      std_logic_vector (1 DOWNTO 0);
    SHAMT  : IN      std_logic_vector (4 DOWNTO 0);
    ShifterR : OUT  std_logic_vector (31 DOWNTO 0)
  );
  END COMPONENT;

  -- Optional embedded configurations
  -- pragma synthesis_off
  FOR ALL : Arithmetic USE ENTITY ALU.Arithmetic;
  FOR ALL : Comparison USE ENTITY ALU.Comparison;
  FOR ALL : Logical USE ENTITY ALU.Logical;
  FOR ALL : Mux4Bus32 USE ENTITY ALU.Mux4Bus32;
  FOR ALL : Shifter USE ENTITY ALU.Shifter;
  -- pragma synthesis_on

```

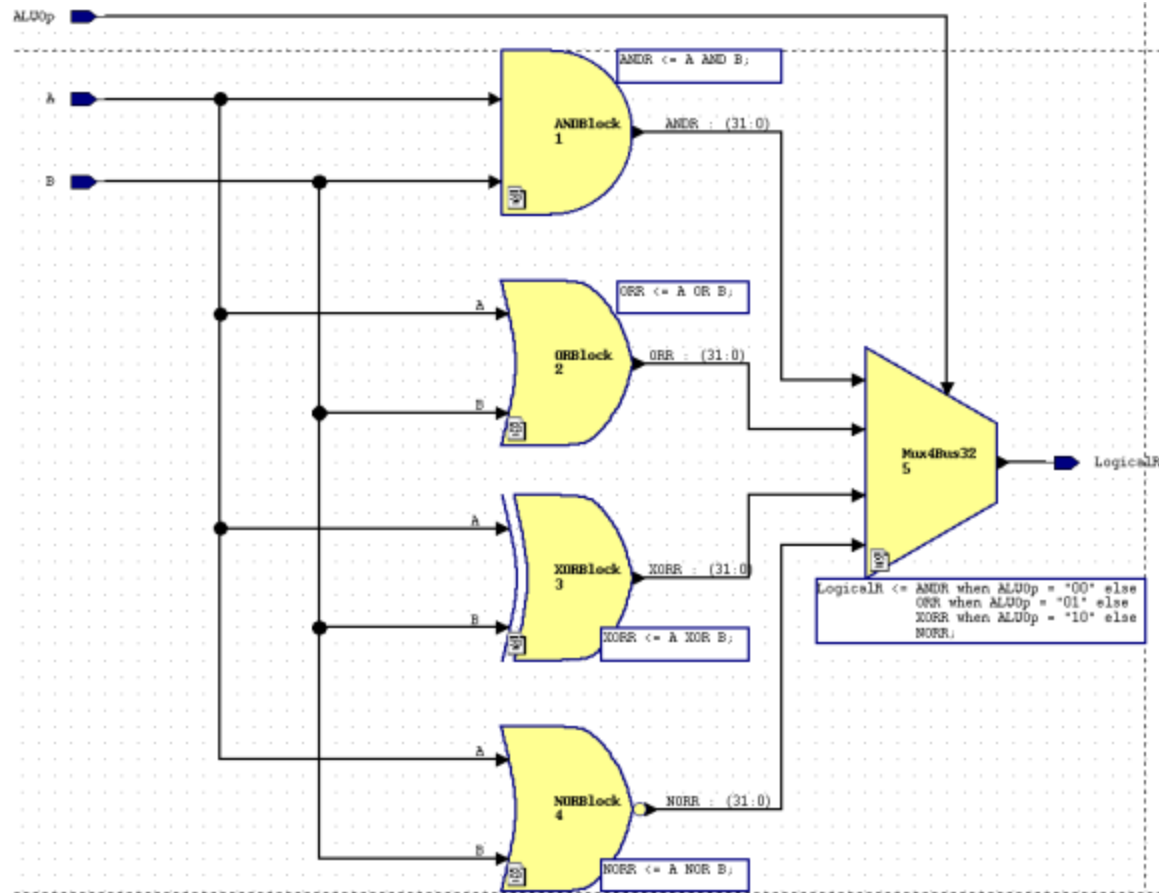


New Generated VHDL 3

```
BEGIN
  -- Instance port mappings.
  I1 : Arithmetic
    PORT MAP (
      A          => A,
      ALUOp      => ALUOp(1 DOWNT0 0),
      B          => B,
      ArithmeticR => ArithmeticR,
      CarryOut   => CarryOut,
      Overflow   => Overflow,
      Zero       => Zero
    );
  I2 : Comparison
    PORT MAP (
      ALUOp      => ALUOp(1 DOWNT0 0),
      Assign     => Assign,
      Bsign      => Bsign,
      CarryOut   => CarryOut,
      Rsign      => Rsign,
      ComparisonR => ComparisonR
    );
  I0 : Logical
    PORT MAP (
      A          => A,
      ALUOp      => ALUOp(1 DOWNT0 0),
      B          => B,
      LogicalR    => LogicalR
    );
  I4 : Mux4Bus32
    PORT MAP (
      ALUOp      => ALUOp(3 DOWNT0 2),
      ArithmeticR => ArithmeticR,
      ComparisonR => ComparisonR,
      LogicalR    => LogicalR,
      ShifterR    => ShifterR,
      R          => R
    );
  I3 : Shifter
    PORT MAP (
      A          => A,
      ALUOp      => ALUOp(1 DOWNT0 0),
      SHAMT      => SHAMT,
      ShifterR    => ShifterR
    );
END struct;
```



Logical Subblock Implementation



Logical Block Generated VHDL

- Let's take a look at the generated VHDL...

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

```
ENTITY Logical IS
  PORT(
    A      : IN      std_logic_vector (31 DOWNTO 0);
    ALUOp  : IN      std_logic_vector (1 DOWNTO 0);
    B      : IN      std_logic_vector (31 DOWNTO 0);
    LogicalR : OUT    std_logic_vector (31 DOWNTO 0)
  );
```

```
-- Declarations
```

```
END Logical ;
```

```
ARCHITECTURE struct OF Logical IS
```

```
-- Architecture declarations
```

```
-- Internal signal declarations
```

```
SIGNAL ANDR : std_logic_vector(31 DOWNTO 0);
SIGNAL NORR : std_logic_vector(31 DOWNTO 0);
SIGNAL ORR  : std_logic_vector(31 DOWNTO 0);
SIGNAL XORR : std_logic_vector(31 DOWNTO 0);
```

```
BEGIN
```

```
-- Architecture concurrent statements
```

```
-- HDL Embedded Text Block 1 ANDBlock
ANDR <= A AND B;
```

```
-- HDL Embedded Text Block 2 ORBlock
ORR <= A OR B;
```

```
-- HDL Embedded Text Block 3 XORBlock
XORR <= A XOR B;
```

```
-- HDL Embedded Text Block 4 NORBlock
NORR <= A NOR B;
```

```
-- HDL Embedded Text Block 5 Mux4B32
LogicalR <= ANDR when ALUOp="00" else
           ORR  when ALUOp="01" else
           XORR when ALUOp="10" else
           NORR;
```

```
-- Instance port mappings.
```

```
END struct;
```



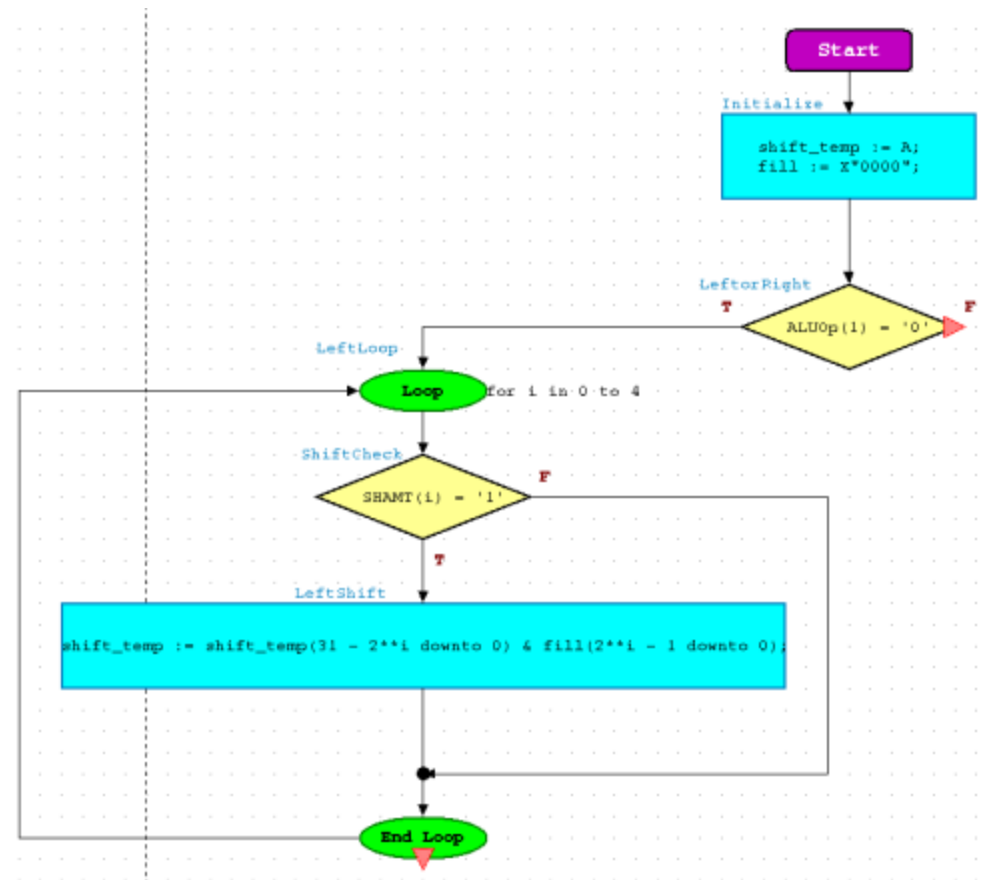
Modelsim Commands

- From this point, we can use force/run commands to simulate the design
 - Examples
 - `restart -f`
 - `view wave`
 - `add wave /ALU/A`
 - `force ALUOp "0010"`
 - `force A X"000000FF"`
 - `force A 10#32`
 - `run 10`
 - `run`
 - Default runlength is 100 ns
- Turn off warnings
- Note that the signals can be represented in hexadecimal
 - Right-click the signals in the wave window to change its properties
- We can also write a text ".do" file to aid in simulation
 - Invoked using "do" command
 - example:
 - `do test_logical.do`

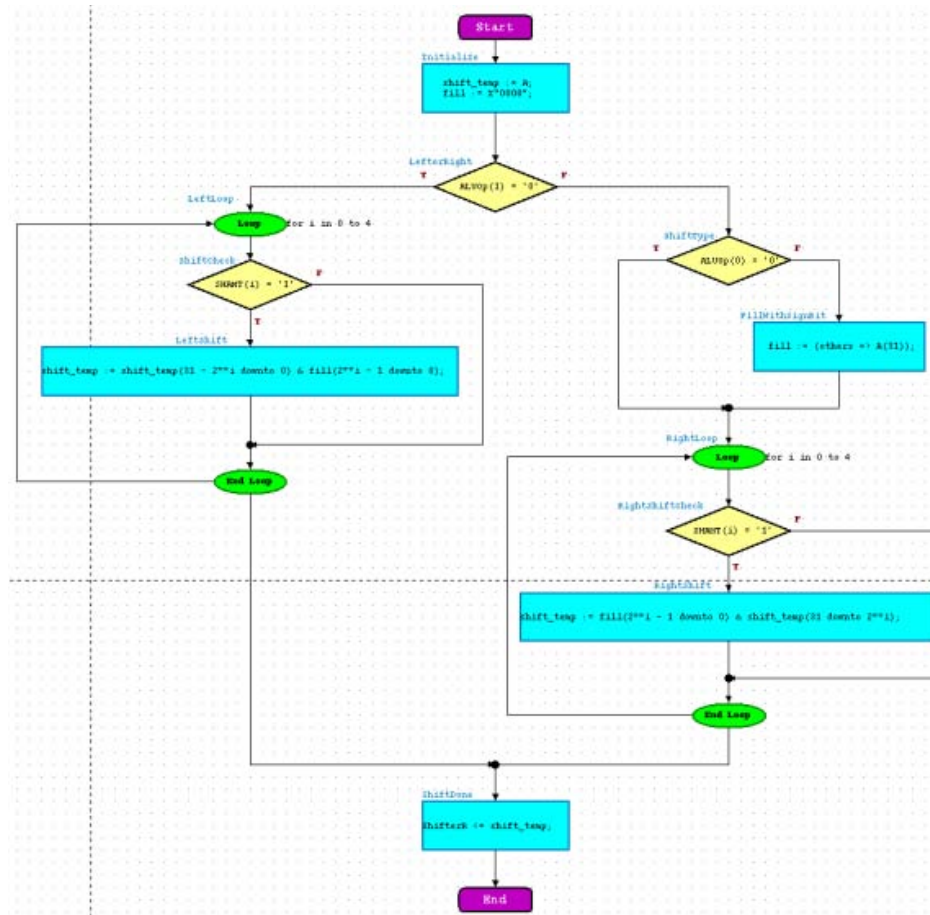


Shifter Subblock

- We'll implement the Shifter as a *flowchart* (useful for testbenches)
- Flowcharts implement a behavioral VHDL architecture as a *process*
 - Processes are executed sequentially, not concurrently
 - Started when signal in sensitivity list changes
 - Allows programming constructs and variables
- Primarily, we use:
 - Start/end points
 - Action boxes (also hierarchical)
 - Decision boxes
 - Wait boxes (*generally* not synthesizable)
 - Loop boxes
 - Flows
- Operations are assigned to ALUOp(1:0)
 - SLL => 00
 - SRL => 10
 - SRA => 11



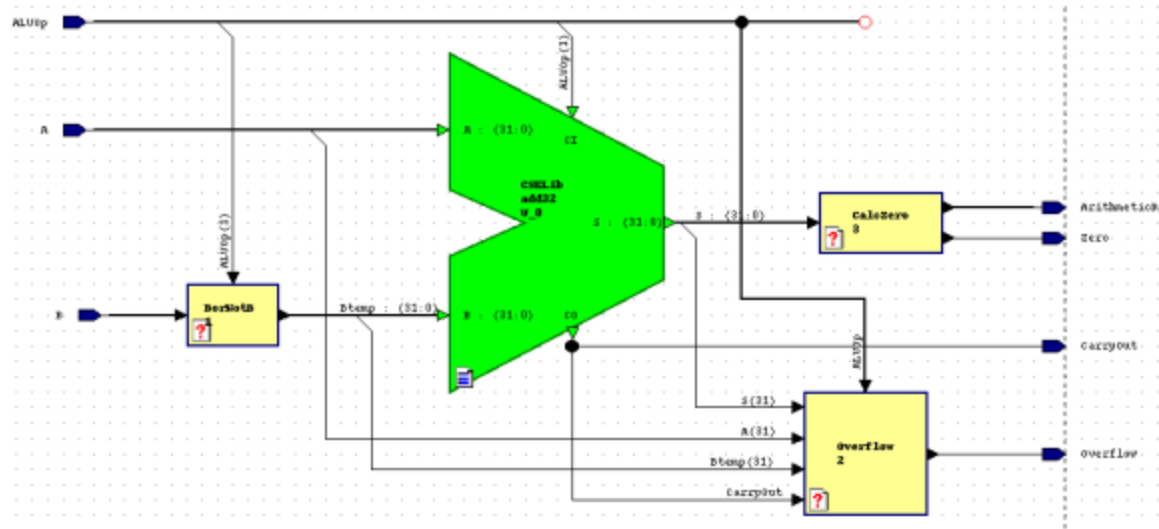
Shifter Subblock



Arithmetic Subblock

- Next, we'll design the arithmetic subblock as another block diagram
- We need to implement signed and unsigned addition and subtraction
- We have a 32-bit adder component in the COELib library we can instantiate for use in this design
- Use the green component button to add the ADD32 component from COELib
- Wire up the design as follows...

```
ArithmeticR <= S; Zero <= NOT(S(31) OR S(30) OR  
S(29) OR S(28) OR S(27) OR S(26) OR S(25) OR  
S(24) OR S(23) OR S(22) OR S(21) OR S(20) OR  
S(19) OR S(18) OR S(17) OR S(16) OR S(15) OR  
S(14) OR S(13) OR S(12) OR S(11) OR S(10) OR  
S(9) OR S(8) OR S(7) OR S(6) OR S(5) OR S(4) OR  
S(3) OR S(2) OR S(1)OR S(0));
```



Computing Overflow

ALU/Arithmetic/struct:Overflow (Truth Table)

File Edit View HDL Table Tasks Options Window Help

	A	B	C	D	E	F
1	ALUop	A(31)	B(31)	s(31)	carryout	overflow
2	"00"	'0'	'0'	'1'		'1'
3	"00"	'1'	'1'	'0'		'1'
4	"10"	'0'	'1'	'1'		'1'
5	"10"	'1'	'0'	'0'		'1'
6						'0'

Structure Navigator

- Signal Table
- Generics Table
- Arithmetic
 - Declarations/Statements
 - Embedded Diagrams
 - Overflow

Content

Content List Flow Help

Design "ALU/Arithmetic/struct" saved successfully.

Comparison Subblock

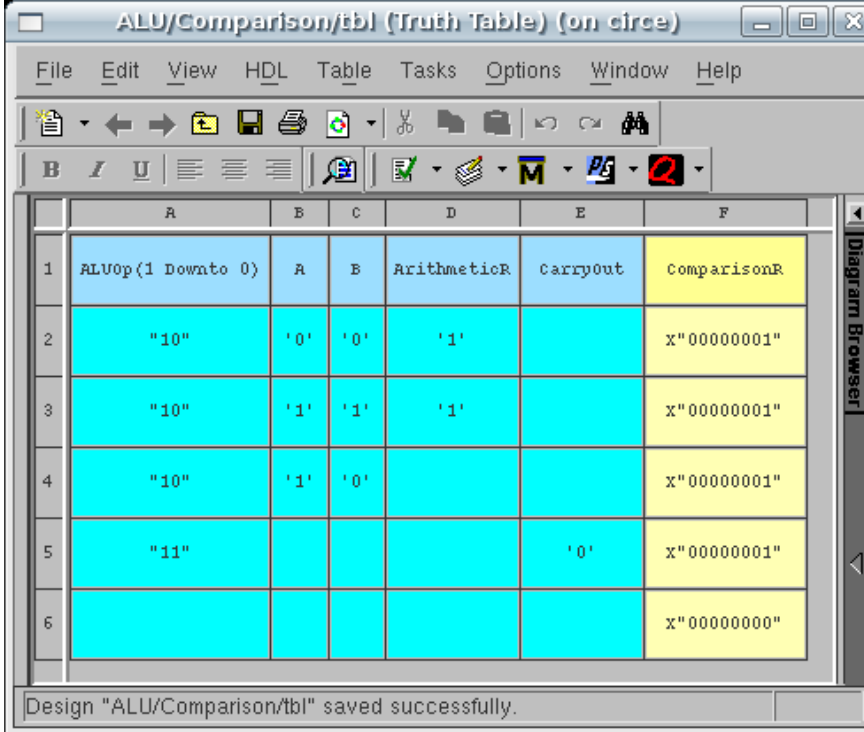
- Now let's design the comparison subblock using the *truth table* view
 - Implementing signed and unsigned “set-on-less than” ($A < B$)
 - We need to utilize the subtraction results from the arithmetic subblock as inputs to the table
 - Need to make sure the two low-order bits match those for subtraction in the arithmetic unit
 - SLT => 10
 - SLTU => 11
 - Outputs from arithmetic unit used as inputs
 - The sign of the result
 - Carryout
 - Other inputs we need
 - Sign of A
 - Sign of B
 - Output
 - Single bit output in low-order bit
 - Rows and columns can be added by a right-click
 - Columns can be resized
 - Note: blank cells are considered “don't cares”
 - Reminder: In VHDL, single bit literals (`std_logic`) are surrounded by single quotes, bit vectors (`std_logic_vector`) are surrounded by double quotes



Comparison Truth Table

Initial truth table view

- You will need to add rows
- You might want to reorder the columns



	A	B	C	D	E	F
1	ALUOp(1 Downto 0)	A	B	ArithmeticR	CarryOut	ComparisonR
2	"10"	'0'	'0'	'1'		x"00000001"
3	"10"	'1'	'1'	'1'		x"00000001"
4	"10"	'1'	'0'			x"00000001"
5	"11"				'0'	x"00000001"
6						x"00000000"

Design "ALU/Comparison/tbl" saved successfully.

Top-level Multiplexor Subblock

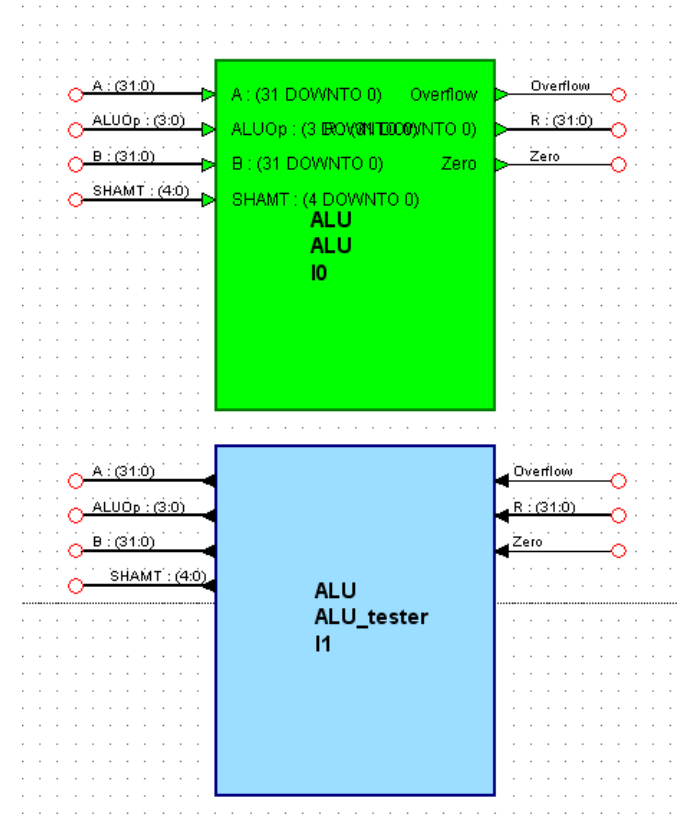
- Finally, let's finish the top-level ALU design by designing the implementation for the Mux4bus32
 - For this, we'll use a VHDL architecture/entity view
 - Note: the entity is not necessary in this view... it will be generated automatically anyway

```
R <= LogicalR when ALUOp(3 downto 2)="00" else  
    ArithmeticR when ALUOp(3 downto 2)="01" else  
    ComparisonR when ALUOp(3 downto 2)="10" else  
    ShifterR;
```



Testbenches

- “Harness” for a component
- Interface matching
 - Inputs \leftrightarrow Outputs
- Allows
 - Stimulation of input signals
 - **Output signal checking**
 - ASSERTs
 - Waiting/branching based on outputs
 - Debugging with waveforms
- Testbench component
 - Block diagram
- Tester component
 - Typically a flowchart



Testbenches

- Advantage over ad hoc methods
 - Ex. do-files
 - Allows simulation of inputs, but no output checking
 - Testbench code reveals interface specification and functionality (“self documenting”)
- Reproducible
 - Can use same testbench for multiple implementations/generations of a component
 - Can generate or utilize data file to share tests between simulation and hardware testing



Testbenches

- A *test procedure* is a methodology for testing a design
 - Sequence of steps
 - Testing aspects of a design's functionality
 - Example: ALU
 - Test each type of operation with different inputs
 - Along with asserting inputs/operations, can also verify correctness of output values
 - Also, use if-then-else semantics



Testbenches

- Facilities in HDL Designer
 - Easy creation of tester/testbench
 - Flowchart view is natural choice for implementing a test bench
 - Mirrors test procedure in a graphical representation
 - VHDL support for testbenches
 - ASSERT/REPORT/SEVERITY clause
 - Can use boolean operators here
 - Testbench operates in simulation



Testbenches

- Simple testbench example

- Drive inputs
- Wait for combinational logic
- Wait for clock edges
- ASSERT/REPORT/SEVERITY
- Repeat

