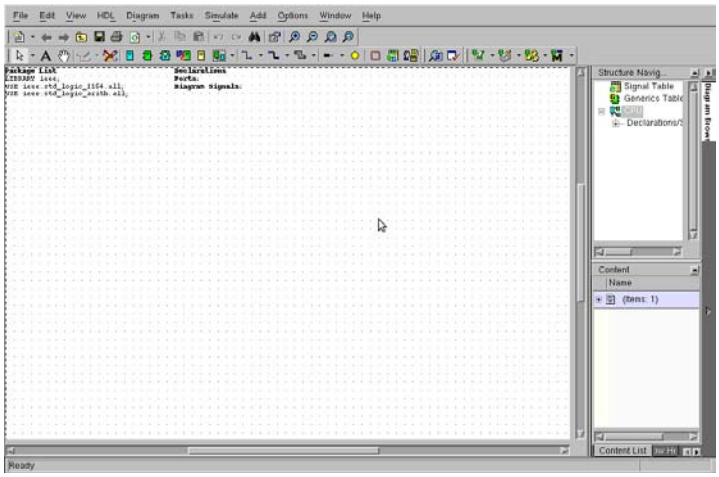
Getting Started with the CPU Design

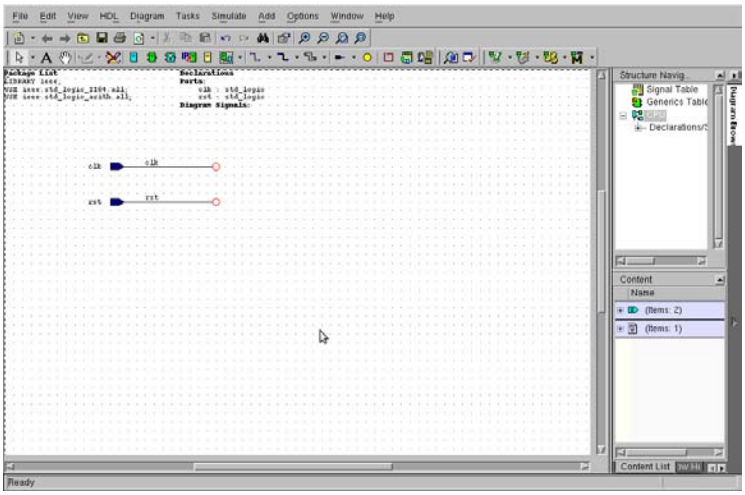
In this tutorial we will create a skeleton of your top-level computer and CPU. You may want to create a new library for these designs, but you may feel free to use your existing library (i.e. the library in which you designed your ALU).

Creating the Top-Level CPU Design

We will start out by creating an interface for your CPU, following the same procedure as we did for the ALU. Start out by creating a new, blank block diagram named “CPU”.

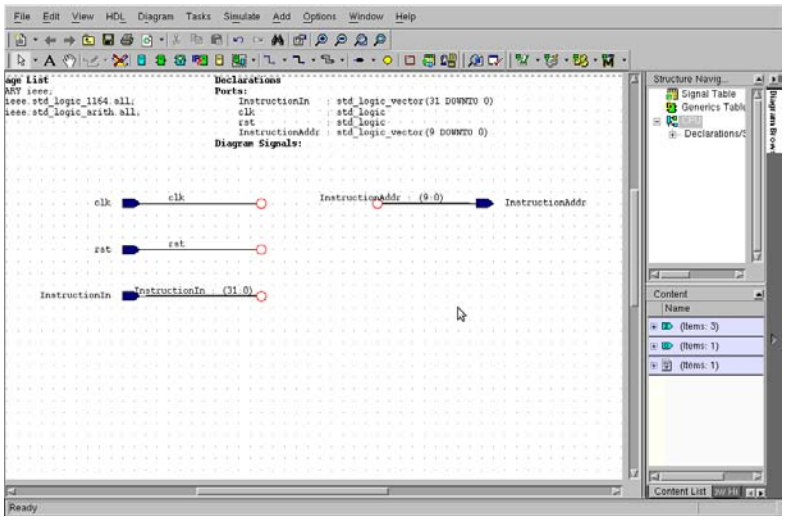


Next, let’s add inputs for clock and reset. Call them “clk” and “rst”. Both of these signals should be of type “std_logic”.



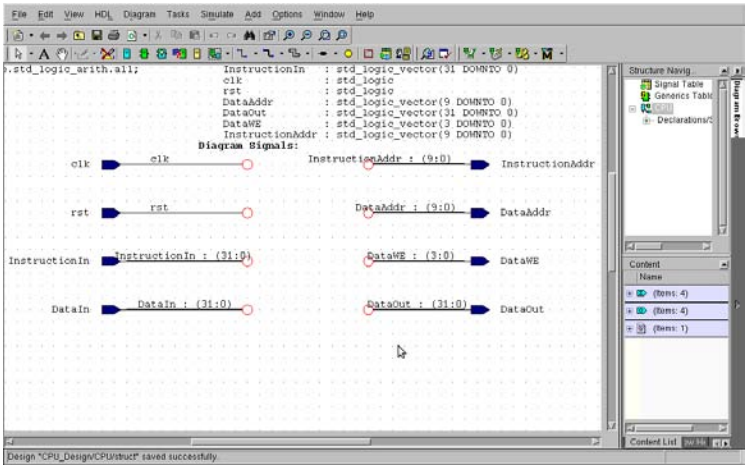
Next, let’s add the instruction memory interface signals. Recall that the instruction memory is read-only, so we only need an address output and a data input. In this case, our instruction memory will only have 1024 entries, so our address bus need only be $\log_2 1024 = 10$ bits wide. The data input will provide one instruction per clock cycle, so it will be 32 bits wide. We will

assume that the instruction memory will always be read enabled, so there's no need to add a control signal for reading. Name the 10-bit output signal "**InstructionAddr**" and name the 32-bit input signal "**InstructionIn**". Both signals are of type **std_logic_vector**.



Next, let's add the data memory interface signals. Since the data memory can both be written (for store instructions) and read (for load instructions), we'll need the ability to write. Recall that programs written in the MIPS instruction set usually perform loads and stores in units of the word size (in our case, 32 bits). However, the MIPS instruction set also provides half word and byte operations, we'll need the ability to write to individual bytes in the data memory. To do this, we'll need four separate write enable signals, corresponding to each of the four bytes within each 32-bit word.

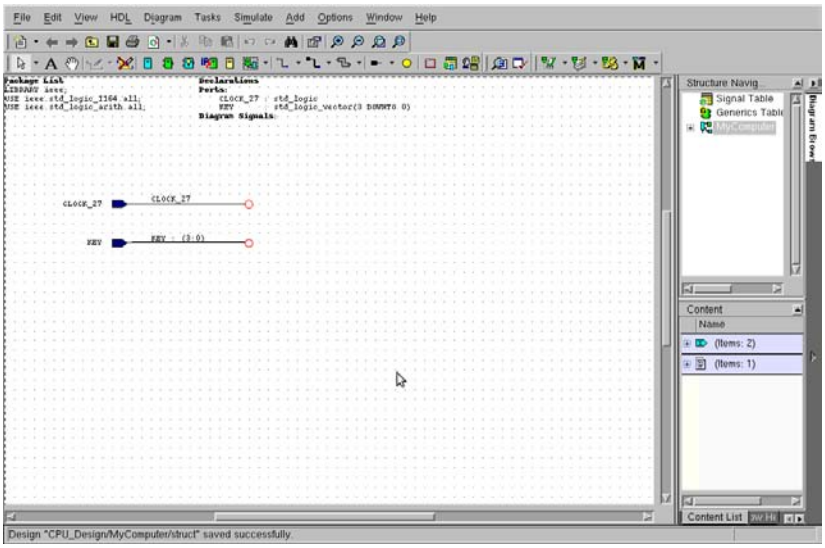
As with the instruction memory, we'll add a 10-bit address output (name it "DataAddr") and a 32-bit data input (name it "DataIn"). Unlike the instruction memory, we'll need a 32-bit data output (name it "DataOut") and the 4-bit byte write enable signal (name it "DataWE").



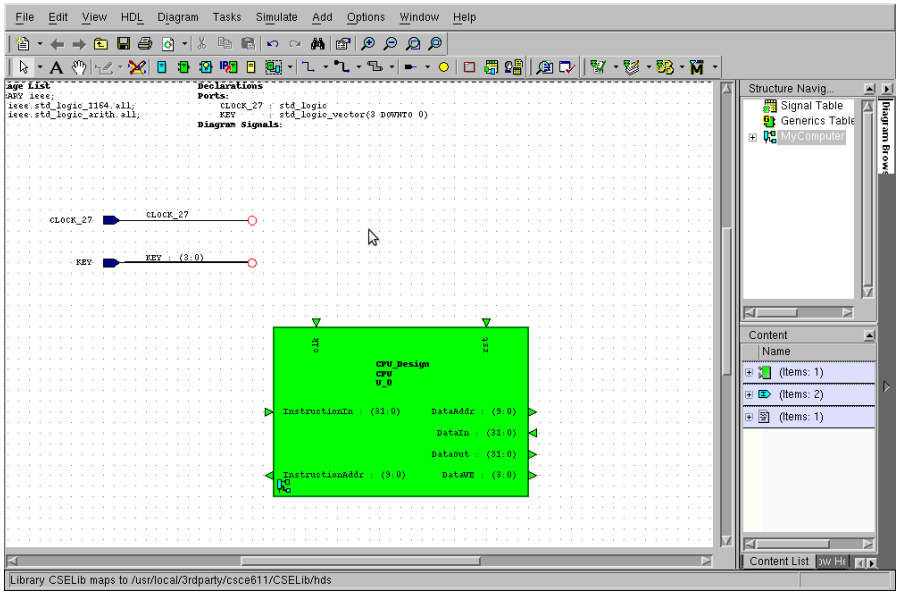
At this point our CPU's interface is complete. Save your design. Before continuing with our CPU design, we will now design our top-level computer design.

Creating the Top-Level Computer Design

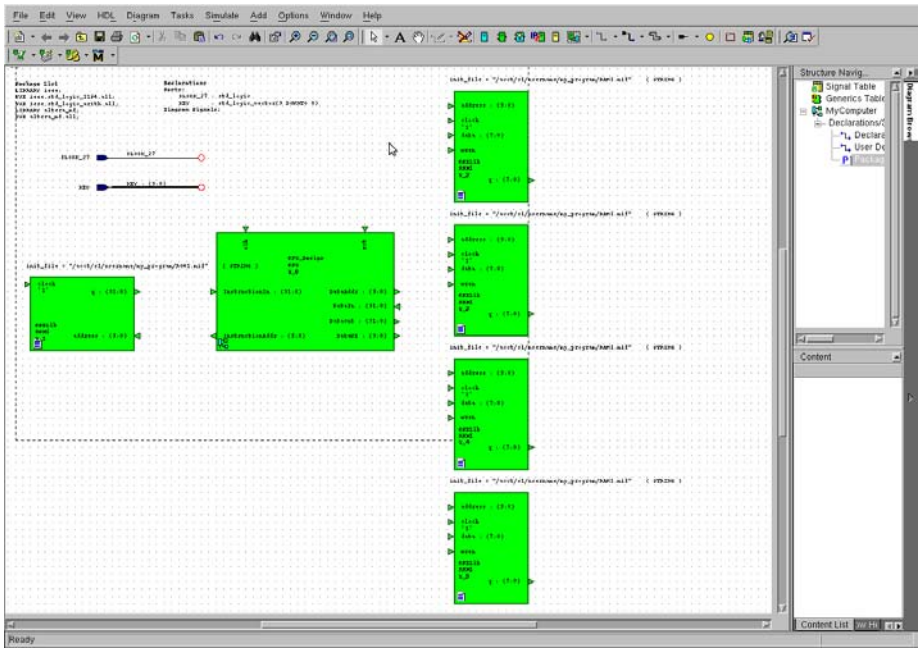
Create a new block diagram named "**MyComputer**". To bring in the clock and reset signals, create input ports of type std_logic named "CLOCK_27" and a signal of type std_logic_vector(3 downto 0) named "KEY" (we will use bit 0 of this vector to connect to the CPU's reset input via an inverter). These signal names correspond to signals that will eventually originate from the DE2 board's systems.



Next let's add our CPU to the design. Before you do this, you may want to edit the symbol for the CPU to give it a distinctive look. In my case, the CPU component is re-sized and the ports are moved around so the Instruction ports were on the left side, the data ports were on the right-side, and the clock and reset ports were on the top. Note that my CPU library is named "CPU_Design" and your library name may be different.



Now we'll instantiate our memories. In the CSELib, there are five memory models, named "ROM1" and "RAM1". ROM1 will be used for your instruction memory. Four instances of RAM1 will be used for your data memory, each corresponding to one of the four bytes of each word.



You'll notice that each of these memories has a text field above the symbol. This is called a **generic**, and this is the way that parameters may be set for instanced VHDL components. In this case, the parameters are being used to specify the path of the memory initialization file. For the ROM, this will be the file that contains the program code to be executed by the CPU. For the RAMs, this will contain any initial data for the program.

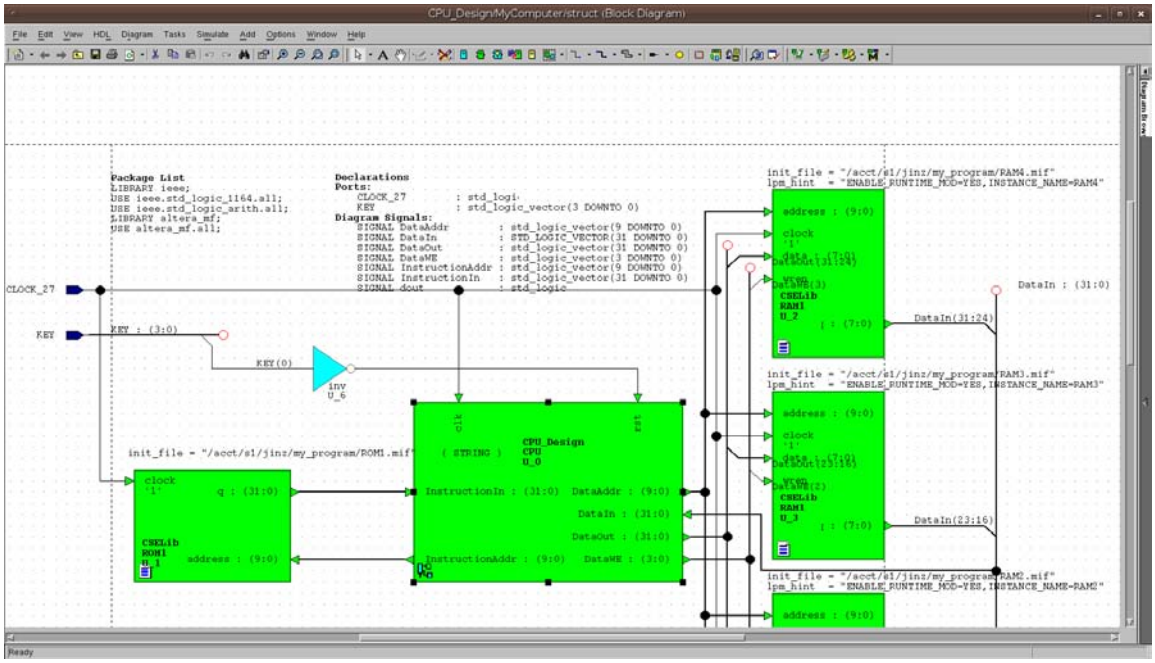
Double-click on each of these five generics to set its filename. First, replace "username" with your actual username. Second, replace "RAM1.mif" with "RAM2.mif" for the second RAM, "RAM3.mif" for the third RAM, and "RAM4.mif" for the fourth RAM (starting from the bottom and moving up).

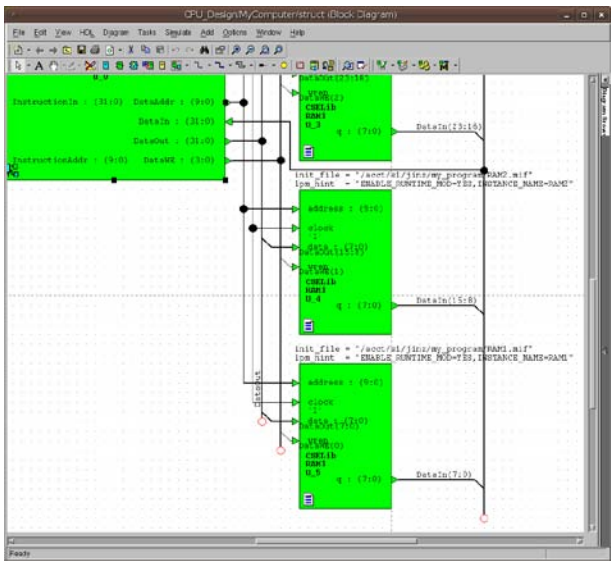
We also use the **generic** to give each memory component an instance name. Right-click on each of these component and rename RAM1 in

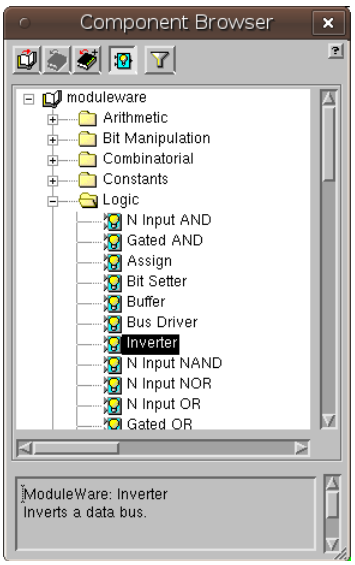
`lpm_hint = "ENABLE_RUNTIME_MOD=YES,INSTANCE_NAME=RAM1"`

with RAM1 for the first RAM, and RAM2 for the second RAM and so on.

Now we can interconnect the CPU with the pins and memory. The DataOut from the CPU must be split off to the data input of each RAM, and the output of each RAM must be merged to form the CPU's DataIn pin.







Notice that bus rippers are used to separate 8-bit segments of the 32-bit signal connected to the CPU DataOut pin in order to feed each of the RAM's data inputs. To merge all the 8-bit outputs from each of the RAMS, the RAM's data outs to the CPU's data in are connected by naming the bus signal "DataIn" and using bus rippers as well.

To add an inverter click the menu Add -> ModuleWare… and select inverter from the **Logic** list as shown in the right image above.

At this point we have finished with the top-level design. Next we'll generate a program and load it into our instruction ROM.