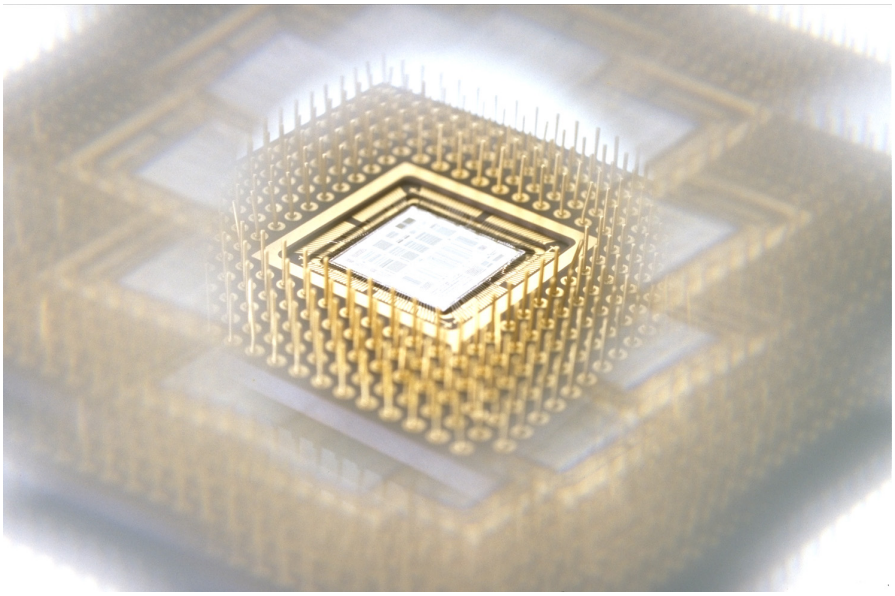
Avertec Tools

Yagle Tutorial



Software Release 3.4p5

June 7th, 2010

About this Document

This document explains:

- How to use Yagle
- Yagle functional abstraction principles
- Yagle usage demonstrated on small netlists

Documentation issued and compliant with Avertec Tools Release 3.4p5.

Please contact support@avertec.com for comments relating to this manual.

Table of Contents

Chapter 1. Introduction

1.1. Directories Description

The tutorial and the related files can be found in the following directory:

```
$AVT_TOOLS_DIR/tutorials/yagle
```

Among the directories presents in `$AVT_TOOLS_DIR/tutorials/yagle`, this tutorial will use the following ones:

- `addaccu/`: For the addaccu example
- `comb/`: For the combinational example
- `glitcher/`: For the glitcher example
- `rom/`: For the ROM example
- `shifter/`: For the shifter example

Along the tutorial the user is expected to change directory as needed to perform the operations related to each specific example.

The technology file used during the course of the tutorial is `bsim4_dummy.hsp` located in:

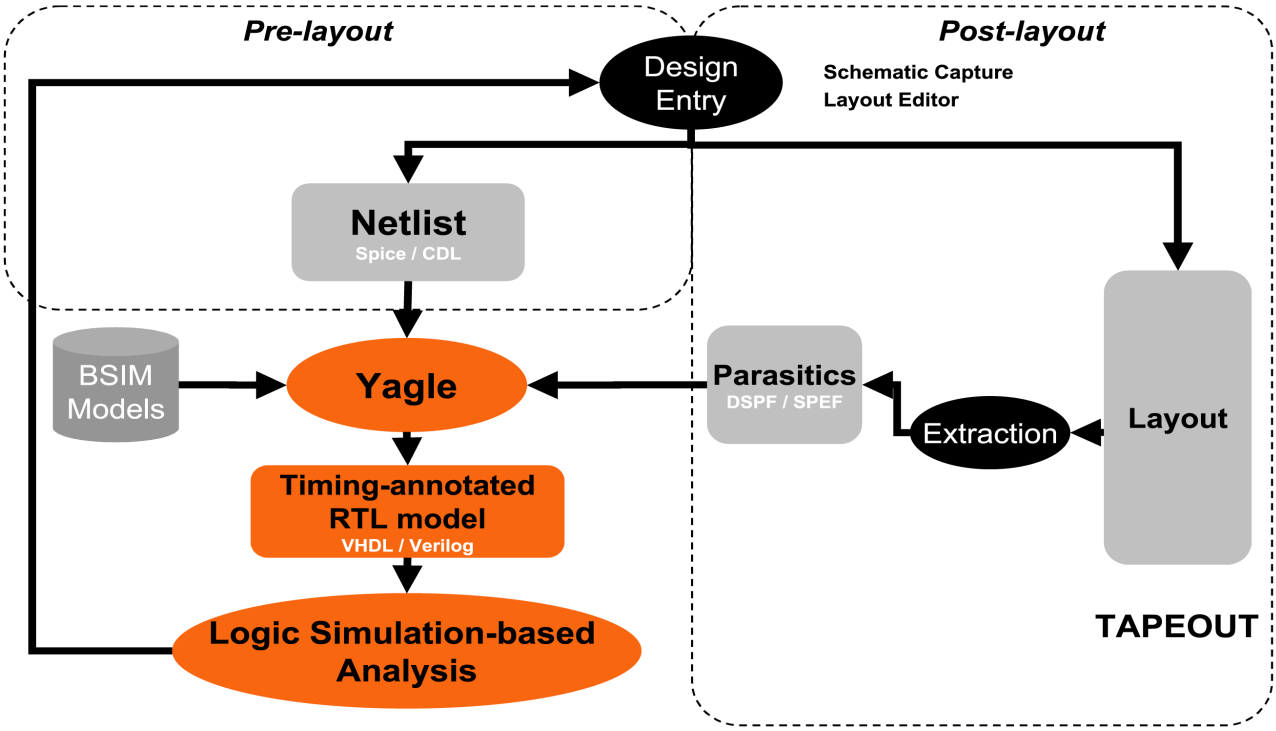
```
$AVT_TOOLS_DIR/tutorials/techno/bsim4_dummy.hsp
```

1.2. Tool Description

Yagle is an automatic transistor-to-RTL functional abstractor, which automatically handles any kind of digital circuitry (CMOS and NMOS, pass-transistor logic, transmission gate logic, dynamic logic) and automatically detects and models latches and registers. Yagle generates industry-standard VHDL or Verilog, with a Spice-accurate timing back-annotation.

1.3. Integration Flows

Yagle integrates in the most common design flows, as illustrated in the following diagram:



1.4. Tool Setup and Execution

1.4.1. Netlist Files

The netlist can be fed into Yagle in various formats, at different levels of hierarchy:

- Spice/CDL flat transistor
- Spice/CDL hierarchical
- Hierarchical Structural VHDL
- Structural Verilog
- Parasitics in Spice, DSPF, SPEF

1.4.2. Execution

Yagle functionalities are provided through a set of functions, that can be accessed through Avertec's Tcl interface: `avt_shell`.

Tool configuration is done with variables. The value of each variable can be set in the Tcl script by the `avt_config` function.

1.4.3. Output Files

Yagle generates the following output files:

- CNS/CNV: intermediate disassembled netlist
- REP: report file

- VHDL/Verilog: abstracted behavioral model
- COR: VHDL/Verilog to CDL/Spice correspondence file

1.4.4. Yagle GUI: XYagle

The XYagle GUI is invoked in the following way:

```
> xyagle &
```

XYagle is mainly used to browse the CNS intermediate disassembled netlist. For further information, please refer to the Yagle reference manual.

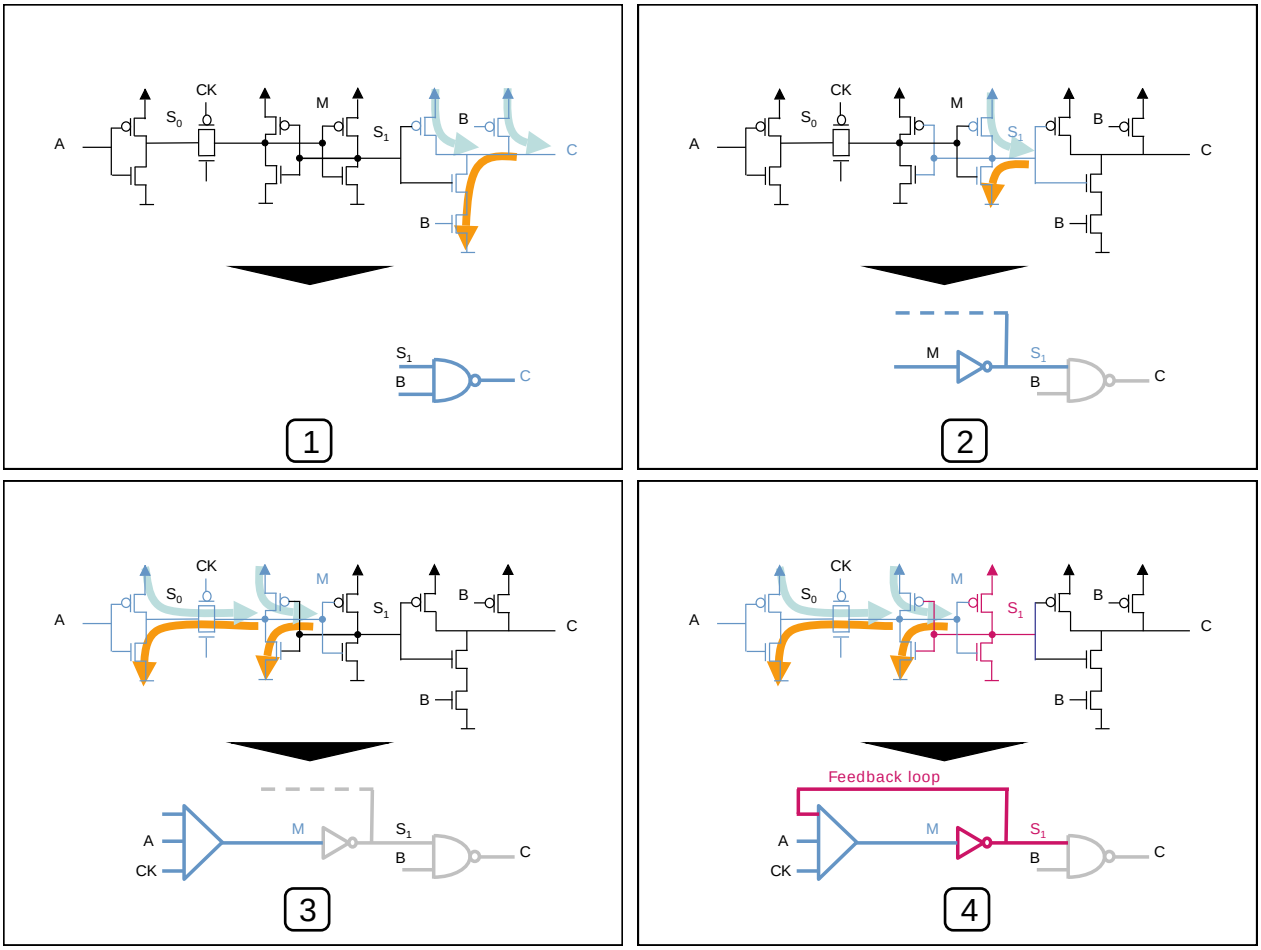
Chapter 2. Disassembly

2.1. Cone Mapping

The base object of the methodology is the "cone". Disassembly can be seen as a conversion of a network of non-oriented transistors into a network of oriented cones.

The starting point of any partitioning strategy is the identification of the nodes on which we intend to build a sub-network. The idea of Yagle disassembly is to build sub-networks between which there is no charge transfer. Therefore, the signals controlling transistor gates define the interface between two sub-networks, and the nodes for which a sub-network is extracted during the partitioning are the nodes driving at less one transistor gate.

The extracted sub-networks are called cones. A cone is a DC-connected object: it contains all the paths that link the node to a voltage source through the source-drain connections of the transistors.



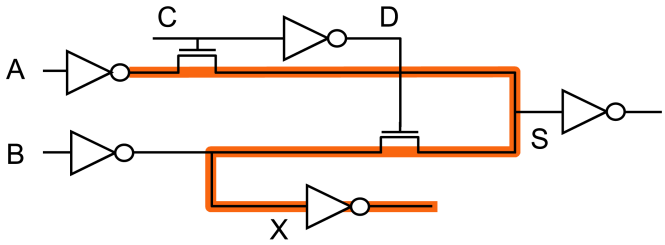
Each cone has a unique output and a certain number of inputs: the signals controlling the gates of the cone's transistors.

The construction of a cone on a node N consists in identifying all the current paths between the node N and a voltage source (Vdd or Vss). We call a "branch" a path that links the node N to a voltage source.

2.2. False Branches Detection

The symmetric nature of MOS transistors has a significant impact on the construction of branches. Without knowing the transistors orientation, we must construct all the paths towards the voltage sources. It is possible that the correlations on the signals controlling the gates block some current paths. We call those current paths "false branches".

As we can see in the next figure, it is the logical context -i.e. the correlations between the inputs of the cone- that allows to establish rigorously the transistors orientation. This logical context also allows the elimination of false branches.

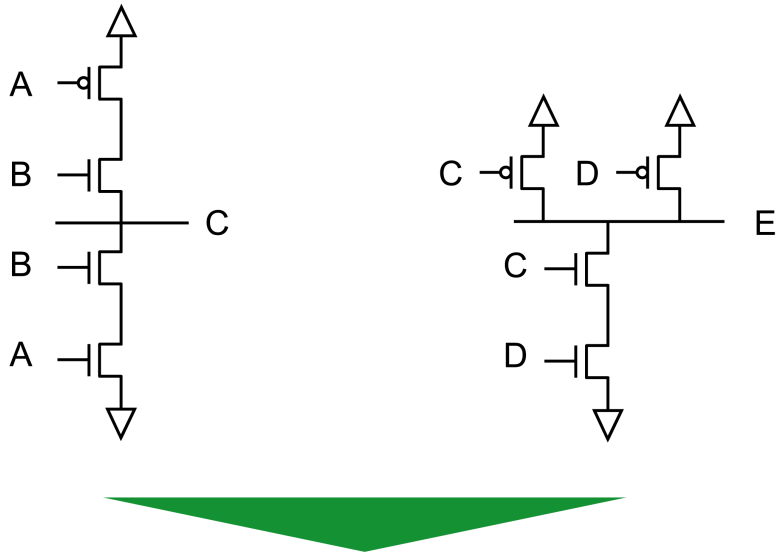


Chapter 3. Behavior Generation

3.1. Functional Characterization

The functional characterization of a cone is computed from the branch structure. A Boolean expression is generated for all the branches leading to Vdd. It gives the set condition of the node. Another expression is constructed for all the branches leading to Vss. It gives the reset condition of the node.

Each branch is considered as a chain of switches. For a N-Channel transistor, the switch is off when the gate signal is high. For a P-Channel transistor, the switch is off when the gate signal is low.



C_{up} = not A and B
C_{dn} = A and B

E_{up} = not C or not D
E_{dn} = C and D

For the signal E, the set condition is Eup, the reset condition is Edn. With those two expressions, it is possible to analyze the functionality of the cone. We must verify their orthogonality and their completeness.

If the two expressions are orthogonal, it exists no combination of the inputs for which a Vdd branch is active simultaneously as a Vss branch, i.e. if Eup+Edn = 0, the cone is non-conflictual. If the two expressions are complete, it exists no combination of the inputs for which no branch is active, i.e. if Eup+Edn = 1, the cone is non-HZ.

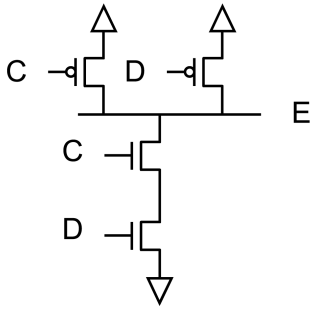
A cone that respects the orthogonality and completeness conditions is said to be CMOS DUAL.

For example, as Eup and Edn respect those conditions, they can then be grouped into a single expression on E: E = not C or not B

Cup and Cdn do not respect orthogonality and completeness conditions, then the up and down conditions remain separated.

3.2. Timing Back-annotation

The timing characterization of a cone is also computed from the branch structure, and then it perfectly maps on the functional characterization. Let's take the cone E of the following figure as an example.

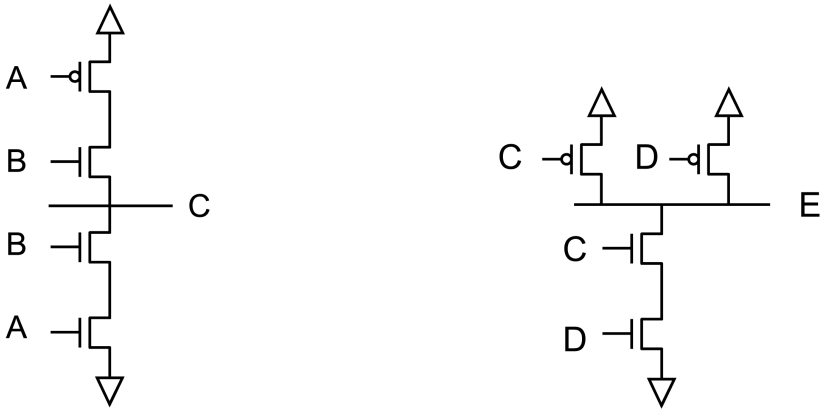


A propagation time is associated to a couple of signals: the output E of the cone, and one of its inputs (C or D). Actually, the switching of C or D does not always lead to the switching of E: the rising of C leads to the falling of E only if D has the logical value 1.

The computation of the propagation time CE is made under the hypothesis that the switching of the E is induced par the switching of C.

The value of the propagation time is then computed by a mixed analytical-numerical method, based on the IV curves and capacitances of the MOS transistors BSIM3/BSIM4 technology models.

Each possible propagation of a input transition towards a transition on the output is characterized; the cones are then characterized as follow:



A down _ C up		30ps
B up _ C up		20ps
A up _ C down		40ps
B up _ C down		30ps

C down _ E up		10ps
D down _ E up		15ps
C up _ E down		30ps
D up _ E down		40ps

3.3. VHDL and Verilog Description

The VHDL|Verilog characterization is done by translating the behavioral models of the cones into VHDL|Verilog syntax:

C_{up} = not A and B
C_{dn} = A and B

E_{up} = not C or not D
E_{dn} = C and D

```
E <= not C or not D;
```

```
process (A, B)
begin
        if (A = '0' and B = '1') then
                C <= '1';
        elsif (A = '1' and B = '1') then
                C <= '0';
        else
                C <= 'Z';
        end if;
end process;
```

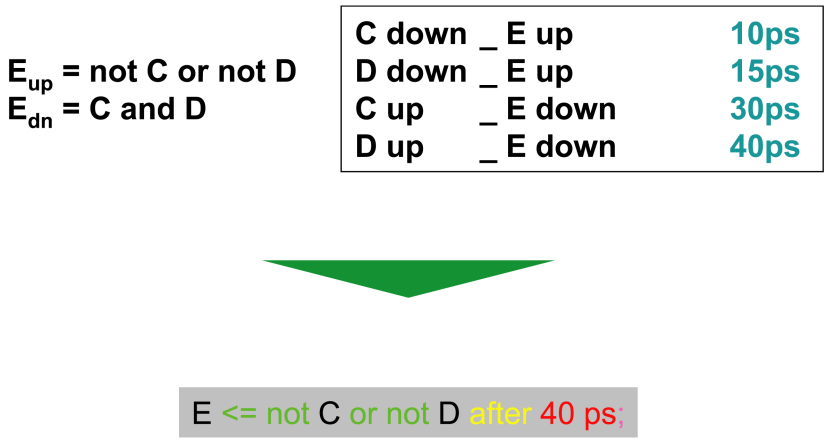
For CMOS DUAL cones, a single signal assignment is generated, for example for the cone E as demonstrated in the first column (left), for non-CMOS DUAL cones, a process is built, listing all the possible assignations as it can be seen in the second column (right).

The final VHDL|Verilog is generated by mapping the timing characterization on the functional characterization.

For CMOS DUAL cones, three levels of precision are available. Let's take the cone E as an example: in term of behavior, the reduction of the expressions Eup and Edn to a single expression E = not C or not D is lossless. It is not the case in terms of timing: there are different timings associated with each expression and with each event occurring on a variable of the expression.

First level of precision:

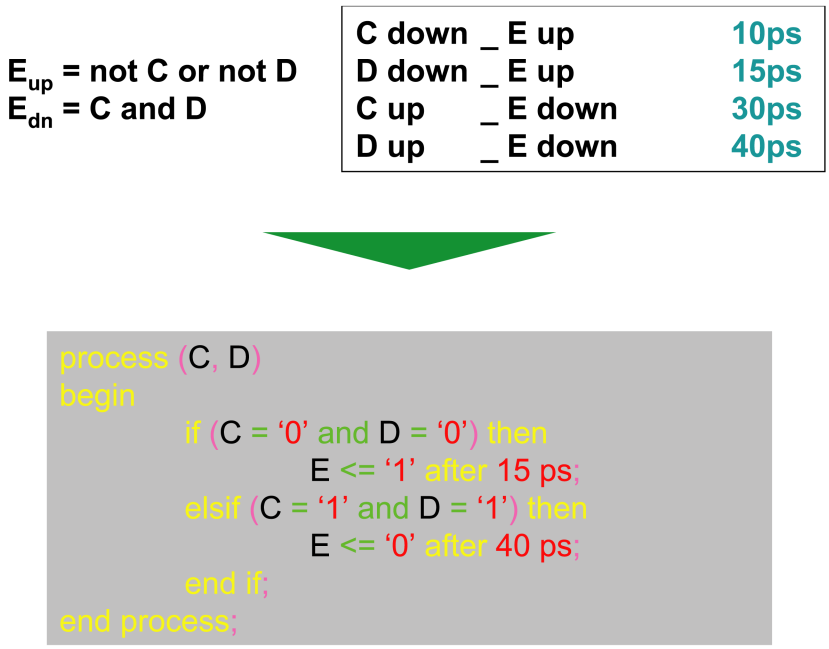
If we intend to keep a compact VHDL|Verilog (i.e. the expression E = not C or not D), we must choose one timing among all the different timings characterizing the cone E. Typically, in Yagle, it is possible to choose between the maximum timing, the minimum timing, and the average timing. With choosing the maximum timing, this first level of precision leads to the following expression:



Second level of precision:

Second and third levels of precision are obtained through splitting the expression E into Eup and Edn expressions. The second level of precision does not take into account the events on the variables of the expressions. The maximum, minimum or average timing can be chosen for up and down expression.

With choosing the maximum timing, this second level of precision leads to the following expression:

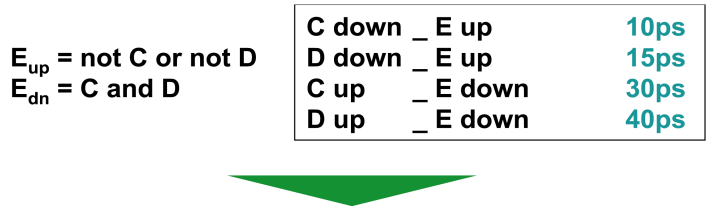



```
process (C, D)
begin
        if (C = '0' and D = '0') then
                E <= '1' after 15 ps;
        elsif (C = '1' and D = '1') then
                E <= '0' after 40 ps;
        end if;
end process;
```

Third level of precision:

The third level performs the complete timing characterization of the cone. A timing is associated with each event of each variable of each expression of the cone:

For non-CMOS DUAL cones, as the up and down expressions are not reduced to a single expression, only the second and third levels of precision are available.




```
process (C, D)
begin
        if (C = '0' and D = '0' and C'event) then
                E <= '1' after 10 ps;
        elsif (C = '0' and D = '0' and D'event) then
                E <= '1' after 15 ps;
        elsif (C = '1' and D = '1' and C'event) then
                E <= '0' after 30 ps;
        elsif (C = '1' and D = '1' and D'event) then
                E <= '0' after 40 ps;
        end if;
end process;
```

Chapter 4. Simple Combinational Example

4.1. Comb Directory

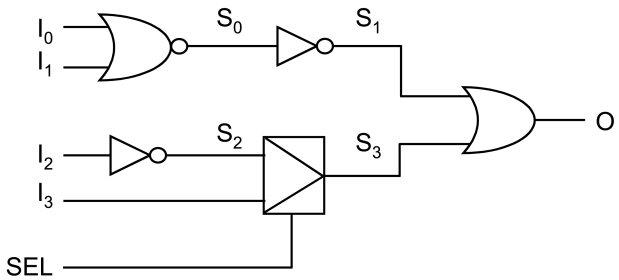
The files and scripts needed for this example can be found in:

```
$AVT_TOOLS_DIR/tutorials/yagle/comb/
```

4.2. Comb Design

This example intends to illustrate the concepts previously described (functional abstraction mechanisms, VHDL|Verilog generation and timing back-annotation) and to give a first glance at Yagle's setup and execution modes.

This example is based on the combinational design below:



This design contains basic CMOS gates (nor, inverters, or) and a multiplexer based on pass-transistors. The design is described in the file comb.spi as a flat transistor netlist.

4.3. Input Files

In order to perform the functional abstraction of the circuit, Yagle needs the following files:

- `comb.spi`: the design itself
- `bsim4_dummy.hsp`: the technology models of the transistors used in the design. Transistors' technology models are essential to compute delays. If the desired behavioral model (the VHDL) should not be back-annotated with timings, this file can be omitted. However, in such a case the tool needs to know the names of the transistor models. Those names should be set in the script with the `avtSpiTnModelName` and `avtSpiTpModelName` variables.
- `run.tcl`: the Tcl script performing the setup and execution of the tool.

4.4. Basic Execution and Output Files

The configuration set in the `run.tcl` script is sufficient to perform a first functional abstraction of COMB. The only variables that need to be set are the names of the alimentations (VDD and VSS). Lines preceded by a '#' are commented. We will see their meaning in the following sections. With this given configuration file, the tool is invoked as follow:

```
> run.tcl
```

The steps of the functional abstraction process are displayed on the standard output. It should have this appearance:

This Yagle run generates two files:

- `comb.vhdl`: the VHDL file resulting of the functional abstraction of COMB.
- `comb.rep`: the report file of the functional abstraction run. This file contains information about the run, warnings and errors that may occur during it. This file should be consulted after each run of Yagle. In the present case, the design being very simple, the file is empty.

It is also possible to generate a verilog file by adding the following line in the Tcl script:

```
avt_config avtOutputBehaviorFormat vlg
```

4.5. Primary Options and Configuration

4.5.1. CNS File Generation

As previously explained, the first step of the functional abstraction process is what is called disassembly. Disassembly is the partition of the original design into cones: it is the conversion of the netlist of non-oriented transistors into a netlist of oriented cones. In the basic execution mode, this netlist exists only as a data-structure in the program's memory, and is hidden from the user. It can be dumped on disk as an ASCII file by positioning the following variable in the Tcl script:

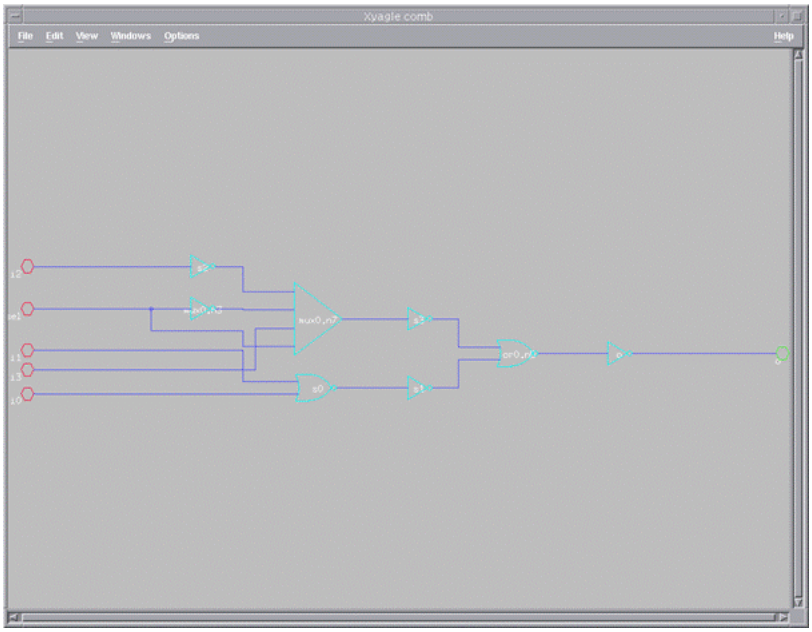
```
avt_config yagleGenerateConeFile yes
```

The programs then generates a CNS file (Cone Netlist Structure): `comb.cns`. For disk saving purposes, this file is compact, and therefore quite uneasy to read. It is possible to obtain a more verbose file by positioning the `avtVerboseCone` variable to `yes` in the Tcl script. The programs will then generate a CNV file (Cone Netlist Verbose): `comb.cnv`.

Both files formats syntaxes are documented in the Yagle User Guide. Browsing the Cone Netlist Structure is far more convenient using the XYagle GUI.

```
> xyagle &
```

Selecting the file `comb.cns` in the dialog box will lead open the following window:



4.5.2. Disabling VHDL and Verilog Generation

For debugging purposes, it is also possible to disable the VHDL and Verilog generation. This can be done by positioning the following variable in the script:

```
avt_config YagleGenerateBehavior no
```

Yagle will stop after the disassembly process.

4.5.3. Timing Back-annotation

In basic execution mode, the behavioral model (VHDL/Verilog) is generated without timing back-annotation. In order to perform this back-annotation, the first step is to include a technology file in the netlist to be abstracted. Here, we will use the file `bsim4_dummy.hsp`, which contains BSIM3 technology parameters of two models of transistors, named TN and TP.

```
avt_LoadFile ../techno/bsim4_dummy.hsp spice
```

As explained in the timing back-annotation relevant section, timing back-annotation supports three levels of precision.

First level of precision:

In basic execution mode, the back-annotation is made with the first level of precision. Timing back-annotation is invoked by setting the following variable:

```
avt_config yagleTasTiming tdmin|tdmed|tdmax
```

For CMOS DUAL cones, the `tdmin`, `tdmed` or `tdmax` directive selects the type of timing to be applied on the cone. For non-CMOS DUAL cones, this directive selects the type of timing to be applied on the up and down expressions of the cone.

Second level of precision:

This level is selected by setting the `yagleSplitTimingRatio` variable to a value V greater than one: if delays associated with up and down output transitions differ by a ratio greater than V, then up and down transitions are differentiated.

This variable allows to split the expression of the CMOS DUAL cones into up and down expressions, and to associate them minimum, average or maximum timing, according to the `tdmin`|`tdmed`|`tdmax` option.

Third level of precision:

This level is selected by setting the `yagleSensitiveTimingRatio` variable to a value V greater than one. For a given output transition (up or down), if delays associated with different input transitions differ by a ratio greater than V, then the delays are differentiated by input.

Chapter 5. Glitcher Example

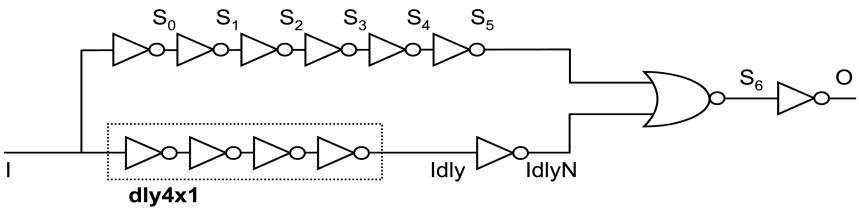
5.1. Glitcher Directory

The files and scripts needed for this example can be found in:

```
$AVT_TOOLS_DIR/tutorials/yagle/glitcher/
```

5.2. Glitcher Design

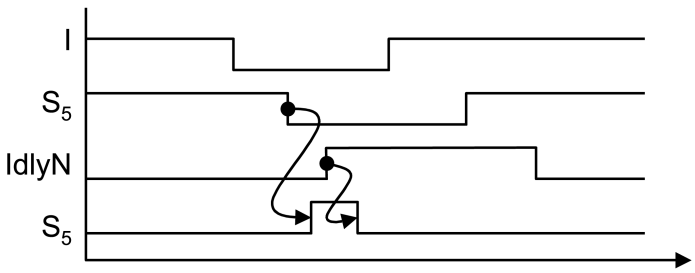
This example intends to illustrate the utility of timing back-annotating an abstracted behavioral model. In this lab, we are going to perform the abstraction of the glitcher design described below.



We will perform functional abstraction with and without timing back-annotation. We will see from simulation results of the behavioral models, that they do not present the same behavior. Actually, we will see that the zero-delay model simply doesn't work.

5.3. Normal Operating

The glitcher design includes a delay cell (`dly4x1`), in order to render the path going through the signals Idly and IdlyN longer than the path going through the signals `S0`, `S1`, `S2`, `S3`, `S4`, `S5`. The operating timing diagram is then as follow:



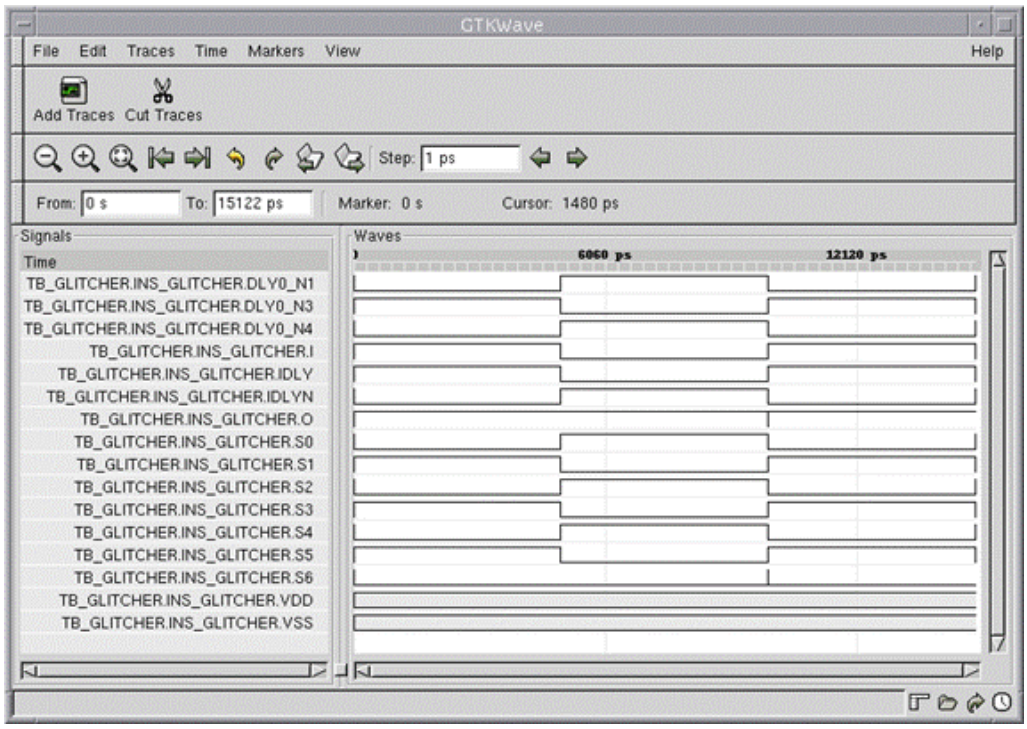
The glitch on `S6` is implied by the falling transition of `I`.

5.4. Zero-delay Model

The zero-delay behavioral model is obtained by running the command:

```
> run.tcl
```

Yagle generates the `glitcher.vhd` file. The result of the simulation is saved in the `glitcher.vcd` file, displayed in the following window.

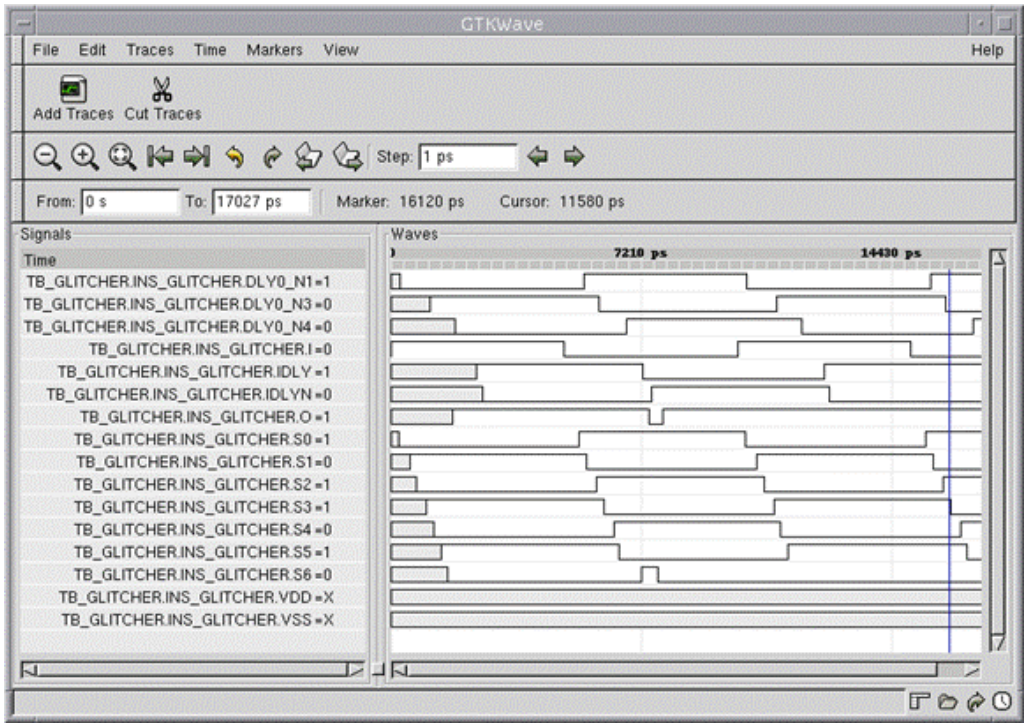


The simulation of the zero-delay model makes the glitch on `S6` appear at the rising transition of the input signal `I`, which is false.

5.5. With-delay Model

The with-delay behavioral model is obtained by setting the variable `yagleTasTiming` to `max`.

To observe the glitch, it is necessary to have a second level precision. The variable `yagleSplitTimingRatio` is positioned to 1.



We can observe on the simulation trace (`glitcher_timed.vcd`) the correct behavior of the glitcher: the glitch is implied by the falling edge of `I`.

Chapter 6. Behavioral Optimization

One often wants to have a compact behavioral description. In order to optimize the VHDL|Verilog, Yagle disposes of three means:

- Inverter minimization
- Expression simplification
- Signal suppression

6.1. Inverter Minimization

Inverter minimization reduces a even chain of inverters to a single buffer, and an odd chain of inverters to a single inverter. This optimization is compatible with timing back-annotation of first level: the delay of the reduced buffer or inverter is the sum of the delays of the chain of inverters.

If timing back-annotation is of level two or three, inverters are split into up and down expressions, and are not minimized.

6.2. Expression Simplification

When expression simplification is invoked, Yagle analyze each cone, and identifies the NOR, NAND and XOR expressions. This optimization is compatible with timing back-annotation of first level. The delay associated with the cone does not change.

If timing back-annotation is of level two or three, no simplification is done.

6.3. Signal Suppression

Signal suppression can lead to aggressive optimization, as we will see in the following example. The principle is to replace a signal `S` by its expression, in all the expressions depending on the signal `S`. Signal suppression is not compatible with timing back-annotation. Expression simplification is always applied after signal suppression.

Chapter 7. Addaccu Example

This example intends to illustrate the VHDL optimization techniques described in the previous sections. Functional abstractions will be performed with and without timing back-annotation, in order to study the compatibility between optimizations and timing back-annotation.

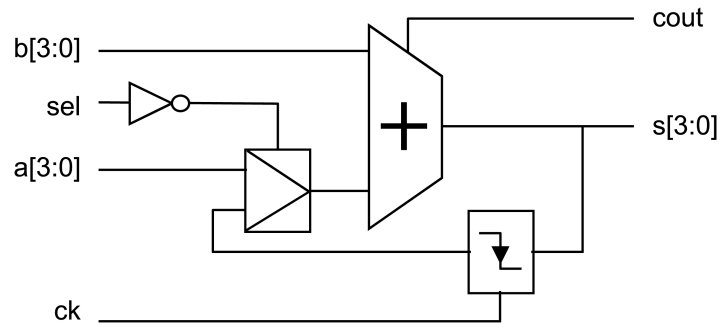
7.1. Addaccu Directory

The files and scripts needed for this example can be found in:

```
$AVT_TOOLS_DIR/tutorials/yagle/addaccu/
```

7.2. Addaccu Design

The addaccu chip consists of a four-bit adder, a four-bit register, and a 2 to 1 four-bit multiplexer.



The circuit performs an addition between either the `b[3:0]` and `a[3:0]` inputs when sel is set to 0, or between `b[3:0]` and the content of the four-bit register when sel is set to 1. The content of the register is overwritten by the values of the outputs `s[3:0]` on each falling edge of the clock, ck..

7.3. Inverter Minimization

To activate inverter minimization, we just need to add the following line in the Tcl script:

```
avt_config yagleMinimizeInvertors yes
```

The tool is invoked in the classical way:

```
> run.tcl
```

For example, let's consider the signal `s(1)`, in the optimized and non-optimized VHDLs:

```
s(1)    <=    not n5    after 259 ps;
n5      <=    not n4    after 768 ps;
n4      <=    not n3    after 533 ps;
n3      <=    not ss_1  after 421 ps;
```

```
s(1)    <=    ss_1    after 1981 ps;
```

In the non-optimized VHDL, the signal `s(1)` is assigned through the first chain of inverters, and through the second simple assignment in optimized VHDL (where 1981 = 259 + 768 + 533 + 421):

7.4. Expression Simplification

To activate expression simplification, we just need to add the following line in the Tcl script:

```
avt_config yagleSimplifyExpressions yes
```

The tool is invoked in the classical way:

```
> run.tcl
```

For example, let's consider the signal `n37`, in the optimized and non-optimized VHDLs:

```
n37 <= not core_mux_3 and not bb_3 after 525 ps;
```

```
n37 <= core_mux_3 nor bb_3 after 525 ps;
```

In the non-optimized VHDL, the signal `n37` is assigned by the first code line, in the optimized VHDL, the signal `n37` is assigned by a more compact expression as displayed on the second code line of the example.

7.5. Signal Suppression

From the design process, we know that the chip addaccu is made up of elementary gates, such as nor, nand, xor. We also know that the names of the internal signals of those building gates are all numbers: they will be prefixed by the tool by a 'n'. If we want to retrieve the expressions of the original RTL design, it is sufficient to suppress all the expressions relative to internal signals of building gates, i.e. all the expressions built on signals beginning with a 'n'.

Signal suppression is performed by the mean of the following function:

```
inf_SetFigureName addaccu
```

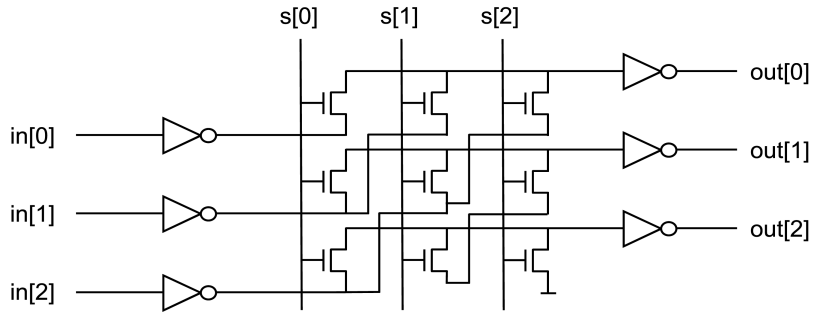
```
inf_DefineSuppress "n*"
```

Looking at the `addaccu.vhd` file, we can see that the tool has retrieved the original XORs building the adder.

Chapter 8. Shifter Example

8.1. Shifter Design

This example intends to illustrate the false branches detection mechanism included in Yagle. A shifter is the typical design leading to the construction of false branches. This example is limited to 3 bits, but it is already possible to see that a simple depth-first search leads to the construction of a large number of branches having a large number of transistors.



In this example, we are going to study the behavior of Yagle on two shifters: a 4-bit shifter and a 32-bit shifter.

8.2. Shifters Functional Abstraction

As previously explained , Yagle performs a functional analysis of the logical context. This functional analysis establishes the correlations between the transistors of the branches under construction. Yagle is then able to orient the transistors and to detect false branches.

It is possible to control the depth of the functional analysis with the `yagleAnalysisDepth` variable. In default configuration, the depth is 7. This depth is sufficient to perform the functional abstraction of most of the chips, in particular the present shifters:

```
> run.tcl
```

The functional abstractions last a few seconds. We are now going to study the effects of lowering the depth of the analysis on the 4-bit shifter. Reducing the depth to 6 still leads to a correct abstraction, but reducing the depth to 5 leads to the construction of some false branches. To study the VHDL generated with a functional analysis depth of 5, set the yagleAnalysisDepth variable to 5 and run the Tcl script.

We can see that the VHDL generated with a depth of 5 has more complicated expressions than the VHDL generated with a depth of 6 and more. These expressions are the result of false branches and actually, this VHDL is not functional. With a depth less than 5, they are a huge number of false branches. The research is only stopped by a security mechanism avoiding combinational explosion (actually the maximum length of a branch).

Chapter 9. Retrieving ROM Content

The files and scripts needed for this example can be found in:

```
$AVT_TOOLS_DIR/tutorials/yagle/rom/
```

This example shows the application of Yagle in the case one wants to verify or to retrieve the content of a ROM. We have here three designs of a ROM of 256 words of 8 bits, programmed in three different ways:

- r256x8_1: data = address
- r256x8_5: data = full one
- r256x8_6: data = full zero

The reference VHDL model of `r256x8_1`, `r256x8_5` and `r256x8_5` are respectively described in the files `r256x8_1.vbe`, `r256x8_5.vbe` and `r256x8_6.vbe`. Here is the reference VHDL model of `r256x8_1`:

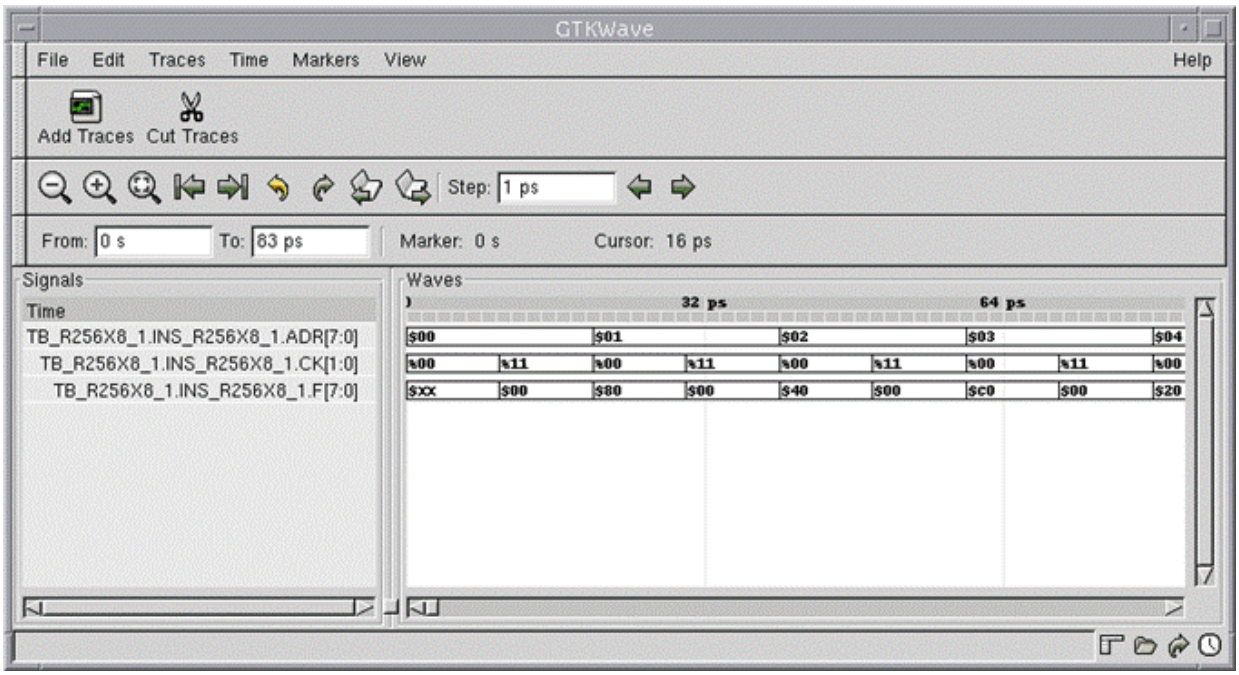
```
ENTITY r256x8_1 IS
  PORT( adr  : IN BIT_VECTOR(7 DOWNTO 0);
        ck   : IN BIT_VECTOR(0 TO 1);
        f    : OUT BIT_VECTOR(0 TO 7);
        vdd  : IN BIT;
        vss  : IN BIT );
END r256x8_1;
ARCHITECTURE VBE OF r256x8_1 IS
   SIGNAL m_out : BIT_VECTOR (0 TO 7);
BEGIN
   F = m_out WHEN (ck = B"00") ELSE B"00000000";
   WITH adr(7 DOWNTO 0) SELECT
      m_out(0 TO 7) = B"00000000" WHEN B"00000000",
                      B"00000001" WHEN B"00000001",
                      B"00000010" WHEN B"00000010",
                      B"00000011" WHEN B"00000011",
                      B"00000100" WHEN B"00000100",
                      B"00000101" WHEN B"00000101",
                      B"00000110" WHEN B"00000110",
                      B"00000111" WHEN B"00000111",
                      B"00001000" WHEN B"00001000",
                      B"00001001" WHEN B"00001001",
                      B"00001010" WHEN B"00001010",
                      B"00001011" WHEN B"00001011",
                      ...
                      B"11111110" WHEN B"11111110",
                      B"11111111" WHEN B"11111111";
END VBE;
```

To perform the abstraction of those designs, the line `avt_config yagleTristateIsMemory yes` has been added in the script.

The `run.tcl` commands generate the VHDL behavioral descriptions `r256x8_1.vhd`, `r256x8_1.vhd` and `r256x8_6.vhd`.

The content of the ROM can then be retrieved with a simple logic simulation.

The following screenshot displays the simulation trace `r256x8_1.vcd` of `r256x8_1.vhd` (data = address).



Index

No index for this document.