Design of ALU components

1. Introduction

The ALU (Arithmetic and Logic Unit) is a digital circuit that performs arithmetic and logical components. This circuit is the basic block of the CPU and GPU and even for the simplest microprocessors.

An ALU must be able to calculate most operations. If there are more complex operations, the value of ALU is higher (i.e. more expensive), uses more space in the processor and dissipates more power. Thus, the design is a tradeoff between computational power and area overhead/power consumption.

Almost all processor operations are done by one or more ALUs. An ALU loads the data from the input registers. The external control unit tells the ALU which operation (i.e. operation code) to perform on the data, and then the ALU stores the result into an output register. The control unit is responsible for moving the processed data between these registers, ALU and memory. Many of the ALU designs also contain a status register that indicates different cases, such as: carry-in or carry-out, overflow, divide-by-zero and so on.

This laboratory work contains the basic designs of the ALU components: adders, multipliers and divisors. The focus is on how to write VHDL code for each ALU component and how to simulate them in VHDL.

2. Addition

The addition operation is the most frequently used arithmetic operation in any computer system. If there are more complex arithmetic functions, they are reduced to a series of additions. Note that by increasing the speed of addition, the speed of the ALU is also increased; but the speed and cost of adders are directly proportional to their complexity.

Full Adder

This circuit is the basic addition block, which adds three 1-bit inputs: 2 bits to be added (X_i and Y_i) and a 1 bit carry-in (C_i). It generates 2 outputs: the sum bit (S_i) and the carry-out (C_{i+1}) bit. The figure below illustrates a full adder symbol.

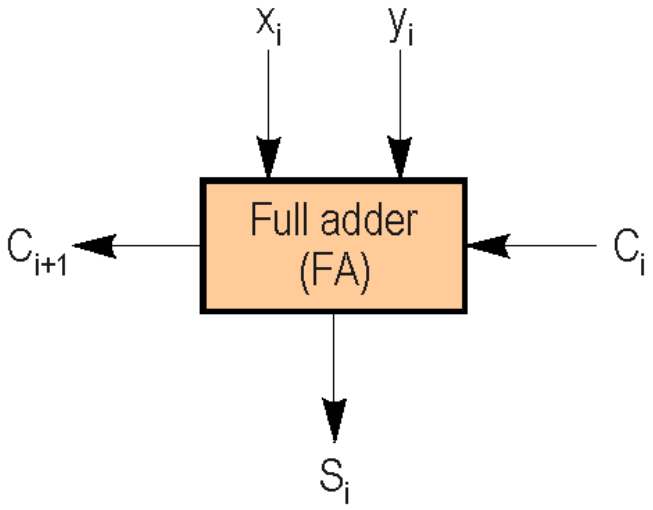

Figure 1. Full adder

Table 1. Full adder truth table

X_i Y_i C_i	S_i C_{i+1}
0 0 0	0 0
0 0 1	1 0
0 1 0	1 0
0 1 1	0 1
1 0 0	1 0
1 0 1	0 1
1 1 0	0 1
1 1 1	1 1

Given the truth table above, the Boolean expressions of the outputs (after reduction) are:

- $S_i = x_i \oplus y_i \oplus C_i$
- $C_{i+1} = x_i \cdot y_i + (x_i + y_i) \cdot C_i$

A **half adder** is basically a full adder without the carry input and generates a sum bit and a carry bit.

Ripple Carry Adder

This type of adders are implemented by several full adders in series, each carry output is connected to the input of the next one. This is a parallel adder and is used for adding n-bit number. It has the advantage of simplicity and low cost, but at the cost of reduced speed. The figure below illustrates the block diagram of a ripple carry adder for adding 4-bit numbers.

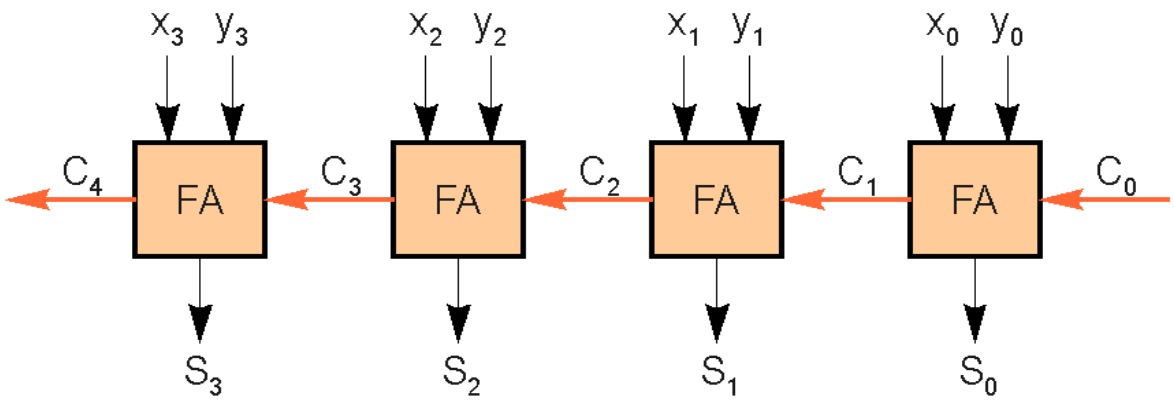

Figure 2. Ripple carry adder using 4 full adders, for 4-bit numbers

Carry Lookahead Adder

In order to reduce the time required to form the carry signals, this type of adders use a separate block which calculates the carry output of each full adder. Thus, there is no "wait time" for carries to ripple from stage to stage (like in the ripple carry adder design). In the figure below, the block diagram of a carry lookahead adder that adds 4-bit integers is exposed.

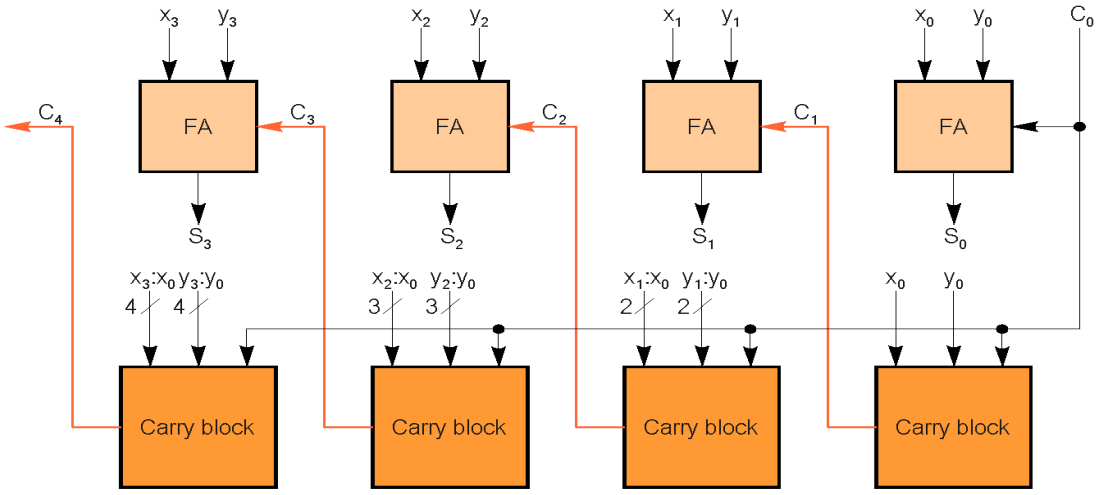

Figure 3. Carry lookahead adder design for adding 4-bit numbers

Given the design above, two functions are used to *generate (G)* and *propagate (P)* the carry output.

- $g_i = x_i \cdot y_i$
- $p_i = x_i + y_i$

Using the 2 functions, an output carry is: $C_{i+1} = g_i + p_i \cdot C_i$. Thus, each carry output can be formed using the g and p functions from the same stage and from the previous stages. The carry blocks from the picture above can be joined in a single carry lookahead generator and each carry output can be generated by a combinational circuit.

Other types of adders

Besides the adders described above, there are several types of adders that have advantages and disadvantages over regular designs and which are worth mentioning.

Carry Select Adder

This type of adder uses redundant hardware to speed up the addition process. It divides the adding operation into 2 parts: the high-order half and the low-order half. First, the high-order half of the sum is calculated for both possible input carries. When the carry from the low-order half of the sum is known, the proper high-order half can be selected.Note that there is the possibility to divide the adder into quarters, so even lower complexity. This is why the complexity of this type of adder overcomes the carry lookahead adder disadvantages.

Carry Save Adder

This type of adder is used when more than 2 numbers are to be added because it reduces the carry propagation time. The design consists of n independent full adders (for n-bit numbers), with the inputs being the n-bit numbers and the outputs are an n-bit sum word and an n-bit carry word. Basically, each full adder works independently from the others and the carry signals are not propagated between the individual full adders. To get the final result, both parts (sum and carry) must be added together using a normal type adder.

Serial Adder

This is the simplest type of adder because it uses a single full adder and a D-latch. Basically, it performs the addition step by step from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). The D-latch is used to propagate the carry of the previous sum to the next bits that are going to be added.

3. Multiplication

The multiplication of binary numbers is similar to that of decimal numbers. There are various methods of multiplying 2 n-bitnumbers:

- Shift-and-Add Multiplication

- Booth's Technique
- Higher-Radix Multiplication
- Array Multiplier
- Wallace Tree

In the following sections, only 2 out of the 6 methods will be explained in detail.

Shift-and-Add Multiplication is one of the basic and simplestmethod for adding 2 numbers.Basically, the whole idea is to add the multiplicand (let's say X) to itself for Y (multiplier) times. The algorithm is based on taking each digit of the multiplicand in turn and multiplying it by a single digit of the multiplier. Each intermediate product is placed in the appropriate positions, to the left of the earlier results. Finally, all the intermediate products are added together, to get the final result. The block design of the shift-and-add multiplication technique is illustrated in the figure below.

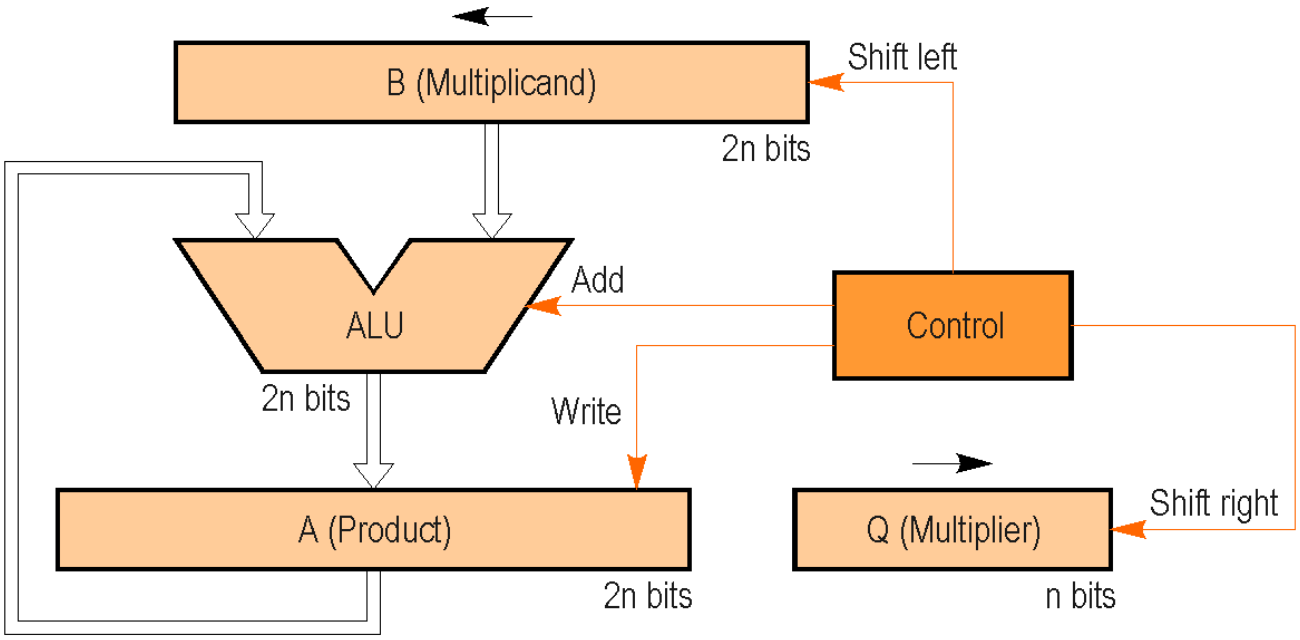

Figure 4. Block design of the shift-and-add multiplication technique

The **Wallace Tree** technique is based on combining pairs of partial products with the help of multiple levels of Carry Save Adders:

- at each level of the tree, the numbers are grouped into three and are added together
- the levels continue until only 2 numbers are left to be added
- a Carry Propagate Adder is used to add the last 2 numbers and deliver the final result

This design reduces the number of terms to be added by a factor of 1.5, thus resulting in a total time of $O(\log_{1.5} n)$. Note that the preceding multipliers have a time of $O(n)$, where n is the

number of bits for each number. Below is a figure of a block design of a Wallace Tree for multiplying two 8-bit numbers.

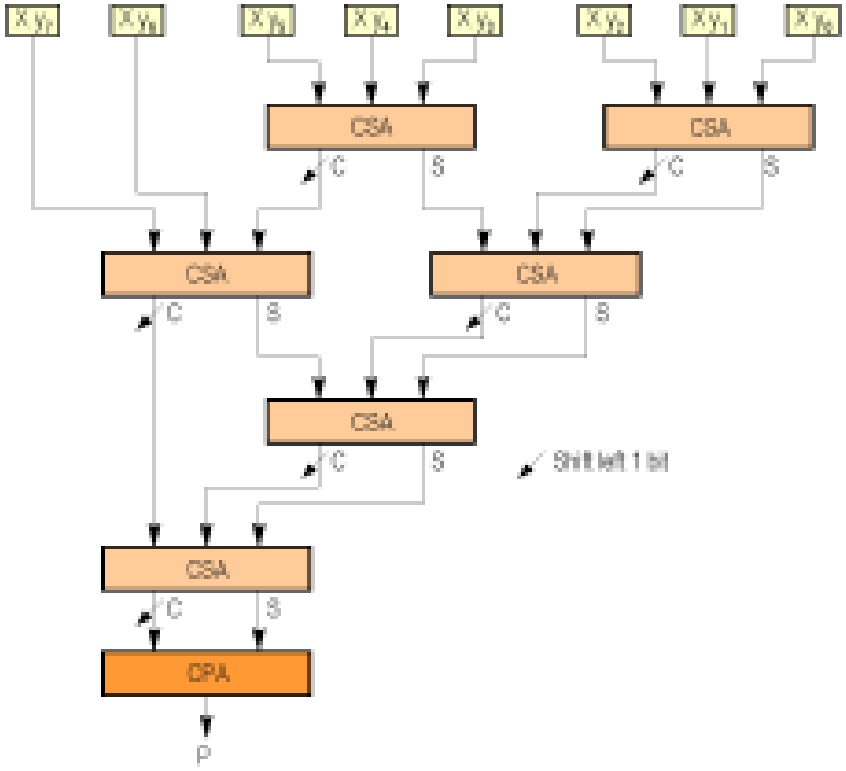

Figure 5. Block diagram of the Wallace Tree design for multiplying two 8-bit numbers

The Wallace Tree method can be combined with other methods to further increase the speed. For example, the Booth technique can be used to produce the partial products, while a Wallace Tree adds the partial products.

4. Division

In any division operation, we have the first operand called *dividend* (X), the second operand called *divisor* (Y) and the results are the *quotient* (Q) and *remainder* (R). The mathematical expression is: $X = Q * Y + R, R < Y$.

The algorithm for decimal division is explained below:

1. Choose a digit and subtract the product between this digit and the divisor from the partial remainder
2. If the result is smaller than the divisor, the digit was chosen correctly
3. Otherwise, choose another digit and repeat the subtraction

The binary division algorithm is based on repeated subtractions of the divisor Y from the partial remainder R, but are executed only if Y$\leq$ R, which results in a quotient digit of 1 (otherwise is 0).

The basic block design of the division operation is illustrated below in figure 6.

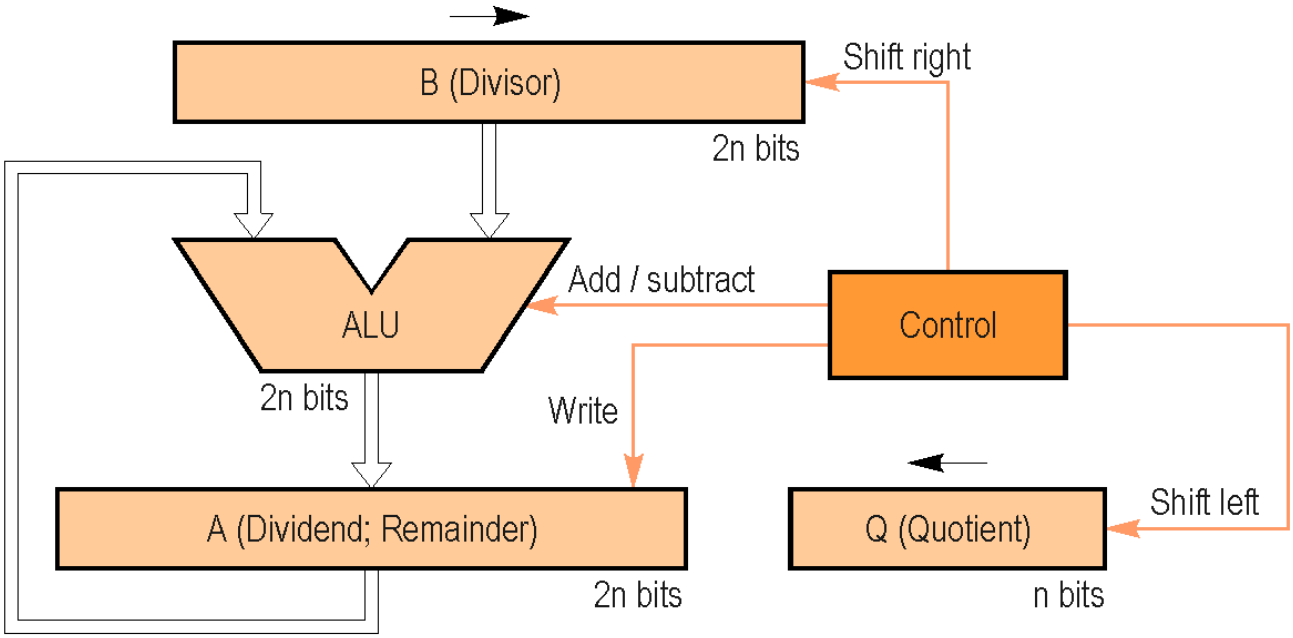

Figure 6. Basic block design of the division operation

Note that this block design can be improved in order to reduce the time and to simplify the hardware needed. For example, shifting the partial remainder to the left (instead of shifting the divisor to the right) produces the same alignment and simplifies the hardware of the ALU and the divisor register (n bits instead of $2n$). Another idea is based on the fact that the first step cannot generate a digit of 1 in the quotient, thus the order of the operations can be switched: first shift, then subtract (one iteration removed). Also, the size of the A register could be reduced to half and could be combined with the Q register. So, the bits of the dividend are shifted into the A register instead of shifting zeros, and both A and Q registers are shifted left together. Considering the above, the improved block design is in figure 7 below.

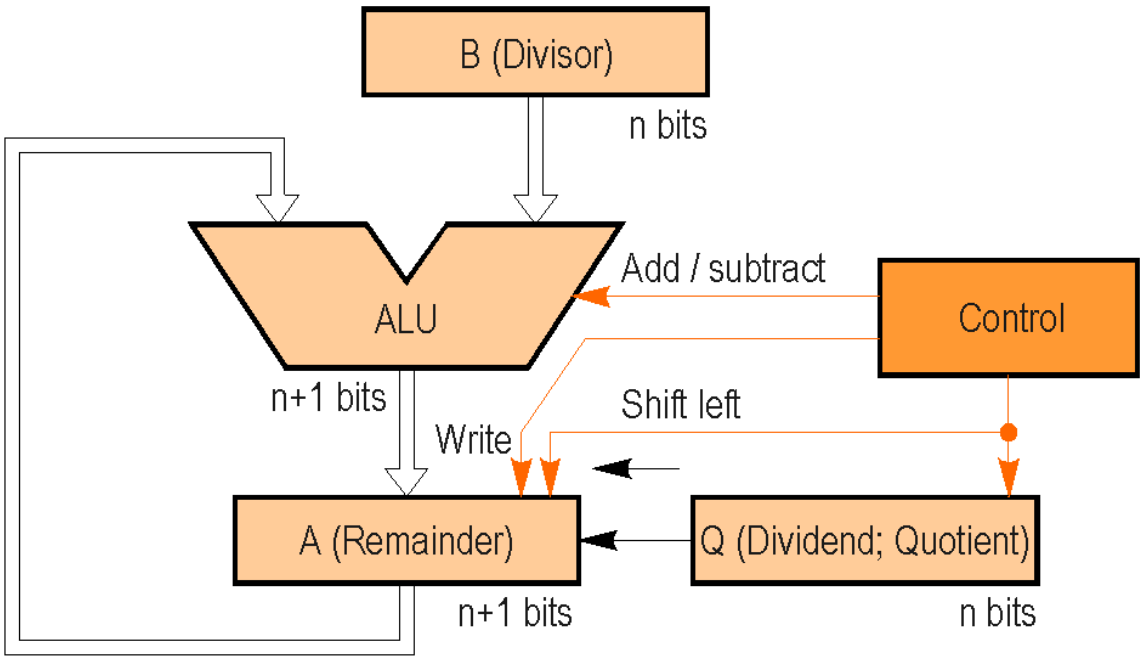

Figure 7. Improved block design of the division operation

5. SimulatingVHDL designs in Xilinx ISE

There are 2 ways to simulate in Xilinx: either by creating a Test Bench file or by using the Test Bench Waveform Editor.

A test bench file contains VHDL code:

- which unit is tested (by using the *component* declaration block)
- initial/default signal values (e.g. initial value for reset is 0, enable signal is 0, etc.)
- clock signal process (e.g. every 0.5ns, the clock signal changes its value)
- general process which describes how the simulation flows (e.g. use of *wait for* statements, toggle some signals from 1 to 0 or vice versa, etc.)

The test bench waveform editor is similar to the simple test bench file, but it is graphical. Basically, you can setup at any time (the max time is the simulation time, typically 1000ns) a value for any signal which was declared in the entity, just by clicking on the waveform. In general, the test bench waveform editor is simpler and more convenient than the test bench file creation when simulating simple and small designs. For more complex ones, the test bench file is more appropriate.

In the following paragraphs, the basic steps for simulating VHDL designs in Xilinx ISE are explained.

Before simulating, the project must be already created and must contain at least one VHDL source file. In order to compile, first select the source file from the *Hierarchy* pane, right-click and select "Set as Top Module". Then, in the *Processes* pane, double-click the *Synthesize – XST* line. This will check the syntax of the VHDL code, compile it and prepare it for simulation.

Creating a Test Bench File

1. In the *Hierarchy* pane, right-click the project name and select *New Source.* In the new window, select *VHDL Test Bench*and enter a name (should be different than the source file). In the *Associate Source* page, select the association with the desired entity.
2. The test bench file is now declared as a component and instantiated with the label *uut* (Unit Under Test). In the *Hierarchy* pane, select from the drop-down menu *Sources for: Behavioral* and below, double-click the test bench file. The test bench can be modified to satisfy needs.
3. In the *Processes* pane, expand the *Xilinx ISE Simulator* and double-click the *Check Syntax* line. Correct any errors if any reported and recompile the file.

Performing Simulation

1. The default time for the entire simulation is 1000ns. This can be changed in the *Process* pane, by right-clicking the Simulate Behavioral Model and selecting *Process Properties.* In the new window, change the *Simulation Run Time*property to any desired value.
2. In the *Process* pane, double-click the *Simulate Behavioral Model* line. Now the ISim simulator is launching, which compiles the source file and the test bench file, builds simulation files and performs the actual simulation for the time specified. The results are shown in a *Wave* window.

Simulation using the Test Bench Waveform Editor

1. In the *Hierarchy* pane, right-click the project name and select *New Source*. In the *New Source Wizard,* choose *Test Bench Waveform,* input a name and press Next. Select which is the unit under test and then finish the wizard. In the new *Initial Timing and Clock Wizard* window, choose and setup the clock signal timings (if necessary).
2. Select *Behavioral Simulation* in the top menu from the *Hierarchy* pane and then double-click the test bench waveform created in the previous step.
3. In the waveform editor, the value of a signal can be changed by clicking on the waveform which corresponds to the desired signal. Note that a value can be changed once per clock cycle. Save the changes.

4. From the *Processes* pane, expand *Xilinx ISE Simulator* and double-click *Simulate Behavioral Model*. This will launch the simulator and the simulation results (as waveforms) will be visible in a new *Wave* window.

6. Applications

6.1. Implement and simulate in VHDL a 4-bit counter (check the previous laboratory work).

6.2. Design and implement in VHDL the modified version of the carry lookahead adder for 4-bit numbers.

- Form the 4 carry outputs using the *g* and *p* functions
- VHDL implementation
- Test and simulate the design for 2 random 4-bit numbers

6.3. Design and implement in VHDL Carry Save Adder for three 4-bit numbers.

6.4. Make a comparison chart that shows the advantages and disadvantages of each multiplying technique.

6.5. Implement in VHDL the Wallace Tree for multiplying two 4-bit numbers. Simulate the design for 2 random numbers.