Programming with HDLs

Paul Chow

February 11, 2008

1 Introduction

The purpose of this document is to encourage the proper approach or mindset for programming in a hardware description language (HDL), particularly VHDL and Verilog.

The motivation for this document is seeing too many students thinking that an HDL can be used the same as any typical programming language, such as C. Taking this approach will:

1. not work well or at all,
2. not generate good logic,
3. cause great frustration because you cannot get your code to work

The goal is to keep this document as short as possible so that everyone learning to use these languages will read it and hopefully something will stick.

Some common problems due to the subtleties of the languages are also provided.

For more details and references, it is suggested that you look at the references listed at the end. The Brown and Vranesic books [1, 2] are excellent places to start learning to write synthesizable code. There is enough of the languages described there to create good designs for any circuit you would want to build. The Thomas [3] book is recommended as one of the better descriptions of the full Verilog language. The Smith [4] book provides many examples for both languages and explains many of the subtleties of how the languages work and don't work.

2 Why is an HDL different from an HLL?

When writing in HDL, you must take a different approach than you would implementing a typical piece of software. It is a different mindset. To write good HDL programs, you must keep this in mind.

This section tries to contrast coding for *software* and coding for *hardware*.

2.1 Software

Software $\Longrightarrow$ design of algorithms and data structures

A HLL program is the description of an **ALGORITHM**. The compiler takes care of the rest.

Compilers are good enough that they can take almost any algorithm description and turn it into a working implementation on a processor. But, they cannot make a poor algorithm into a good one. For example, a compiler cannot convert an n^2 Bubble Sort procedure into an $n\log(n)$ Quick Sort procedure.

With more thought, you can write software that results in better implementations, like using less memory or having a slightly better run time. More often the concern is to get something working correctly versus working correctly in a highly efficient way because typically you have lots of memory and lots of CPU power so whatever you write is likely good enough.

Programming embedded processors is somewhat a different environment where memory, speed, and power are greater concerns. Still compilers can do a lot for you.

In a CPU, you have a fixed target instruction set and not too many choices for an *implementation*, i.e., machine code sequences. The *RISC* philosophy was a realization that you can do better optimizations with a good set of atomic operations versus using highly complex instructions, based on the compiler optimization technology of the time (mid 1980's). This is still mostly the case.

2.2 Hardware

Hardware *compilation* technology, i.e., *synthesis* is a much more difficult problem than compiling for a processor. If you start with an algorithmic description and want to turn it into hardware, the number of possible implementations is extremely large: use NAND or NOR gates, adders or multipliers, how many of each? This general problem falls under the area of *behavioural synthesis*, which is still a subject for many more Ph.D.s. This is why you still cannot get efficient hardware implementations starting with a language such as C. There are some commercial products available, but they work best only in specific, highly-constrained situations. You cannot take the source for GCC and expect to compile it into a hardware GCC circuit.

In software, where a compiler cannot turn a bad algorithm into a good one, the compiler can still create an implementation of the bad algorithm. Usually, hardware cannot implement any arbitrary algorithm because there are more constraints to consider, such as memory, area, speed, power and available resources. An algorithm to be implemented in hardware must be developed with consideration of the architecture and other constraints (area, speed, power, etc.) in mind.

An example of an architectural choice would be whether the system contains one multiplier that is shared, or one multiplier for each multiplication required. This is clearly a choice between area and speed, as well as power.

Consider the example of building a circuit to do $N!$. You could use a counter, or a subtracter/register combination. One multiplier is the obvious solution. What if that is too slow? Can you do it with two multipliers? Consider, one down counter from N and one upcounter from 1, stop when they *meet*. You must also handle special conditions for odd/even numbers. The point is that thes are different architectures with different tradeoffs, including the complexity of the design.

You could build the entire design with NAND gates, or NOR gates. You must decide whether they should be 2-input, 3-input, or 4-input gates with weak drive, medium drive, or high drive based on power versus speed considerations. You could decide to use a more complex function, like an adder, but should it be a ripple-carry adder, a carry-lookahead adder, or a carry-select adder? These are all architectural choices.

The architecture must be developed with consideration of the algorithm and other constraints. This relationship between algorithm and architecture means that there can be a significant iterative process, up front, to develop the architecture that will be implemented in the hardware.

This iterative process relies a lot on the experience of the designer to significantly prune the implementation space, something that is extremely difficult to automate at this time.

Without an architectural specification, you cannot start any HDL coding.

So, the architecture is the implementation of the algorithm. The hardware is the implementation of the architecture. Your HDL is a description of your hardware (NOT the algorithm). The synthesizer (compiler) can then do a good job of taking care of mapping your HDL to the technology.

Hardware $\Longrightarrow$ data flow as defined by the architecture and the control of the flow

3 The Hardware Design Process

This is the approximate procedure that is required to achieve a proper hardware implementation of some function:

1. Come up with an algorithm/architecture. Iterate between the algorithm and the architecture.
2. Create a schematic diagram of the circuit you wish to build. This is the actual design of the hardware you want to implement. As you get more experienced, the schematic may only exist in your mind, but you should still think *schematic*. **The HDL is only a textual description of a schematic.**

3. Write HDL to describe your hardware in a form that can be mapped by the synthesis tool.

Do not just start writing code!

4 What is an HDL?

- needs to be able to describe hardware
- hardware is typically inferred from behavioural description
- hardware is a collection of functional units operating in parallel
- HDL must be able to express parallelism
  - really a parallel programming language
- explicit timing
- originally developed for modelling and simulation
  - then design specification $\Longrightarrow$ synthesis
- verification
  - use modelling capability to help verify the correctness of a design intended for implementation/synthesis $\Longrightarrow$ testbenches
  - all versions (RTL,netlist) should give same result
  - careful coding required to achieve this
  - can have simulation not equal to (simulation of) synthesis result

5 Describing your schematic

- i.e., partitioning your HDL
- schematic has many *boxes* or functional units
- how to express these in an HDL?
- partition your code in the same way as you partition the design (schematic)

5.1 entity/architecture : module

- distinct partitions in the design
- design entities – an instantiation is a component
- hierarchy possible – components within components
- replicated/reusable blocks
- has explicit ports declared
- connected at a higher level where the component is instantiated with signals (wires) that connect to the ports

5.2 processes : always

- no processes within processes
- *simulation* partition – sequential execution
- state machines
- chunks of related combinational logic
- state registers
- exist within a module
- connected together with signals (wires) declared in the module
- "a page or so"

5.3 concurrent assignment statements

- combinational logic
- VHDL – simple assignment, with/select, when/else
- Verilog – continuous assign, with conditional operator
- VHDL generate/Verilog for – repetitive structures of combinational logic or components

Processes and concurrent statements execute in **parallel**.

5.4 Signals, Wires and Regs

- need to connect various components together, where component could be the instantiation of some logic or process/always blocks
- in VHDL, use signals for connectivity
- can be on left-hand sign of an assignment
- in Verilog, more subtle
  - **reg** declaration does not mean that you get a **register**, like, say flip flops. See Example 4c. First always block is combinational logic, but `X` and `Y` are declared as **reg** and they do not result in flip flops.
  - from simulation perspective, **reg** means that you get a static variable.
  - **reg** can be inside or outside an **always** block
  - **wire** cannot be inside an **always** block
  - **wire** mainly used to connect components together within a module
  - **wire** can be on the left-hand side of a concurrent assign statement.

5.5 Verilog Gotchas

How not to design a language...

- declares variables for you if you forget to or **make a typo**
- assignments to unmatching bit widths **work**
- 3'b1010 is not a 4-bit constant!

6 How a beginner should code in a HDL

- Draw a schematic diagram, not necessarily to the gate level, but at the functional block level, where the implementation of the functional block is clear (adder, mux, combinational equations, register, etc.)
- HDL should be structured in the same way as your schematic diagram. The functional blocks should clearly map to your schematic diagram.
- Functional blocks can be modules/entity+archs, always/process blocks or concurrent assignment statements.
- Keep datapath and control functions (FSMs) separate. Keep sequential logic and combinational logic separate. In FSMs, the NextState/PresentState structure is easy to read.
- Be able to count your flip flops in your HDL, i.e., you should be able to predict, prior to synthesis, the number of flip flops in your design. Look at the synthesis reports to confirm.
- Look at reports and check for inferred latches. If you see any, you most likely have a problem.
- Look at other warnings.

The examples in the Brown/Vranesic books are good and should synthesize well.

For those that plan on doing lots of HDL coding, you might check out *HDL Chip Design* by Douglas Smith [4].

6.1 Other Wisdom

Starting with coding style:

> If you don't follow simple rules, strange things will happen to you...
>
> If you do follow simple rules, you have a better chance of getting something to work.

Then with verification:

> If you don't simulate, it won't work.
>
> If you do simulate, it might work.

7 Gotchas to watch for

These are common problem areas when coding with an HDL. Also look at Appendix A.15 in the Brown/Vranesic Verilog book [2] and Appendix A.11 in the Brown/Vranesic VHDL book [1].

7.1 sensitivity lists

Make sure that you have all inputs of the process/always block in the sensitivity list. If you don't, then your simulation will most likely behave differently from the circuit that is synthesized. Simulation uses the lists as a way to make the simulation more efficient. If no inputs in the list changes, that particular block will not be simulated. Synthesis ignores the lists.

Check the warning messages of your simulator. Tools often flag this because it is usually a mistake.

7.2 Inferred latches when using IF/CASE

If any control path is not defined or does not assign to an output, then a latch will be generated. For example, an **if** statement without an *else* clause will infer a latch. For *case* statements, make sure you use the others/default clauses and assign a value to every output in every branch. Remember that signals/wires do not just have the value of '1' or '0' so if you are switching on a 2-bit signal, there may be more than four cases!

7.3 VHDL Signals/Variables and Verilog Non-blocking/Blocking Assignments

You can run into real problems if you are not careful how you use these features of the language.

Some basic points:

- VHDL variables update immediately
- A Verilog blocking statement blocks the simulation till it completes having the same effect as updating the value immediately
- VHDL signals and Verilog non-blocking outputs only update at the end of the process/always block. Use this property to build shift registers.
- Assume that all assignments in an edge-triggered Verilog *always* block will result in a register, even blocking assignments. If the result of that assignment is not used outside of the module, then the register may disappear due to an optimization.

7.4 General Guidelines

There are many ways to write code that does strange things, though they can be explained if you fully understand all the semantics of the languages. If you just want to write working code without having to worry about the weird possibilities, which is what most of us want, then here are some general guidelines to follow that make it less likely for you to get into trouble:

Keep combinational logic and sequential logic in separate process/always blocks. In other words, **do not mix** combinational logic and sequential logic in the same blocks.

State machines can get particularly confusing when the combinational and sequential logic are mixed. Use separate blocks for the next state logic and the state registers. For clarity, use another block for the output combinational logic.

For combinational logic process/always blocks

VHDL use variables and assign one signal for the final output

Verilog use blocking assignments

For sequential logic process/always blocks

VHDL use signals for every value to be registered.

Verilog use non-blocking assignments.

7.5 Some Examples of Non-obvious Results

If you want to know more and to see some examples of what can happen, read on. At first glance, the code may look reasonable, but non-obvious results arise due to *features* of the languages. All examples are written in a style that can lead to problems. First, combinational and sequential logic should always be separated. Next, other subtle issues are demonstrated here. None of these examples are written using *good* code.

The message of all these examples is that if you are not careful with these languages, unexpected things (from your intended result) can happen. The language will not protect you from making these mistakes. The best way to *reduce* the likelihood of a problem is to adhere to the basic guidelines given above in Section 7.4.

The figures are generated using *Synopsys Design Analyzer* and the default *GTECH* library to just read in the design and show the schematic. This will show unoptimized synthesis results that demonstrates better what the language is inferring from the code. The figures can be found at the end of the document.

7.5.1 AND Gate and Shift Register

```
-- Example 1a -- VHDL
-- AND gate and shift register
-- any order of statements works
-- Statements executed in sequence, but effect as if concurrent

library ieee;
use ieee.std_logic_1164.all,ieee.numeric_std.all;

entity ex_1a is

  port (
    clk, A, B : in  std_logic;
    Z         : out std_logic);

end ex_1a;


architecture RTL of ex_1a is
  signal X,Y : std_logic;

begin  -- RTL

-- 3 ffs

shift: process (clk)
  begin  -- process shift

    if clk'event and clk='1' then
        X <= A and B;   -- X(t+1) <= A and B
        Y <= X;         -- Y(t+1) <= X(t)
        Z <= Y;         -- Z(t+1) <= Y(t)
    end if;
  end process shift;

end RTL;
```

```
// Example 1b -- Verilog

module ex_1b(clk,A,B,Z);
input A,B,clk;
output Z;
reg Z;

reg X,Y;

// 3 ffs

always @(posedge clk)
begin
        X <= A & B;
        Y <= X;
        Z <= Y;
end

endmodule
```

```
-- Example 2a -- VHDL
-- AND gate and register for Z only

library ieee;
use ieee.std_logic_1164.all,ieee.numeric_std.all;

entity ex_2a is

  port (
    clk, A, B : in  std_logic;
    Z         : out std_logic);

end ex_2a;


architecture RTL of ex_2a is

begin  -- RTL

-- 1 ff for Z

shift: process (clk)
variable X,Y : std_logic;

  begin  -- process shift

    if clk'event and clk='1' then
        X := A and B;
        Y := X;
        Z <= Y;
    end if;
  end process shift;

end RTL;
```

```
// Example 2b -- Verilog
// Only 1 FF.  Can think of this as actually generating a FF for
// each of X and Y, but since the FF outputs are not used outside the
// module, then the FFs are optimized away

module ex_2b(clk,A,B,Z);
input A,B,clk;
output Z;
reg Z;

reg X,Y;

// 1 ff for Z

always @(posedge clk)
begin
        X = A & B;
        Y = X;
        Z = Y;
end

endmodule


// Example 2c -- Verilog
// Make X, Y as ports to the module and you get 3 FFs now!
// Note that after optimization, two of the FFs will
// disappear and the resulting circuit looks like ex_2b above.

module ex_2c(clk,A,B,X,Y,Z);
input A,B,clk;
output X,Y,Z;
reg Z;

reg X,Y;

// 3 ff
// X and Y are now ports to the module

always @(posedge clk)
begin
        X = A & B;
        Y = X;
        Z = Y;
end

endmodule
```

```
-- Example 3a -- VHDL
-- Reorder statements of Example 2
-- AND gate and shift register like Example 1
-- Note that a process generates a "driver" for each signal
-- assigned in the process, i.e., to make it accessible outside
-- the process.  In this example, only Z is visible outside the
-- process, whereas in Example 1, X, Y, and Z are visible.
-- Synthesized circuit is the same as Example 1.

library ieee;
use ieee.std_logic_1164.all,ieee.numeric_std.all;

entity ex_3a is

  port (
    clk, A, B : in  std_logic;
    Z         : out std_logic);

end ex_3a;


architecture RTL of ex_3a is

begin  -- RTL

-- 3 ff

shift: process (clk)
variable X,Y : std_logic;

  begin  -- process shift

    if clk'event and clk='1' then
        Z <= Y; -- Y not assigned before read so FF for Y
                --  i.e., use value from last time
        Y := X; -- X not assigned before read so FF for X
        X := A and B;
    end if;
  end process shift;

end RTL;
```

```
// Example 3b -- Verilog
// Also get shift register chain like in Example 1b


module ex_3b(clk,A,B,Z);
input A,B,clk;
output Z;
reg Z;

reg X,Y;

// 3 ff

always @(posedge clk)
begin
        Z = Y;          // Y not assigned before read so connect Y FF output
                        // to Z input
        Y = X;          // X not assigned before read so connect X FF output
                        // to Y input
        X = A & B;
end

endmodule
```

7.5.2 Mixing Combinational and Sequential Logic

Example 4a In this VHDL example variables X and Y are used to compute some intermediate values. One flip flop for Z registers the result of the combinational logic.

```
library ieee;
use ieee.std_logic_1164.all,ieee.numeric_std.all;

entity ex_4a is

  port (
    clk, A, B : in  std_logic;
    Z         : out std_logic);

end ex_4a;


architecture RTL of ex_4a is

begin  -- RTL


process (clk)
variable X,Y : std_logic;

  begin

    if clk'event and clk='1' then
        X := A and B;
        Y := A nor B;
        Z <= X or Y;
    end if;

  end process;

end RTL;
```

Example 4b This Verilog example attempts to mimic the same code as the VHDL in Example 4a. However, flip flops are actually created to register the outputs of the gates in the assignments to **X** and **Y**. Note that **X** has been made an output to prevent the removal of the flip flop. The **Y** flip flop disappears so the result is that this circuit has two flip flops: **X** and **Z**.

```
module ex_4b(clk,A,B,X,Z);
input A,B,clk;
output X,Z;
reg Z;

reg X,Y;

always @(posedge clk)
begin
    X = A & B;
    Y = !(A | B);
    Z = X | Y;
end

endmodule
```

Example 4c This fixes the problem in 4b by putting all the combinational logic in a separate *always* block. Only one flip flop for **Z** is generated. This is almost the same as Example 4a, except that X is also output from the module.

```
module ex_4c(clk,A,B,X,Z);
input A,B,clk;
output X,Z;
reg Z;

reg X,Y;

always @(A or B)
begin
    X = A & B;
    Y = !(A | B);
end

always @(posedge clk)
begin
    Z = X | Y;
end

endmodule
```

Example 4d To get **X** as an output in the VHDL example of Example 4a exposes a few other issues with VHDL.

The variable **Y** can be left in the process for the **Z** flip flop since it will not result in a flip flop.

To make **X** available as an output to the design module, it cannot be computed as a *variable* inside the process because the scope of that variable is only in the process. If you tried to use the signal assignment `X <= A and B`, you would get a flip flop for **X**. You must put the combinational logic in a separate process. In this case, **Xt** is declared and the combinational logic is generated in process `comb`.

The logic for **Z** cannot read the value for **X**, because **X** is declared as an output. The internal signal **Xt** is created to solve this. To get to the output port, **Xt** is assigned to **X**, while **Xt** can be used internally.

```
library ieee;
use ieee.std_logic_1164.all,ieee.numeric_std.all;

entity ex_4d is

  port (
    clk, A, B : in  std_logic;
    X,Z       : out std_logic);

end ex_4d;

architecture RTL of ex_4d is
  signal Xt : std_logic;  -- connects processes
begin  -- RTL

comb: process (A,B)
begin  -- process comb
  Xt <= A and B;
end process comb;

X <= Xt;  -- drives output

seq: process (clk)
variable Y : std_logic;

  begin

    if clk'event and clk='1' then
      Y := A nor B;
      Z <= Xt or Y;  -- cannot read X directly here
    end if;

  end process seq;

end RTL;
```

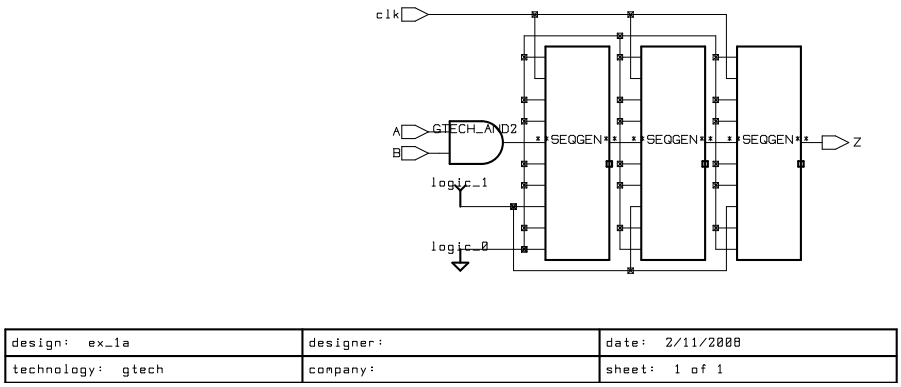


Figure 1: Example ex_1a

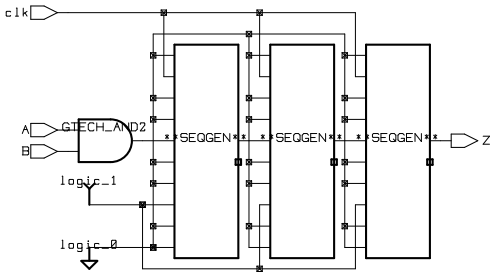
References

[1] Stephen Brown and Zvonko Vranesic. *Fundamentals of Digital Logic with VHDL Design*. McGraw Hill, 2nd edition, 2005.

[2] Stephen Brown and Zvonko Vranesic. *Fundamentals of Digital Logic with Verilog Design*. McGraw Hill, 2003.

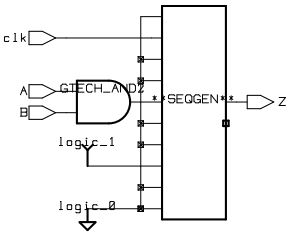
[3] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 4th edition, 1998.

[4] Douglas J. Smith. *HDL Chip Design*. Doone Publications, 1996. Out of print.



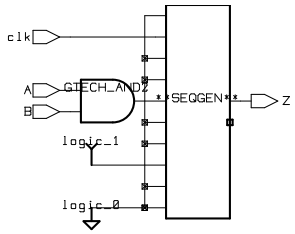
design: ex_1b	designer:	date: 2/11/2008
technology: gtech	company:	sheet: 1 of 1

Figure 2: Example ex_1b



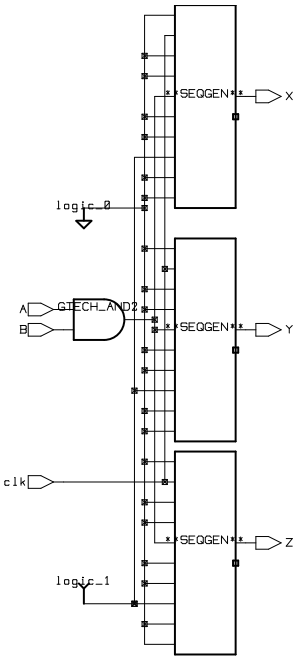
design: ex_2a	designer:	date: 2/8/2008
technology: gtech	company:	sheet: 1 of 1

Figure 3: Example ex_2a



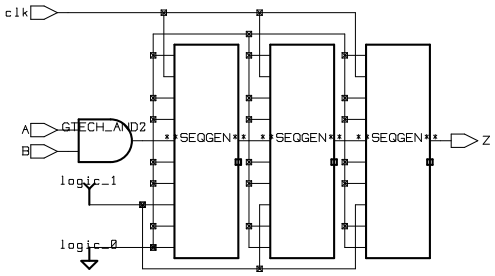
design: ex_2b	designer:	date: 2/8/2008
technology: gtech	company:	sheet: 1 of 1

Figure 4: Example ex_2b



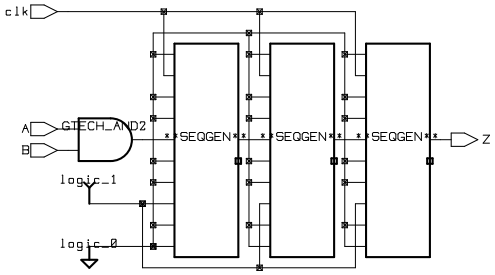
design: ex_2c	designer:	date: 2/8/2008
technology: gtech	company:	sheet: 1 of 1

Figure 5: Example ex_2c



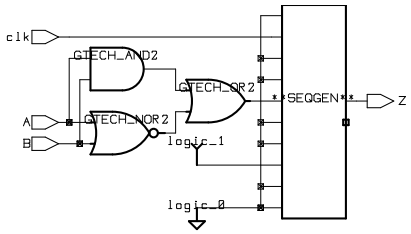
design: ex_3a	designer:	date: 2/11/2008
technology: gtech	company:	sheet: 1 of 1

Figure 6: Example ex_3a



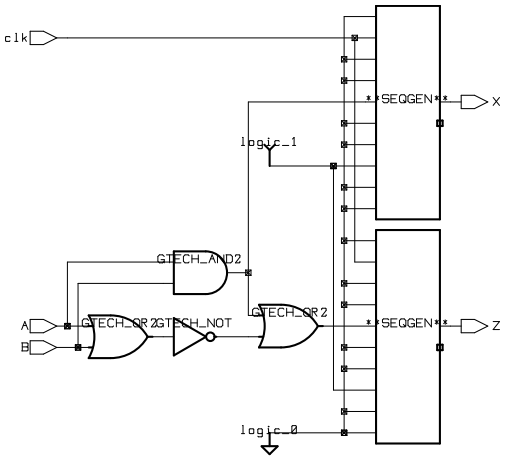
design: ex_3b	designer:	date: 2/11/2008
technology: gtech	company:	sheet: 1 of 1

Figure 7: Example ex_3b



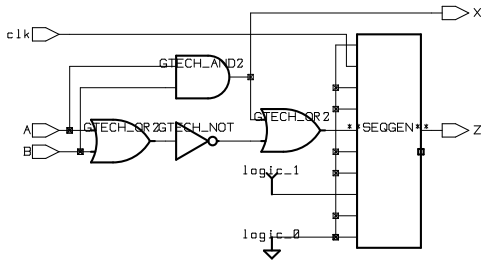
design: ex_4a	designer:	date: 2/11/2008
technology: gtech	company:	sheet: 1 of 1

Figure 8: Example ex_4a



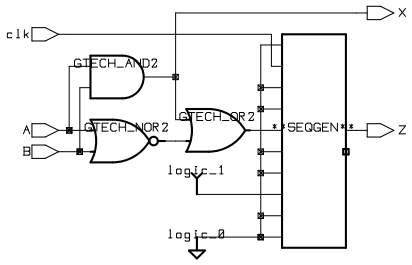
design: ex_4b	designer:	date: 2/11/2008
technology: gtech	company:	sheet: 1 of 1

Figure 9: Example ex_4b



design: ex_4c	designer:	date: 2/11/2008
technology: gtech	company:	sheet: 1 of 1

Figure 10: Example ex_4c



design: ex_4d	designer:	date: 2/11/2008
technology: gtech	company:	sheet: 1 of 1

Figure 11: Example ex_4d