Hardware Implementation of an Invariant Observer

Marko Stanisic

Department of Informatics

Technical University of Vienna

This dissertation is submitted for the degree of

Bachelor of Science

2014

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Marko Stanisic
2014

Abstract

The Invariant Observer is a Runtime Verification Unit monitoring a signal ϕ from a System under Test in real time and determining whether the signal was in an active state and had been active up to τ clock cycles before. If this is the case then signal ϕ is observed as being invariant in the past within the time interval $[0,\tau]$.

In their paper „Runtime Verification of Embedded Real Time Systems“, Reinbacher et al. [4] presented a hardware implementation of an Invariant Observer. In this thesis a different hardware implementation is shown, that allows for parallel execution of several instances, leading to significant performance improvements if the time required for determining whether ϕ holds, is not neglectable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview of the Bachelor Thesis

In embedded real time systems it is necessary to make efforts to verify a system design. A system design can be formalized by a mathematical specification within a properly chosen dynamic system model. One approach to system design verification is a deduction, which shows that the design implies the requirements.

In critical Real Time Systems (RTS) timing constraints have to be considered in the requirement engineering. Such Real Time Systems are modelled by states changing over time. Time constraints can be formulated as constraints on the duration of critical states. A real time logic should be able to specify that real time constraints. Generally it seems that two main classes of real time logic are present, explicit or implicit temporal logic.[3]

Explicit temporal logic makes use of explicit expressions of time variables. The time variable can be the representation of a time interval or a variable in temporal logic. Implicit temporal logic (for example MTL - Metric Temporal Logic) is using temporal operators that constrain the extent of a state. It is based on interval temporal logic and the duration concept. Implicit temporal logic can be very useful to express before/after relations between concurrent actions. For further details [3] can be a good source of information. In runtime verification a monitor evaluates executions of a **System under Test (SUT)** [4]. The evaluation is formalised from a formal specification described in temporal logic.

For ultra critical systems it is important to meet four major requirements:

1. Functionality: cannot change target's behaviour
2. Certifiability: must avoid re-certification
3. Timing:must not interfere with the target's timing
4. Swap:must not exhaust size, weight and power tolerance

A **Runtime Verification Unit (RVU)** is a verification monitor that meets these four major requirements. As part of this requirements, the RVU must be separated from SUT. In fact it is a synthesized hardware that monitors the execution of a SUT.
The topic of my thesis "Hardware Implementation of an Invariant Observer" can also be considered as a RVU, it evaluates the execution of a SUT and checks it for invariance conditions. My observer is an alternative implementation of the invariant observer INVARIANT-SYMBOL published in [4], that bypasses the problem of resource limitation and makes use of the significant advantages of a highly parallel **Field Programmable Gate Array(FPGA)** hardware implementation. The most important difference is that my observer is not bounded to a specific τ, but the observers in [4] are bounded.This feature will be explained in the next section.

In the publication "**Real-Time Runtime Verification on Chip**" [4] the concept of a RVU and the principles of that Verification Framework are described in greater detail.

A survey about the functionality of the invariant observer is given in the following sections.

1.2 The Invariant Observer

This section is a survey about the invariant observer and how it works. More details about the observer algorithm are presented in the next chapter.
The Invariant Observer acts like the temporal (invariant previously) operator $\boxdot_\tau \phi$ of the Metric Temporal Logic (MTL) and is certainly restricted to the past (ptMTL). Such a temporal operator takes an input ϕ, the calculation of a propositional formula, and evaluates if ϕ holds for the past τ execution times, including the current execution time in a discrete time setting. For example the logical consequence $e^n \models \boxdot_3 \phi$ expresses that the operator $\boxdot_3 \phi$ is true at current time n iff (if and only if) the evaluation of ϕ is true now and was also true the last $\tau = 3$ execution times. In fact the $\boxdot_\tau \phi$ is a specialization of the $\boxdot_{[0,\tau]} \phi$ ptMTL operator which restricts the range of the invariance qualification.
Figure 1.1 and Figure 1.2 show an example for such a temporal operator.

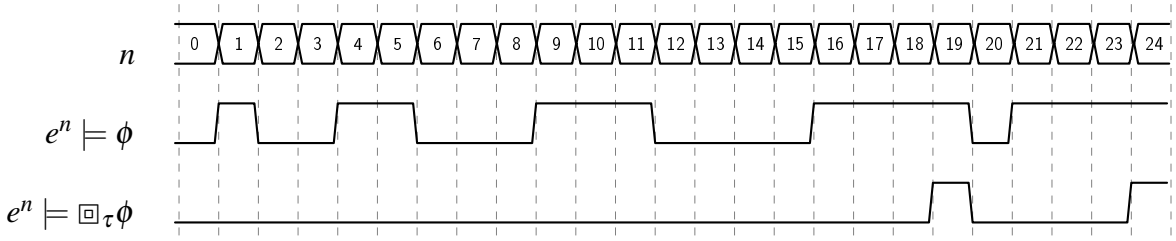


Fig. 1.1 Example for Invariant Operator $\boxdot_\tau \phi$ with τ=3

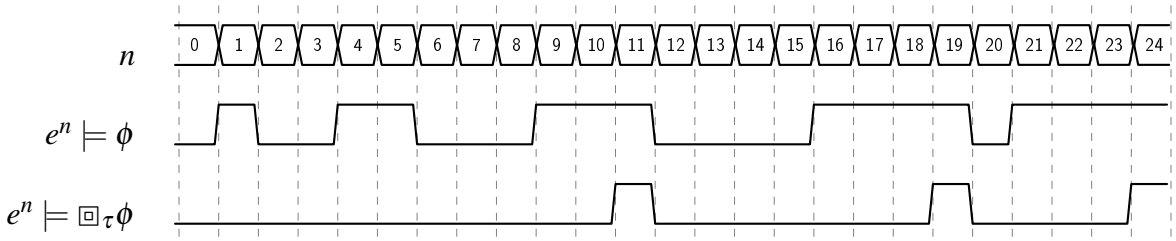


Fig. 1.2 Example for Invariant Operator $\boxdot_\tau \phi$ with τ=2

My approach of the invariant observer is based on some certain requirements. The following subsections will discuss these requirements.

1.2.1 Arbitrary calculation time of logical propositions

To introduce the first requirement we begin with the discussion of the problem that the calculation of a propositional formula ϕ could take several clock cycles (execution times). This means that an observer has to wait until the calculation of the proposition is finished. In [4] the observer needs to guarantee that it finishes evaluation of atomic propositions within a tight time bound. In our case, if we start calculation of a propositional formula ϕ at every clock cycle and the calculation itself needs y clock cycles, than we need at least y observer stages to cover finished calculations at every clock cycle. These observer stages are part of the whole observer. After y clock cycles, at every following clock cycle, a calculation of ϕ will be available. At least one observer stage will be ready, too, to evaluate a calculation ϕ at any time. In other words, we are implementing temporal pipeline stages that represent components of the invariant operator and these components together are evaluating the invariance qualification of the proposition ϕ. We don't have one observer, in fact the observer is composed from several observer stages. As a matter of fact, this solution only requires the calculation time of a proposition ϕ to be bounded by some previously known time y.

Propositional formulas can be composed from several other complex propositional subformulas or atomic propositions. In some cases a subformula is waiting for the resolution of an another subformula. In [4] this balance is achieved by the restriction of the atomic proposition class in sense of the abstract domain of logahedron.

1.2.2 Pipelined Observer Stages

Consider every finished calculation of a propositional formula ϕ as a signal value of the signal W(ϕ), every value in that signal is timely ordered just in the order it was calculated. Obviously, every represented execution time of W(ϕ) is the same as the execution time of the Observer. This means, at every execution time (clock cycle) an observer stage is evaluating a signal W(ϕ) at that time. In our case, signal W(ϕ) is apparently shifted by y clock cycles to the right. This view is encouraged by the fact, that at the beginning of the monitoring, the observer stages have to wait, until the first valid value of the signal W(ϕ) is available for evaluation of any observer stage which is duly put at disposal. The following observer stage evaluates, at execution time $n = y+1$, the second valid value of W(ϕ), and so on and so forth. It should be considered, that the evaluation of a signal value from W(ϕ) relates to a propositional formula ϕ, which was relevant y clock cycles before. But it should also be mentioned that the signal values between execution time $n = 0$ and $n = y-1$ are evaluated

as well, but obviously with a negative result, because no calculation can be started before any input is available.

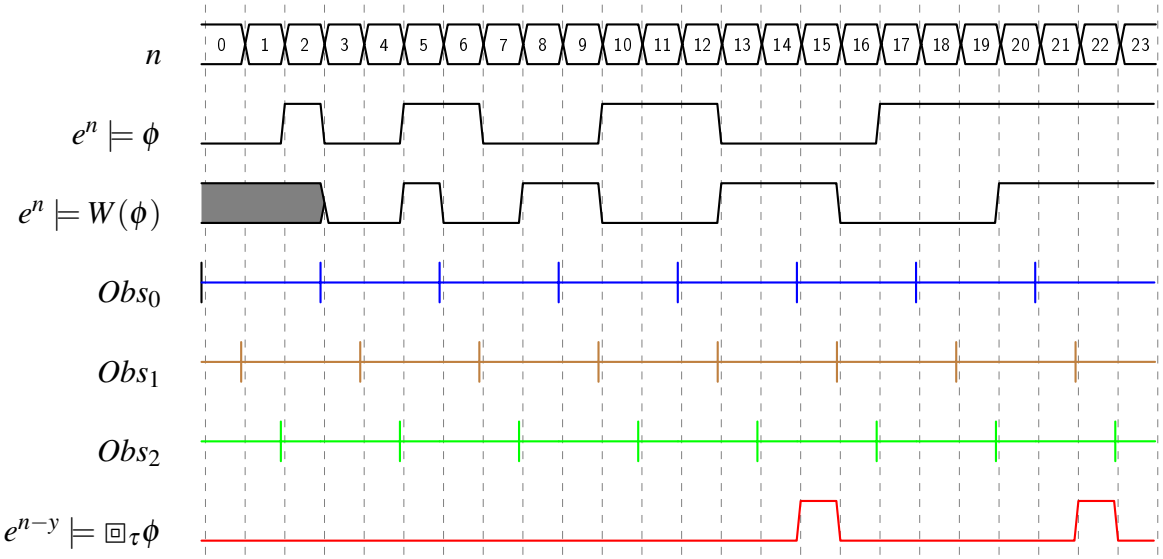


Fig. 1.3 Example for m=3 observer stages monitoring a signal W(ϕ), τ=2

The example in Figure 1.3 shows that the calculation of the first value from W(ϕ) needs y execution times. So no value is defined in the cycles before. If we take the view only from the proposition ϕ as a signal, then we calculate at every execution time n, from the latest inputs and subformulas at that time, exactly the proposition ϕ at that moment. But this holds under the assumption that the result of such a calculation is available immediately. The signal W(ϕ) shows us, what happens if calculations of proposition ϕ need some time, here we assume y=3 clock cycles. As mentioned, W(ϕ) is like signal ϕ shifted to the right. In Figure 1.3 we also see at which execution time the observer stages evaluate the signal W(ϕ). But this is only a preview about these observer stages. The algorithm about their real behaviour will be explained in the next chapter. The most important thing here is that these observers are working delayed. Every new observer stage starts its work after the observer stage before. This is important, because every clock cycle must be covered as long as the calculation of W(ϕ) takes. The last signal shows us the invariance qualification $e^{n-m} \models \boxdot_\tau \phi$ for τ=2, which holds at execution time $n = 15$ and $n = 22$, but either for one cycle only.

The invariance qualification will be resolved if we connect every observer stage in a binary “and” operation. In Figure 1.3 the result is simply calculated with:

$$\boxdot_2 \phi = Obs_0 \text{ and } Obs_1 \text{ and } Obs_2$$

Summarized, every observer stage is included in a binary "and" operation and all these observer stages together are computing at every execution time n , if the logical consequence $e^{n-m} \models \boxdot_{\tau}\phi$ holds. If we consider all these observer stages together as one temporal (invariance before) operator, than this operator checks at every execution time n, the invariance qualification of the proposition ϕ, m clock cycles before. Regarding Figure 1.3, it is true now if the proposition ϕ, m clock cycles before, was invariant with τ=2 .

1.2.3 System Settings of the Invariant Observer Stages

This was a brief introduction about the specific behaviour of the invariant observer as a whole. We will next detail on the parameter settings. It starts with the question of how many observer stages do we actually need ? We already know that at least y are necessary. If we have to define a number of observer stages with variable ”m” then the following condition must be hold: $m \geq y$. Depending on how long the calculation of a proposition ϕ needs, a minimum of y observer stages are necessary, but upwards can be chosen arbitrarily. This shows us also the possibility to apply the Observer with his observer stages on different evaluations of system inputs despite the time they need and it works in real time. If we use only one observer stage (means immediate evaluation of ϕ), then it works as a native invariant operator.

The next interesting aspect, and maybe the most powerful argument of this design way, is the invariance parameter τ. The number of observer stages ”m” stays in no relation to the invariance parameter τ. Following conditions are possible: $m \geq \tau$ or $m < \tau$. As long as the observer is configured to cover the computation time of W(ϕ), it can be used for every arbitrary choice of τ. In [4] this setting is fixed. My implementation of an invariant observer is able to change that parameter during run-time, but this feature is not specified in the requirements, and should be treated as such.

The next chapter shows the algorithm of the observer stages and how every observer stage works. Also a representation of the software implementation will be explained in great detail.

Chapter 2

Software Implementation and Algorithm

2.1 Algorithm of the Invariant Observer Stages

The following algorithm shows the proper behaviour of an observer stage.

Algorithm 1 Pseudo Code of an Observer Stage

Require: Precondition: $m \geq y$

```
1:  Initialize: count = 0
2:  if (clock mod m) = 0 then
3:      if W(φ) = 0 then
4:          /*evaluates finished calculation of φ after m clock cycles*/
5:          count = 0
6:      else
7:          /*do nothing*/
8:      end if
9:  end if
10: /*Following code executes every clock cycle*/
11: if count = τ + 1 then
12:     output = 1
13: else
14:     output = 0
15: end if
16: count = min(count + 1, τ + 1)
17: return output
```

As shown in Algorithm 1 the algorithm is split in two main parts. From the start of the observer stage, and periodically every m clock cycles, the upper part checks the status of the current signal value W(ϕ). If W(ϕ) has an active state (e.g. W(ϕ)=1), the counter keeps his

old value, otherwise it will be set to zero. It is important that one observer stage recognizes, that the invariance qualification was not satisfied at this time. When the conjunction of all observer stages is computed, at any arbitrary execution time n, and at least one stage does not have an active output, then the result is false. This means, that at the execution time n, the invariance qualification is not fulfilled. The bottom part is executed at every clock cycle and increments the counter value up to the maximum value of the invariance qualification's range. If the counter reaches the maximum value, the respective observer stage activates its output to an active state. In fact, the counter represents the invariance qualification of length 'τ'. The term '$\tau+1$' indicates that the present value must also be involved in the invariance qualification. The counter value will be initialized with zero at the beginning of the algorithm. Hypothetically, if the counter is initalized with '$\tau+1$' the ouptut is activated immediately, because of the bottom algorithm. But this is a contradiction to the assumption that W(ϕ)=0 for all execution times n before zero.

It should be mentioned that this current design does not implement or handle the calculations of the truth value of propositions ϕ, which is indicated with W(ϕ). On the other hand, the observers from [4] are responsible for taking the necessary inputs, calculate the atomic propositions (with ATCheckers) and immediately evaluate the ptMTL-operator qualifications. These steps have to be done in a tight time bound. In our case, W(ϕ) must be updated from another entity in such a way, that at every clock cycle an observer stage must have a consistent value for evaluation. The following subsection is an overview about the implementation of the Algorithm 1 in VHDL.

2.2 VHDL Implementation of the Algorithm

In Appendix A.2, there is an implementation in VHDL which follows the meaning of Algorithm 1. We will discuss the different process entities and the relations with Algorithm 1. Finally, we get an overview about the improvements which are significant for a faster design. This VHDL design of an observer stage has got the following inputs and outputs:

(a) inputs:

- **invariance_tau** is a signal variable which gets the value for τ
- **enable_in** signal activates the observer stage
- **signal_phi** is the signal state of W(ϕ)

(b) output:

- **enable_out** is a signal whose state is set to active after the activation of the current observer stage, but delayed for exact one clock cycle.
- **output signal** is simply the output state of the observer stage.

The Observer Stage is split in a synchronous and an asynchronous design, in terms of a Moore State Machine. The process labelled “sync” (at line 83 of the VDHL-Code A.2)represents the synchronous part of that design where the register states are changed at every clock cycle. The synchronous process is only executed if the combined signal **enable_logic** has been activated, indicating a specific behaviour of the observer stages. (VDHL-Code A.2, line 22) The observer stages are linked to each other through cascade connections. The first stage activates the next observer stage through the signal **enable_out** after being activated through **enable_input**. If the signal **enable_input** of an observer stage is being activated,in a clock cycle, then the signal **enable_out** will be activated in the clock cycle afterwards.

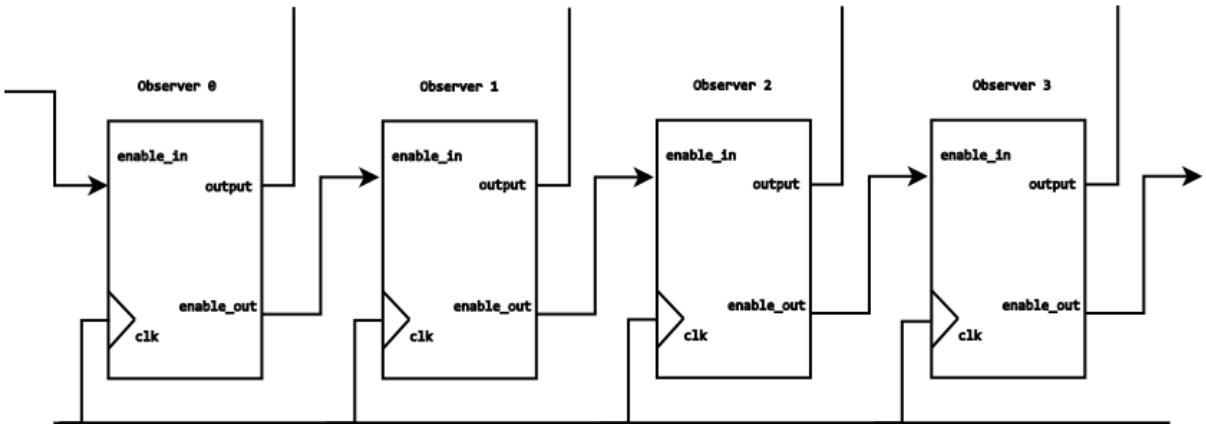


Fig. 2.1 Example for m=4 Invariant Observer Stages concatenated in cascade.

This ensures, that these observers are starting delayed, each 1 clock cycle after its predecessor.
Signal **inc_tau**(VDHL-Code A.2, line 21) increments the input signal **invariance_tau**, which represents $\tau + 1$ at every execution time. The asynchronous part works immediately after every change of the system state. (VDHL-Code A.2, line 25 and line 51) In the following descriptions of the asynchronous parts, only temporary signals are changed, but

these signals transfer their values inside the synchronous part at every clock cycle, except **inc_tau** and **enable_logic**. Every of these signals are indicated with "_next" at the end of their names, but in the following descriptions only their register counterparts will be mentioned.

The process entity **comb_cycle**(VDHL-Code A.2, line 25) is the description of an internal clock counter which only counts up and down between values 0 and m. The signal **cycle** is significant for checking the condition "(clock mod m) = 0" from Algorithm 1.

The process entity **comb_logic** implements the main part of the algorithm. Signal **count_p** will be incremented in each clock cycle until it reaches $\tau + 1$. The counter should be initialised to zero (as indicated in the algorithm), but is incremented immediately at every clock cycle. (VDHL-Code A.2, line 55) Or in other words, if **count_p** is reset to zero in a clock cycle, it will be incremented immediately in the "same" clock cycle. To demonstrate this fact we have an additional signal **count**, initialised with 1. But this signal is only for reasoning and will be reduced by the syntheses tool. The synchronous design is cumbersome in a way, which is due to the way of how the states change. If the counter is supposed to be incremented after evaluating some conditions, this does not imply an immediate change of the counter. To overcome this handicap the counter **count_p** is initialised to 2(VDHL-Code A.2, line 56) which means, that at every clock cycle we check if the counter count_p will reach a maximum at the next clock cycle. This enables us to change things on time. According to the Algorithm 1 at line 5, **count_p** will only be reset if signal $W(\phi)$ has been evaluated as being not active in **comb_cycle**. (VDHL-Code A.2, line 56)

The first if-branch at line 53 of the VDHL-Code A.2 checks two cases. At first it will be checked if the clock passed m cycles or not and second if the signal **enable_logic** is supposed to be active. The first question is in term of Algorithm 1, line 2. The second question concerns the signal **enable_logic**. This signal combines **enable_in** and **reset** in one signal. If signal **enable_logic** is active, means that the observer stage is active and no reset condition happens. The else-branch of the first if-branch at line 67 of the VDHL-Code A.2 checks the condition in terms of Algorithm 1, line 11. The content of the if-branch (line 53) and the else-branch (67) are nearly the same, but there are two exceptions. Inside the if-branch, beginning at line 54, the status of signal W(ϕ) will be checked, according to Algorithm 1, line 3. And inside the else-branch at line 75, if m cycles are no yet passed and signal **count_p** is already at the maximum, then signals **count_p**, **count** and **output** keep their old values. The

other parts of **comb_cycle** are straightforward, if you compare it with the algorithm. In case the counter **count_p** reaches the maximum, the **output** of the observer stage is activated.

The current design has been made optimizing for throughput. Besides from design decisions like using count_p instead of count, this led to the following general design rules:

It is very important that no Latches are built by the syntheses tool, so every if-branch must contain the same changes on the same signals. A further point is to reduce the number of if-branches to a minimum. If-branches inside of an If branch extends signal paths and reduce the maximum clock design of the whole design. In the next chapter we get an overview of the hardware realisation of the current design, which shows us a more visual view on that.

Chapter 3

Hardware Implementation

The following sections show an overview about the hardware implementation of the Invariant Observer Design. It starts with a description about the hardware platform on which the observer runs and some details about the synthesis tools. At the end of this chapter, there comes a design view about the synthesized hardware and a discussion about the design steps.

3.1 Hardware Platform for a Prototype

The Invariant Observer is synthesized on a Field Programmable Gate Array(FPGA), on this platform it was possible to get an insight about the runtime behaviour and to see the observers in action. The FPGA Board that was used for the simulations is a **DE2-115** Board from ALTERA[1]. This FPGA Board is ideal to illustrate fundamental concepts and advanced designs, and it gives a possibility to meet the necessary real-time requirements. The DE2-115 is equipped with a **Cyclone IV EP4CE115F29** FPGA Chip and it is powerful enough to emulate a CPU(Nios). In [4] this FPGA board was used for performance studies, besides other FPGA's, so it was logical to use a similar environment. The Board has an on-board oscillator of 50 Mhz, and with the use of a Phase Locked Loop(PLL) it is possible to increase the working frequency for tests. In the following section it will be shown how the increase of the frequency changes the design on the Register Transfer level(RTL) and why bad design decisions can influence the maximal speed of the design. In Appendix C there is a schema illustration about the components of that FPGA board.

3.2 Synthesis and Design

This section gives an overview about the synthesis tool and details about the Register Transfer Level design view. An interesting part in this section is to see how the synthesis tool creates hardware structures based on the code of the Observer VHDL implementation.
For synthesis and compilation of the observer VHDL code, the **Altera Quartus II Version 12.1 Build 177 Full Version** was used.

3.2.1 Design View of the Register Transfer Level

In Quartus II, after compilation of the design and after creation of the netlists, you can use a tool named **RTL Viewer**, which shows how the components of the design will be created on a Hardware Platform. It is an abstract view on the **Register Transfer Level**. This tool can be very handy if you want to see how the hardware should look like. Based on the illustration of the hardware, some decisions can be reconsidered in terms of the performance.
Figure 3.1 is a hardware plan of a single observer stage (m=1), which is modelled as a single Observer with input frequency of 50Mhz. In which way, this test cases was modelled, will be discussed in the next chapter. The next Figure 3.2 shows exact the same design case like Figure 3.1 but with the difference that the input frequency is 150Mhz. Here it is illustrated how the syntheses tool reorientates the hardware design, to meet the requirements of that clock speed. The worst case path determines the maximum frequency, which is allowed for that design. A more detailed description about the timing model is in the next subsection.

3.2.2 Timing Model of the Design

One important thing that have to be considered is that the signal path from the beginning of the observer entity to the end should be as short as possible. Lets overview some important design facts.
The maximum frequency of the whole system is limited by the longest path of the design. This results that the performance of a system is expressed by the maximum frequency at which it may be operated. It is logical that, at every entity in the netlist, the path should be as short as possible. The performance maximum can be computed by the Quartus Tool named **TimeQuest Timing Analyzer**, which gives estimations about the maximum frequency of the design. The longest signal path of the design is indicated by the **worst case path** in the tool, as mentioned in the last subsection.
The first version of the observer stage had a low performance level. This version is illus-

trated in the Appendix B.2. The TimeQuest Timing Analyzer consider some uncertainties for the performance estimations. From [2] it is known how the Quartus Tool creates timing models which are build in the TimeQuest Timing Analyzer after compilation of that design. A FPGA have to operate in a continuum of conditions. These conditions include the die junction temperature, which varies. Commercial parts have a range of 0°C to 85°C and industrials a bigger range. A second condition aspect is the voltage supply level. Critical voltages for maintaining FPGA performance is the Vcc and the various I/O supplies.
The timing analyzer shows estimations for three cases

1. Slow 1200mv 85°C model
2. Slow 1200mv 0°C model
3. Fast 1200mv 0°C model

The operating-condition corners are usually the combinations of end points of the ranges in temperature, voltage and manufacturing processes. Each operating-condition is used to model the timing delays under specific end point of temperature, voltage, and manufacturing process conditions. The first case with "Slow 1200mv 85°C model" seems natural and should be considered for our test cases, because it shows frequencies under long term operating conditions. A lot of tests about estimations of the maximum frequency of different design cases, are always took from that "Slow 1200mv 85°C model", and have to be considered for further discussions. In the next chapter these tests will be discussed.

3.2.3 Design Decisions based on Register Transfer level

After the first time, the VHDL Code of the observer stage was behavioural correct, some performance and optimization decisions had to be reconsidered. The most important thing is to separate synchronous and asynchronous design. At the beginning, it was necessary to split a completely synchronous design, where every action is triggered after every clock cycle. A more parallel approach has to be done where only state changes are done synchronously, but other actions should be triggered immediately after a change of a system state. And the asynchronous system part can separated in several independent asynchronous components, which achieves more system parallelism and a better performance. It should be mentioned again, from chapter 2, that this system design is similar to a Moore State Machine, where the logic is separated in a transition logic, an output logic and a state memory. And only the state memory is changed on every clock cycle.
Our design approach of the invariant observer is similar, but with the difference that the

output logic contains transition logic components. The hardware schemas from the Register Transfer Level plans where beneficial, because it shows which components where created from the VHDL background. If statements should be created in a way, that allow no Latches. This means that every else branch should contain the same components like the if branch. If you follow that rule, you can see for example the created mutexes in Figure 3.1. Every mutex has two inputs(according to if, else), which are selected depending on the third input(MUX21).
At the beginning of the entity there are several adders and these are representations for the different additions which has to be done according to the behaviour of the observer. **Add2** emulates "$cycle\_next = cycle + 1$", **Add1** emulates "$cycle\_next = cycle - 1$" and **Add3** emulates "$count\_p = count\_p + 1$" according to the sourcecode of the Observer Stages in Appendix A.2. In Figure 3.1, the MUX 21 Mutex at the beginning, indicated by **cycle_next[15..0]**, is the if-branch, according to the sourcecode in Appendix A.2, which decides if "$cycle = 0$" or not.
The **MUX21[cycle_next[31..16]]** below stands for the branch "if $cycle = observernumber$". **MUX21[cycle_next[15..0]]** decrements cycle_next in normal case, but increments it only once at the bottom border. **MUX21[cycle_next[31..16]]** works the opposite way. As you can see, the tool enhanced cycle_next up to 48bit to reserve memory for intermediate results.
In the hardware scheme, there are two another **MUX21[cycle_next[15..0]]**, the first(left to right) decides between "$cycle\_next = cycle + 1$" or "$cycle\_next = cycle - 1$" and the second between the result from first Mutex before and the reset value. **MUX21[cycle_next[47..32]]** decides between keeping the old value "$cycle\_next = cycle$" or taking over the value from **MUX21[cycle_next[31..16]]**. The overview is similar to count_p, but a deeper insight about the Quartus translation behaviour would exceed that subsection. As you can see in the hardware schema, there are 5 registers(cycle, count_p, direction, output, enable_out) and these representates the state of the Invariant Observer and they may only change their values after every clock_cycle. The dimensions of the registers are conform to the dimensions of the input conditions. For example if inc_tau has a value with only 5 bits, then count_p is accordingly large-dimensioned to 6 Bit. This is only a cut-out, about how the Quartus II translates VHDL Code. A lot of other hardware representations are shown in Appendix B, actually they have the same components but another orientations. The next subsection shows something about problems during the progress of that development.

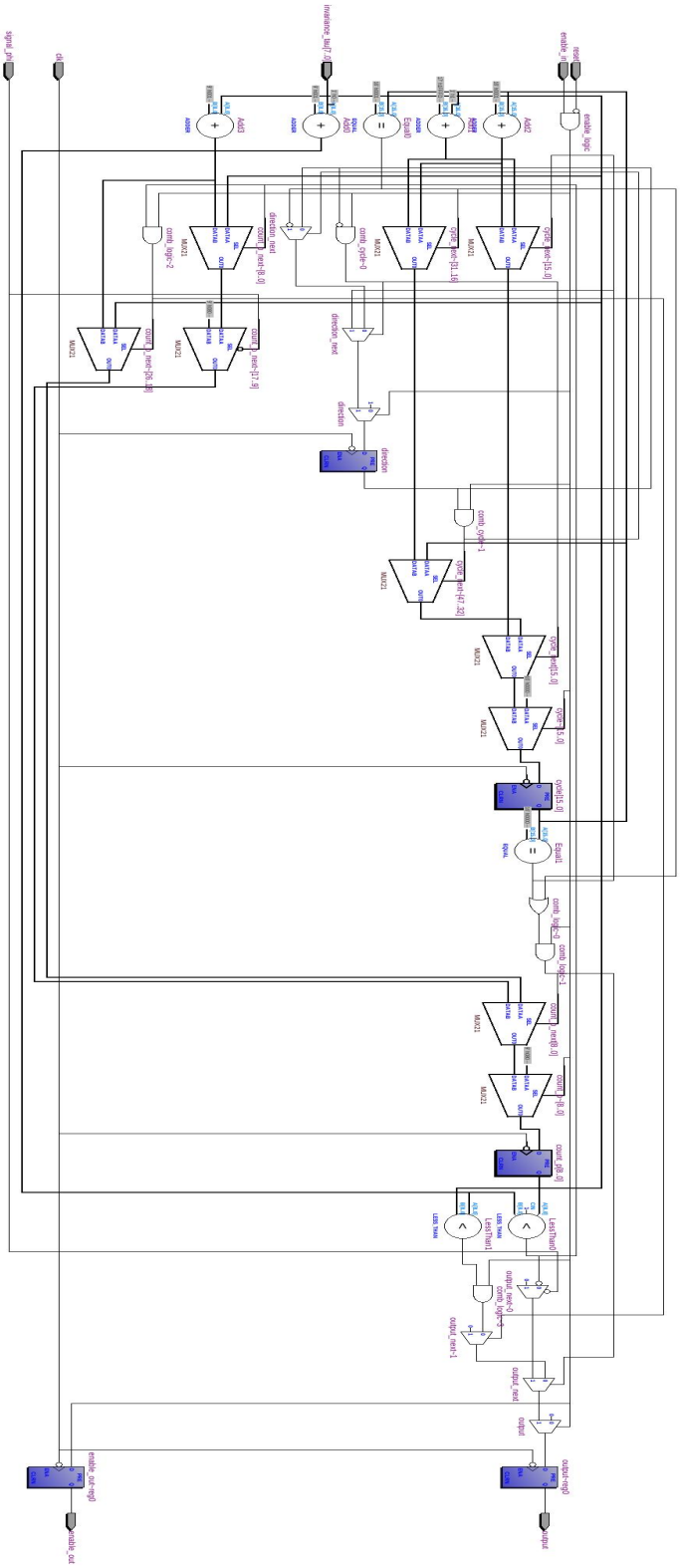


Fig. 3.1 Shows a RTL View of a single observer stage with input clock of 50Mhz

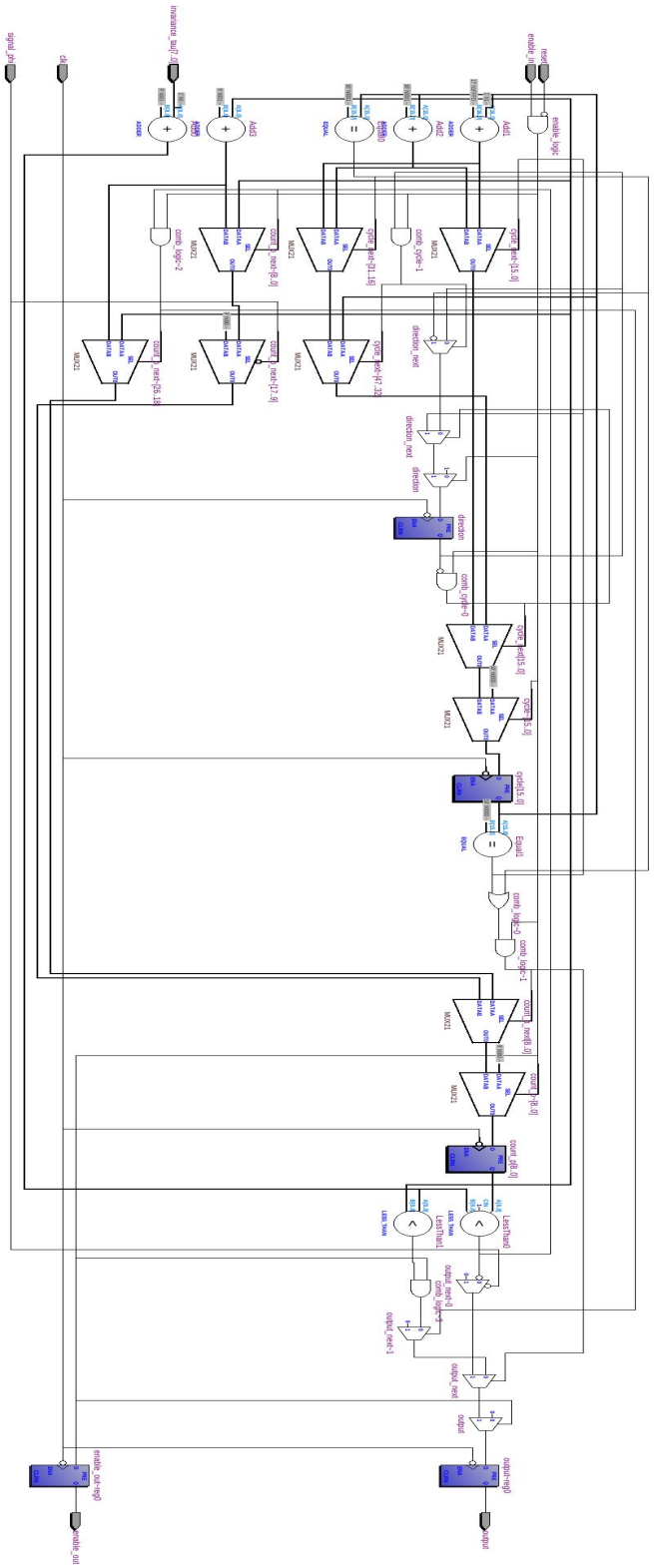


Fig. 3.2 Shows a RTL View of a single observer stage with input clock of 150Mhz

3.3 Design problems in the development

This subsection is only a kind of a protocoll about problems during the development.

1. The starting condition of the Algorithm 1 was uncertain, the question was, should variable count be initialised with 0 ot with $(\tau+1)$. If count=$(\tau+1)$ then the output is immediately switched on, which correspond to an invariance qualification. From the specification of an observer in [4], it is known from the proof of Theorem 1(in [4]) that $\forall i : (i \in [max(0, n-\tau), n] \rightarrow e^i \models \phi)$ which follows that at the beginning of an execution if $ni-\tau$ exceeds th bottom border, but the signal ϕ is true, the invariance qualification is fulfilled by that specification. This condition only holds if signal ϕ is knows at that time, in our case the status of signal ϕ is always(if $m \geq 1$) taken from the past. So it is not known which status the signal ϕ has before execution time e^0. Hence, the Algorithm 1 was initialised in a manner, that no decisions can be made about invariance qualifications.

2. The Invariance Observer was tested with a self-made signalgenerator which imitates input ϕ. As a result, two clock domains were created, although the clock signals came from the same PLL(Phase Locked Loop). If one clock is an "even" multiple of the other clock frequency, there was no problem, but if the clock frequencies are choosed in a different way, then it comes to clock drifts.

3. The file (*.sdc file)for timings must always be adjusted, if there are changes in one of the clock domains. The PLL output timings have to be analyzed separately by the TimeQuest Timing Analyzer. If this does not happen, false estimations are created for "Slow 1200mv 85°C model" maximum frequencies and worst case paths.

4. The first download of the compliated design to the FPGA was no succesfull, because the Quartus Tool was not correctly adjusted to that plattfom. Following points have to be changed:

   - The correct architecture must be set
   - the voltage must be set for all inputs and outputs

5. Always consider what is the active state of an input component or output component, for example input KEY is low active and was used for reseting the board.

Chapter 4

Experiments and Testing

References

[1] Altera de2-115 fpga-board description. http://www.altera.com/education/univ/materials/boards/de2-115/unv-de2-115-board.html, . Accessed: 2014-04-24.

[2] Altera timing model. ttp://www.altera.com/literature/wp/wp-01139-timing-model.pdf, . Accessed: 2014-04-24.

[3] A.P. Ravn, H. Rischel, and K.M. Hansen. Specifying and verifying requirements of real-time systems. *Software Engineering, IEEE Transactions on*, 19(1):41–55, Jan 1993. ISSN 0098-5589. doi: 10.1109/32.210306.

[4] Thomas Reinbacher, Matthias Függer, and Jörg Brauer. Real-time runtime verification on chip. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification*, volume 7687 of *Lecture Notes in Computer Science*, pages 110–125. Springer Berlin Heidelberg, 2013.

Appendix A

Source Implementation in VHDL

The following source Codes in this Appendix A shows the implementation of the Invariant Observer. The Source Codes are written in VHDL 1993 and shows a description about the Behavioural Specification of the design.

Observer.vhd is the entity specification of an Observer Stage and shows specifications about the input and outputs. Figure 2.1 shows us an excerpt of the input and output but at most how the several observer stages interact together.
Observer_behave.vhd describes the exact behaviour of an observer stage. A detailed description of this behaviour is shown in Chapter 2.2.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.all;
3  use IEEE.NUMERIC_STD.ALL;
4
5
6  entity observer is
7  generic (
8    observernumber:unsigned(15 downto 0):=x"0001"  -- how many observer are instan
   tiated
9        );
10 port (
11
12   clk          :in   std_logic;
13   reset        :in   std_logic;
14   enable_in    :in   std_logic;
15   invariance_tau  :in   std_logic_vector(7 downto 0);
16   signal_phi      :in   std_logic;
17   output    :out  std_logic;
18   enable_out      :out  std_logic
19   );
20
21 end entity;
22
```

Fig. A.1 Code of Observer Entity Design

```
1
2   LIBRARY ieee;
3   USE ieee.std_logic_1164.all;
4   use IEEE.NUMERIC_STD.ALL;
5
6
7   architecture Behavioural of observer is
8
9   signal inc_tau                          : unsigned(8 downto 0):= "000000000";
10  signal count,count_next                 : unsigned(15 downto 0):= x"0001";
11  signal count_p,count_p_next             : unsigned(15 downto 0):= x"0002";
12  signal cycle,cycle_next                 : unsigned(15 downto 0) := x"0000";
13  signal direction,direction_next         : std_logic := '1';
14
15
16  signal enable_logic                     : std_logic := '0';
17  signal output_next                              : std_logic := '0';
18  begin --BEGIN ARCHITECTURE
19
20  -- parallel logic
21  inc_tau <= unsigned(invariance_tau) + to_unsigned(1,9) ;
22  enable_logic <= enable_in and not reset;
23
24  -- changes cycle up from 0 to observernumber and down back to 0
25  comb_cycle: process(cycle,direction,enable_logic)
26  begin --changes cycle_next, direction, changeDirection
27    if(direction = '0' and enable_logic = '1') then
28      if(cycle = 0)then
29        direction_next <= '1';
30        cycle_next <= cycle + 1;
31      else
32        direction_next <= '0';
33        cycle_next <= cycle - 1;
34      end if;
35    elsif(direction = '1' and enable_logic = '1') then
36      if(cycle = observernumber)then
37        direction_next <= '0';
38        cycle_next <= cycle - 1;
39      else
40        direction_next <= '1';
41        cycle_next <= cycle + 1;
42      end if;
43    else
44      direction_next <= direction;
45      cycle_next <= cycle;
46    end if;
47  end process comb_cycle;
48
49
50   -- main logic of the observer
51  comb_logic: process(inc_tau,count,count_p,cycle,signal_phi,enable_logic)
52  begin
53    if ( (cycle = observernumber  or cycle = 0) and enable_logic = '1') then -- m
    cycles passed
54      if(signal_phi = '0') then   -- w(phi) = 0
55          count_next   <= x"0001";
56        count_p_next <= x"0002";
57        output_next  <= '0';
58      elsif(count_p <= inc_tau) then
59        count_next   <= count   + 1; --every clock cycle
60        count_p_next <= count_p + 1 ;
61          output_next       <= '0';
62      else
63        count_next   <= count;
64        count_p_next <= count_p;
65        output_next       <= '1';
66      end if;
67    elsif(count_p <= inc_tau and enable_logic = '1') then
68      count_next   <= count + 1; --every clock cycle
69      count_p_next <= count_p + 1 ;
70        output_next       <= '0';
71    elsif(count_p > inc_tau and enable_logic = '1') then
72      count_next   <= count;
73      count_p_next <= count_p;
74      output_next  <= '1';
75    else
76      count_next   <= count;
77      count_p_next <= count_p;
78      output_next  <= '0';
79    end if;
80  end process comb_logic;
81
82    --the synchronisation logic
83  sync: process(clk,enable_logic)
84    begin
85      if(clk'event and clk = '0')then
86        if(enable_logic = '1') then
87          cycle           <= cycle_next;
88          direction       <= direction_next;
89          count           <= count_next;
90          count_p            <= count_p_next;
91              output    <= output_next;
92          enable_out      <= '1';
93        else
94          cycle           <= x"0000";
95              direction       <='1';
96          count   <= x"0001";
97          count_p <= x"0002";
98              output    <= '0';
99              enable_out <='0';
100       end if;
101     end if;
102   end process sync;
103 end architecture; --END ARCHITECTURE
```

Fig. A.2 Code of the Behavioural Design of an Invariant Observer

Appendix B

Register Transfer Level Design View

B.1 Design of the First Version

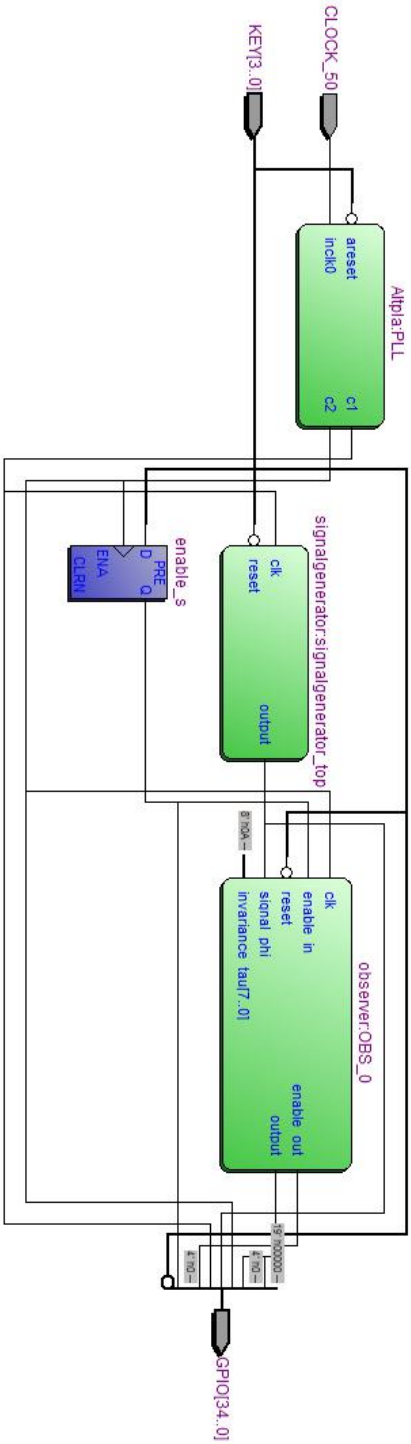


Fig. B.1 Illustration of the TOP Design of the first correct version,but without improvements

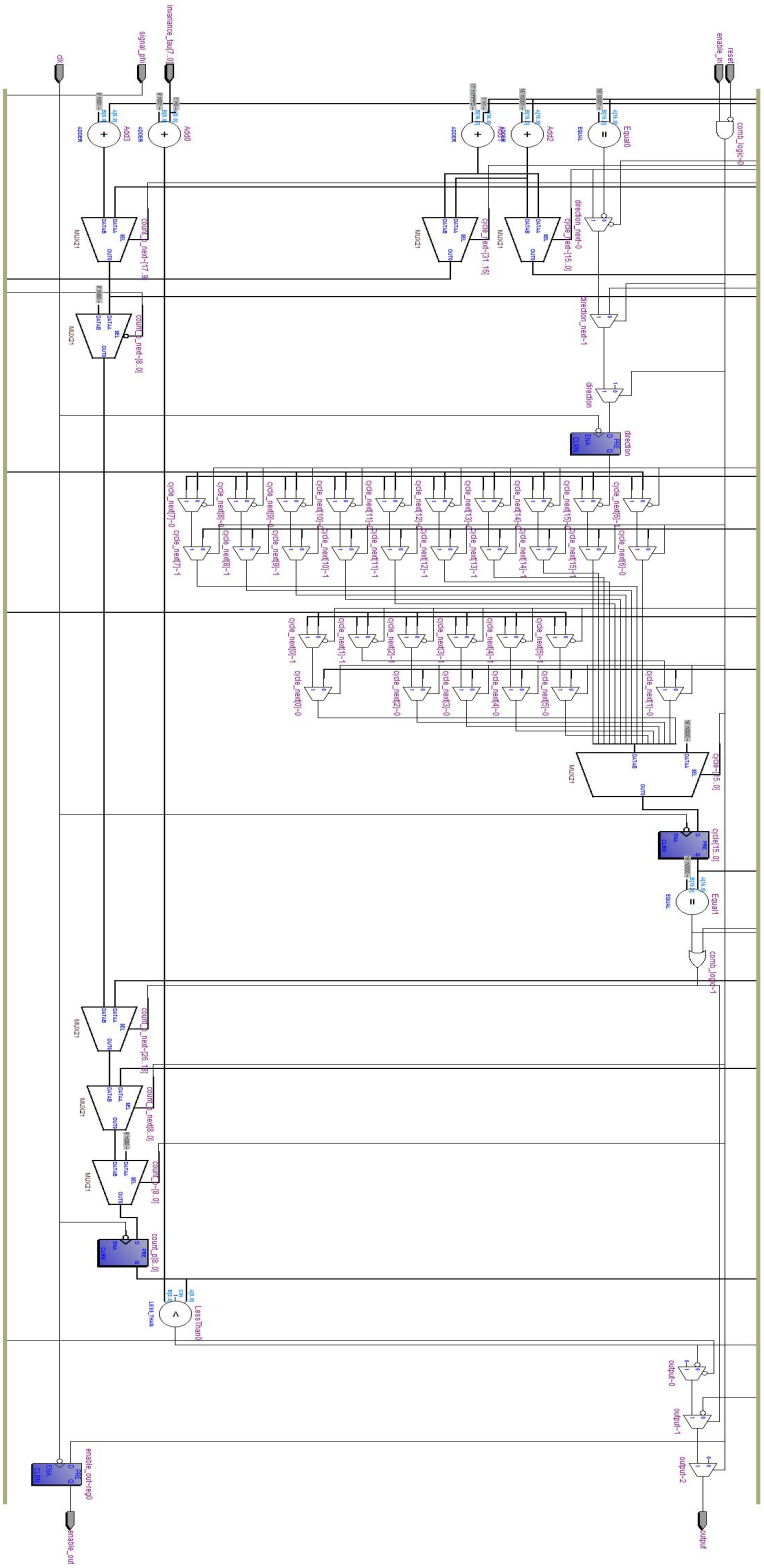


Fig. B.2 Illustration of the Observer Design of the first correct version,but without improvements

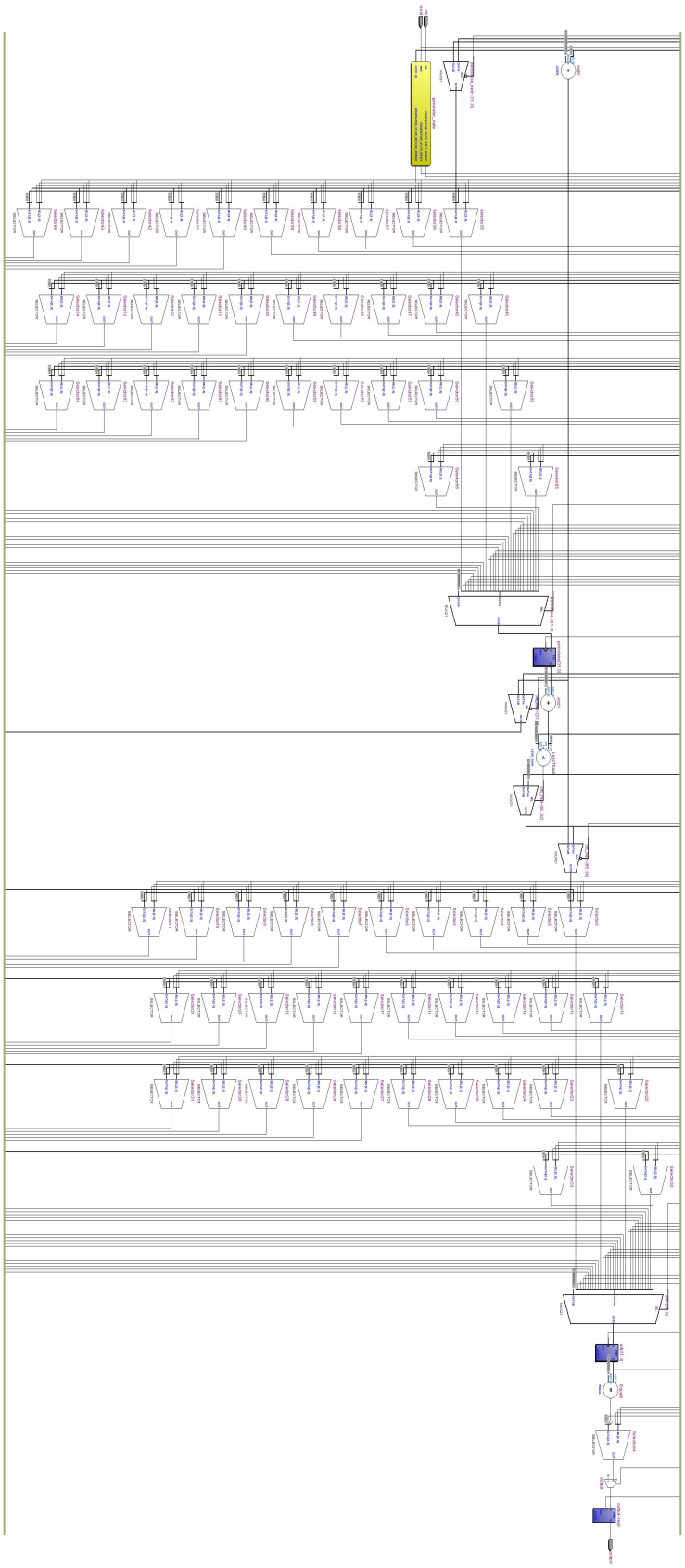
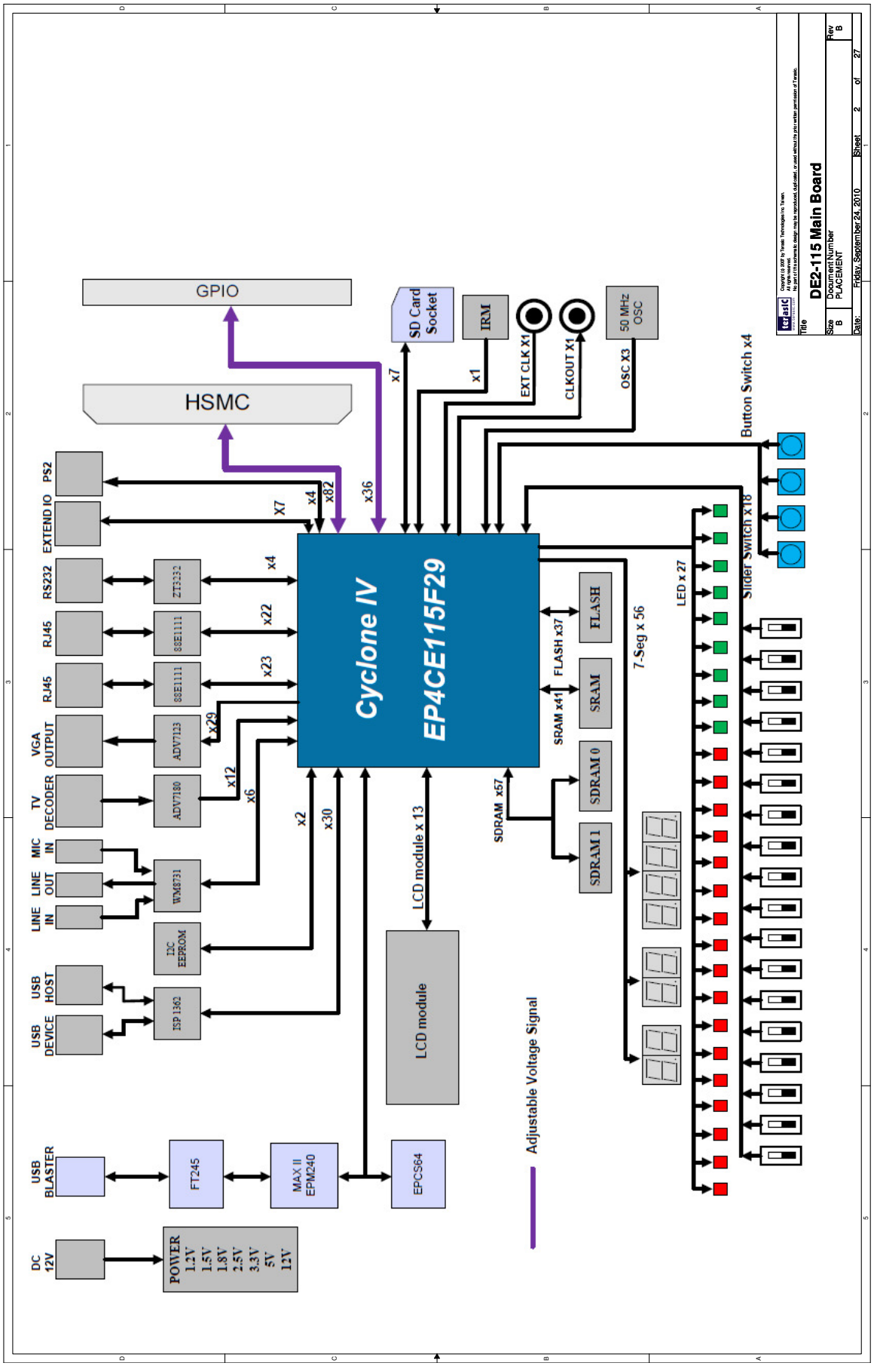


Fig. B.3 Illustration of the Signalgenerator of the first correct version,but without improvements

Appendix C

Datasheets and Descriptions


GPIO
HSMC
SD Card Socket
IRM
EXT CLK X1
CLKOUT X1
50 MHz OSC
OSC X3
Button Switch x4
Slider Switch x18
LED x27
7-Seg x56
Cyclone IV
EP4CE115F29
FLASH
SRAM
SDRAM 0
SDRAM 1
FLASH x37
SRAM x41
SDRAM x57
LCD module x13
LCD module
PS2
EXTEND IO
RS232
RJ45
RJ45
VGA OUTPUT
TV DECODER
MIC IN
LINE OUT
LINE IN
USB HOST
USB DEVICE
ZT3232
88E1111
88E1111
ADV7123
ADV7180
WM8731
I2C EEPROM
ISP 1362
USB BLASTER
FT245
MAX II EPM240
EPCS64
DC 12V
POWER
1.2V
1.5V
1.8V
2.5V
3.3V
5V
12V
Adjustable Voltage Signal
x7
x1
x36
x4
x82
X7
x4
x22
x23
x29
x12
x6
x2
x30
Title
DE2-115 Main Board
Size
B
Document Number
PLACEMENT
Rev
B
Date:
Friday, September 24, 2010
Sheet
2
of
27