AN ABSTRACT OF THE THESIS OF

Gerald Lai for the degree of Master of Science in Electrical and Computer Engineering presented on August 25, 2006.

Title: Development Test Suite for FPGA TekBot Learning Platform.

Abstract approved:

____

Roger L. Traylor

As the TekBots® program expands into senior and graduate level classes at Oregon State University, so does the need arise for more complex learning platforms. These complex hardware platforms cannot be adequately tested in a manufacturing environment as we have done previously. Also, due to their complexity, these platforms require substantial collateral documentation to allow first-time users to quickly become productive learners.

This thesis details the development of a post-manufacturing test suite, known as OMICRON, to comprehensively test an FPGA learning platform. It also documents the development of a user guide for the board that explains user accessible features as well as providing the necessary startup information so students can quickly become acquainted with the new learning platform.

While developing OMICRON, a new feature surfaced that provides a cycle-accurate hardware testbench debugger for testing student component modules that are implemented within the FPGA. This functionality serves a practical as well as an educational use by enabling test generation for detecting logic errors at a hardware level.

Students can probe their own designs from an intuitive low-level command line interface once the designs have been loaded into the FPGA. The debugger can also be used to probe external circuits connected to the FPGA.

In addition to simple probes, the hardware debugger is able to output testbench bit vectors in a continuous flow, and simultaneously receive cycle-accurate vector results. These test vectors can either be manually constructed, or extracted from simulation software. This thesis shall also demonstrate this unique test flow.

Development Test Suite for FPGA TekBot Learning Platform

by

Gerald Lai

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented August 25, 2006
Commencement June 2007

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

Development Test Suite for FPGA TekBot Learning Platform

1 TEKBOTS PROGRAM AT OREGON STATE UNIVERSITY

1.1 TekBots – An engineering initiative

In what began as a simple class-based project at Oregon State University (OSU), the TekBots program [1] has grown and matured into an innovative, and effective, means of educating upcoming student engineers. Under the TekBots program, Electrical and Computer Engineering (ECE) students keep their TekBot throughout their years in college, and continue to build on them as they take additional classes. When complemented with the traditional in-class methods of learning, the TekBots program helps to capture the essence of education. For instance, intangible qualities such as a student's commitment and tenacity to excel in her or his school assignments are evident in the work of a student's TekBot.

TekBots are learning platforms that students build to reinforce engineering concepts they learn in class. Specifically, a TekBot for an ECE student currently takes the form of an electronic robot. Since the TekBots stay with the students throughout the curriculum, this encourages personal ownership and attachment of the students for their TekBots. This fact alone yields many advantages. Once students are passionate about the work that is required to build a TekBot, they work harder, learn better, and interact more with peers [2, 3]. When students are engaged in the multitude of learning activities the TekBots program has to offer, they begin to work more intensely on their projects, and are able to learn and absorb more of the lecture material.

The class project that eventually led to the creation of the TekBots program was influenced by a program to teach ECE topics in context at Carnegie Mellon University (CMU) [4, 5, 6]. The goal that Carley et. al. was trying to achieve was to teach ECE courses to first semester freshmen without relying on an extensive background in mathematics or physics. This was done by taking a behavioral, or top down, modeling approach for the devices that were taught in class. For example, the behavior of nonlinear electronic devices was characterized using piecewise linear modeling techniques. By learning these simplified behavioral models for nonlinear devices such as diodes and transistors, rather than complex nonlinear equations, they believed that students may actually develop a better insight into the operation of electronic circuits.

The CMU program focused its effort of contextual teaching around a small programmable robot. This idea was adopted at OSU as a practical laboratory tryout for a class project in a freshman-level ECE course. While working on the project, students found the hands-on experience of physically building an extensible electronic robot to be fun and fulfilling. The extensibility of the robot meant that more circuit boards were allowed to be stacked on top of the original circuit board base. This continued to engage students in improving the functionality of the robot well beyond what was intended for the project in the first place.

From the enthusiasm the students showed, OSU faculty recognized that they had stumbled upon an opportunity to integrate the dynamic learning environment, which was observed in the laboratory, to the entire curriculum. The next major step was to take the approach of incorporating the project into the existing syllabi such that it would tie together the subjects being taught into a coherent whole. This would involve having a project that evolves with the students as they progress through the curriculum.

At the same time, a proposal to Tektronix was secured for a grant of USD$500k that would allow OSU to continue its pursuit in creating a learning platform based curriculum for ECE students. This was the birth of the TekBots program at OSU in which the original robot project was given the catchy name "TekBot". The grant from Tektronix was the boost that allowed additional grants and parts donations to be subsequently leveraged towards the improvement of the program.

So far, the TekBots program has already been implemented for many engineering classes at OSU since 2002, including five Electrical & Computer Engineering classes, two College of Engineering courses, and one Mechanical Engineering class. It has also been adopted by other universities, such as Texas A&M, the University of Nebraska, and the Fukuoka Institute of Technology in Japan.

With the help of the learning platform, the TekBots program incorporates real-world instances to help students develop competencies in innovation, community, design, troubleshooting, and professionalism [2, 3, 7]. This adaptability of the Platform for Learning^{TM} concept makes it a powerful catalyst that could reinvent not only engineering education as a whole, but also education for other disciplines besides engineering. Chapter 2 of this thesis shall expound the concept and ideals for effective learning platforms in more detail.

1.2 Summary of common terms

Before proceeding further, common terms used throughout this thesis will be defined to prevent misinterpretations. A brief description for each of the terms is given below:

Platform for Learning^{TM} concept: This concept refers to the educational strategy that is comprised of ideals to teach students effectively in a dynamic learning environment. It is also known as the ***Platform for Learning^{TM} model***.

Learning platform: A learning platform is an object used to implement the educational strategy of the Platform for Learning^{TM} concept. Sometimes, it can also be referred to as a ***platform for learning***. Depending on the context, this is not to be confused with the actual concept that is a proper noun. Students usually begin with a bare bones learning platform. As they progress through their curriculum (e.g., a 4-year college program), they continue to build on it. The platform can be a physical or virtual object.

TekBots® program: The TekBots program is an engineering educational initiative that uses learning platforms as well as other proven learning methods to attain the goals of the program.

TekBot: A TekBot is one of the learning platforms presently used at OSU. It consists of electronic circuitry mounted on a sheet metal base with motors attached to wheels. This type of TekBot is loosely referred to as a robot. ECE students use this platform to apply and validate the knowledge they have acquired in class.

TekBots: The word "TekBots" has different interpretations depending on the context. It can be used to indicate more than one TekBot. It is also used to refer to the TekBots program that was created at OSU. One of the most common misinterpretations is to use "TekBots" to refer to a robotics program. The TekBots program at OSU is not a robotics educational program. However, the robot learning platform is (correctly) referred to as a robotics platform.

1.3 Goals of the TekBots program

The TekBots program is directed towards a central set of objectives. Listed below are the main goals for the TekBots program and how they are reached. For each goal, it is clear to see how the use of a learning platform is vital.

(a) Improved learning

The TekBots program aims to improve learning by adopting several proven teaching strategies. The most obvious strategy, given a physical learning platform, is to increase the retention of knowledge in students by adopting a *hands-on* learning style. By having students apply concepts learned in class towards building physical robots, the class knowledge is then "converted" into a more useful and concrete form. The physical robot becomes the result of the application of that knowledge. As long as the student remembers how the robot was constructed, the concepts acquired in class that were used to build the robot can be recalled in a natural manner.

The learning platform also enhances learning by *giving purpose* to what students have learned in every class. For example, freshmen ECE students are usually taught how to read voltage drops across resistors in a passive resistor-only circuit. A learning platform at this stage may be a simple flashlight made from an ultra-bright light-emitting diode (LED), a switch, a battery, and some odd valued resistors in order to test the student's knowledge of creating a resistor network with the right Thevenin resistance. The product of this simple project is a cheap flashlight that students can take home. A small competition can even be held to see which student can achieve the lowest cost with the brightest flashlight that lasts the longest, given that the students are to purchase each of the odd valued resistors for a certain price.

By *giving theoretical concepts a practical home*, student intuition is also improved tremendously. Practical demonstrations give students a better insight and "feel" into engineering concepts. As a follow-up from the previous example, students who proceed on to learn about capacitors can modify their flashlight resistor-only circuit to include capacitors, more resistors, and an extra LED. From this, a blinker circuit can be constructed to alternately blink 2 LEDs one at a time. This can be useful to demonstrate the concept of a time constant by having students tweak RC values to affect the rate of the blinker.

Another learning strategy used is *just-in-time learning* [2, 8]. Students are led through a problem to a place where no obvious solution can be obtained with their present level of knowledge. At that point, a new solution method or approach is introduced. This gives students new ways to overcome the problem.

Some of the old methods of teaching may lack appeal to the younger generation of upcoming students that were raised in the age of media globalization and the Internet. The old tried-and-true ways of teaching engineering subjects have encouraged a staunch tradition to continue teaching new students using methods that have been around for a long time. This presents challenges in attracting and retaining engineering students who often lose interest in engineering because of the slow build-up to the upper-level courses where they finally learn and apply discipline-specific knowledge [3].

One of the main reasons for the lack of interest is that these new students that are introduced to engineering concepts are given very little guidance in terms of the direction they are heading in with the knowledge they are attempting to learn. The build-up of knowledge in lower-level courses offers no end in sight. Many do not persevere to the upper-level courses when "things start to get interesting". This is because topics are usually taught in isolation, perhaps for the unspoken reason to ease the topic separation process for the sake of exam coverage.

Topics taught in isolation form islands of knowledge in a student's understanding that are not very useful on their own and degrade quickly over time. The TekBots program enhances learning by *linking islands of knowledge* together through the use of a learning platform. This linking occurs within a class and also between classes in a curriculum.

(b) Produce work-ready graduates

When students are immersed in fun *real-world projects* complete with real problems, they end up gaining useful practical experience. There are a host of skills learned in the context of a real project that are directly applicable in the real world but are not taught in traditional curriculums. For example, a real-world project can teach students how to troubleshoot a malfunctioning system, how to choose the correct capacitor for a circuit, or how to setup an oscilloscope in delayed sweep mode. These skills set them apart from students who come from a traditional lecture-homework-test-based education system.

TekBots program projects make use of *contemporary tools and practices* that mimic established design flow processes often found in the engineering industry. Crippled simulators and stripped-down tools are not part of the curriculum. An atmosphere of the real engineering world is kept intact by having students use the tools they will use after leaving the university.

When working in a real-world setting, students are exposed to interaction within a *community of learners*. The mutual benefits that are generated within the community foster a warm and welcoming environment where students can learn and experiment at their own pace, and be encouraged along the way. The community of learners also serves an indirect purpose of teaching students the life-long communication and cooperation skills. Studies have shown that cooperative learning tends to stimulate academic diversity and personal growth [9]. This would result in work-ready graduates who are more likely to succeed because of social interaction within the community of learners as well as the community at the work place. In contrast, graduates who are unprepared often find themselves rushing to acquire skills in the competitive work-place environment without realizing how vital it is to have community support.

It is important to point out that not all graduates go out to find jobs. Some may choose to remain in academia. Nonetheless, the experience that the TekBots program offers still applies in producing graduates who are ready to join and support the academic research community.

(c) Create innovative students

The TekBots program sees innovation as something that needs to be nurtured steadily. It is a worthy investment because even though innovation takes time to build, it is a trait that is hard to relinquish once the student has earned it.

Current teaching methods that churn students through the cycle of lectures, assignments and exams only measure the retention level of students for the knowledge they have acquired throughout the duration of the class. These teaching methods deal mostly with the quantitative aspects of education (i.e., assignment and test scores) in the hopes that the qualitative aspects will improve. This passive education approach of ignoring the qualitative aspects, such as innovation, creativity, craftiness, maturity, and professionalism, creates an environment that is detrimental to these intangible traits.

When it comes to students flexing their innovative skills at problem-solving, the exercises offered on paper are often found to be boring, contrived, and sanitized with no loose ends or real constraints, and typically have only one correct solution [2]. If students find that they keep arriving at the same solutions as their peers for the problems presented in class, they will be discouraged from looking for other better solutions and begin to take the problem-solving process for granted.

Instead, students should be allowed the freedom of *multiple solution paths* to a given open-ended problem, much like how it would be in a real-world situation. This vital factor does not stifle growth and would encourage the students' capacity for innovation.

To nurture innovation, first, students are *inspired at the freshman level*. They are introduced to great work that has been done in the past by engineers, and shown what heights can be achieved by good innovation. From the beginning, students should be provided with easy avenues accessible for academic development. New students need to be shown creative possibilities in a direct manner to give them that extra push to explore an innovative solution. This is different from spoon-feeding students into performing the same solution task repeatedly term after term.

Innovation is *fueled by the flexibility* of the learning platform. For this reason, a TekBot is modular and is composed of replaceable modules. These modules function a lot like LEGO [10] building blocks that could be put together. Parts of the robot circuitry can be customized, within reasonable constraints, by students any way they wish. In fact, similar work has been done before by Jadud with LEGOBots at Indiana University [11]. He has proposed TeamStorms as a way to teach robotics by making use of the LEGO Mindstorms Robotics Invention System [12]. According to Jadud, the problem-solving process should allow room for creativity, and be fun, open-ended and challenging at the same time.

2 PLATFORM FOR LEARNING

2.1 The Platform for Learning^{TM} concept

The Platform for Learning^{TM} concept described in the previous chapter defines how the TekBots program is executed. It is an educational strategy that strives to bring together other effective and successful educational strategies into one combined package. It anchors its own educational strategy plan in an omnipresent learning platform to bring out the best practices in a dynamic learning environment. The general underlying idea of having a student apply what she or he has learned in class to a platform, and then having that platform relate back to the student as they both evolve, is key to the TekBots program at OSU. An electronic robot acts as the learning platform for the ECE program at OSU, and the TekBots program acts as the vehicle that carries out the educational strategy outlined by the Platform for Learning^{TM} concept.

Educational Strategy	Description	Impact
Active/cooperative learning	Instructional activities engage students in doing and thinking instead of passive listening.	Improved retention. Higher academic achievement. Improved individual accountability. Improved small-group skills. Enhanced creative thinking.
Technology enhancement	Computing resources introduced into classroom to enhance learning by using software tools.	Increased comfort level using computers as tools. Mundane tasks reduced to allow focus on higher-order thinking.
Just-in-time learning	Theoretical concepts introduced when students' experiences create a demand for them.	Improved academic performance. Life-long learning skill development. Theory and practice kept in context.
Curriculum integration	Learning activities restructured to build contextual connections between topics.	Enhanced ability to transfer knowledge to new situations. Better program retention because of material relevance. Better recall of material.

Table 1: Educational strategies that enhance engineering education [2]

By definition, a platform for learning is a *common unifying object or experience* that weaves together the various classes in a curriculum [2]. It is a concrete yet dynamic system that is built upon as students progress through the curriculum. This helps students to better grasp the connection between concepts presented in a variety of classes and gain a much richer understanding of a discipline as a whole [7].

The Platform for Learning^{TM} concept also applies effectively to other fields of study besides engineering. For other fields of study, a learning platform can be something that is non-physical. For example, a business student may use a business plan as a learning platform. That business plan would be modified and adapted accordingly to the syllabus that is currently being taught. A computer science student, on the other hand, may start with a basic shell program that would eventually be built into a full-fledged OS (Operating System).

The ideals of the Platform for Learning^{TM} model advocate a fresh perspective in education organization. This is to incorporate the learning platform into the entire curriculum in order to energize and consolidate the engineering topics that are taught in class and in laboratory work. By using a common learning platform throughout a degree program, the integration of knowledge is enhanced. The platform also provides the conceptual "glue" between lecture topics [2].

In a practical sense, the syllabi for the entire curriculum are tweaked slightly in such a way as to arrange the engineering topics to provide a coherent flow between the core courses and the learning platform activities. When topics are introduced to students in an incremental logical manner, the students are able to get more out of the entire experience. When that experience is coupled with a valuable platform that is used to experiment and demonstrate the concepts that are learned in class, this will solidify the students' understanding and increase retention of knowledge.

As mentioned in the previous chapter, the TekBots program capitalizes on the Platform for Learning^{TM} concept by introducing a robot as the learning platform that ties experiences together from various different areas in engineering such as electronics, mechanics, communications, systems programming, and testing. The robot motivates lecture topics and meshes them with laboratory experiences. This expands the learning opportunities and effectiveness in multiple dimensions by providing a context for learning that connects the knowledge among classes, develops innovative abilities, and enhances troubleshooting skills [2].

One goal is to have students realize that even though there exists a hierarchy of knowledge which stems from basic fundamentals, the application of engineering knowledge is inevitably interrelated and interwoven with other disciplines of knowledge. The transition of engineering education to embrace a *transdisciplinary* model that integrates the use of the tools, techniques, and methods from various disciplines is vital for its future welfare.

At the moment, educational programs face many difficulties because of the rapid change of technology in today's environment [13]. According to Ertas et. al., the *transdisciplinary* model does not mean that the traditional engineering disciplines must be completely disassembled. It does mean, however, that the areas of knowledge typically included in each of the disciplines will be presented within the transdisciplinary structure, and that the boundaries between the knowledge areas will be much more porous.

While there is a need to push forward in the multi-discipline direction, studies have shown other directions of adapting learning effectiveness that we also need to centralize our efforts on. For instance, one area of study focuses on a *learner-centered* approach to education which states that knowledge must be actively constructed by learners and not passively transmitted by teachers [14]. Catalano et. al. proposed using visual tools in a learner-centered environment to assist students in "seeing" how information can be connected, and to teach them to use these tools independently themselves [15]. This was accomplished in a group-learning setting that provided a non-threatening "no risk" mechanism for indirect dialogue between teachers and students.

There has also been a lot of research in the area of *cooperative learning*. Clark found that when a multimedia group project format was evaluated against the traditional lecture/homework format, student satisfaction increased while student learning either remained constant or may have increased slightly [16]. Even though Clark's study seems to suggest that the effect of cooperation on student learning may be less than significant, it should be noted that the improvement in student satisfaction would highly be in favor of motivating students who find certain subjects rather technical, dry, and boring [17].

What we notice from most of these engineering education studies is that the different adaptations of engineering education seem to converge at a proactive and synergistic effort to enrich learning. This convergence point is what the Platform for Learning^{TM} is attempting to characterize. So far, we realize that we have to merge both traditional and modern education methods in order to produce a more wholesome and beneficial education experience. The Platform for Learning^{TM} model puts forth the notion of a learning platform to deal with that merger.

2.2 Learning platforms

A learning platform is an object or experience, based on the Platform for Learning^{TM} model, that is introduced into an entire curriculum to unify the various classes and knowledge of a discipline. It gives practical application to the knowledge that students acquire in-class to consolidate concepts in the students' understanding.

What is different about the learning platform as compared to a standard laboratory exercise is that it assimilates and executes effective educational strategies to provide students with concrete experience. These strategies were derived from numerous research studies on academic education [8, 9, 11, 13-18] in addition to the discoveries made by OSU faculty while working on the TekBots program [2, 3, 7].

In section 2.2.1, we shall provide the general context of how a learning platform integrates the current educational strategies that have been presented in the previous section 2.1. Then, in section 2.2.2, we shall introduce the attributes of effective learning platforms. Through these attributes, the Platform for Learning^{TM} concept defines its ideals to teach students effectively in a dynamic learning environment.

2.2.1 Integration of educational strategies

Conceptually, the learning platform itself is *learner-centered* in the general context of its use. It has to be able to get students actively involved in knowledge construction. The main idea is to allow students to be in control of their lessons. Concepts that need to be taught should never be forced upon students. Instead, the concepts should be presented in small digestible amounts that would allow students to attain the next-in-line design goal of the platform's current project. In other words, information should be provided to students just-in-time for their needs and does not need to be complete. This will give students enough of an incentive to make a continual effort of applying concepts to the learning platform to reinforce what they have learned.

As the construction of knowledge unfolds as a result of the students' own efforts, the experience they receive will increase their retention of knowledge and further engage them in the learning activity. To get students actively involved in knowledge construction, learning activities should focus around a set of intrinsically motivating problems that are situated in real-world tasks [14].

The platform itself is taught in a *cooperative learning* environment that employs a *proactive* learning style. Learning should take place in a collaborative environment that involves social interaction and negotiation [14]. This would help create the atmosphere of a large engineering team where students can engage in formulating and evaluating problems, conjectures, arguments, and explanations, just as professional engineers do in the workplace [2].

Finally, the platform takes on a *transdisciplinary* teaching approach. For instance, the project assignment structure that is presented in each class would encourage students to ask questions, to cross disciplines in seeking information, and to be creative in defining a problem and developing solution alternatives [18]. In addition, concepts and knowledge from traditionally non-engineering areas, such as business, economics, human relations, etc., will be included in the learning mix much more naturally. Thus, the engineers produced by the transdisciplinary educational process will be well-rounded and capable of dealing with complex problems which involve many issues that span the educational spectrum [13].

2.2.2 Attributes of effective learning platforms

Listed below are the attributes which define an effective learning platform that is conducive towards enhancing learning.

(1) Inspires exploration and innovation

An effective learning platform is able to inspire students to explore different design decisions when solving an engineering problem. This can be achieved by providing students with a strong background and understanding of the fundamentals required to analyze and construct a solution method. The platform should then stimulate students into thinking outside the box and beyond the given problem at hand.

The platform is also able to inspire innovation. The learning platform encourages students to begin the journey of exploration on their own. For example, early projects of the platform provide small but exciting hurdles of problems for new students to accomplish in order to engage the students and boost their confidence.

(2) Puts theory into practice

The learning platform represents a practical implementation of theoretical knowledge. It should challenge students to tasks that involve practical usage of the knowledge they have learned so far in the curriculum.

As students continue to work on the learning platform, they can begin to appreciate how the knowledge components learned in class can come together to help them achieve goals of their project. This subtly strings together the class syllabus into an interconnected web of knowledge that is much more useful than the knowledge derived from fragmented topics taught in isolation. According to Hadjerrouit, the process of constructing interrelated knowledge requires higher order thinking skills, such as analysis and design skills [14]. With the help of an effective learning platform, students are able to cultivate those skills.

(3) Keeps focus on core topics of a class

The learning platform needs to be faithful to the course syllabus and not stray from what is intended to be taught. When implementing a platform for learning, we also need to reduce the amount of redundant work that students have to do. Redundant activities are those that take away the focus from the core topics of the current class. For example, too much soldering work can really dampen the objectives of an electronics project. Instead, a prefabricated circuit board can ease the work of the students and free them to concentrate on higher level tasks such as design and testing.

(4) Deepens understanding

When students face minor problems while working on a learning platform project, they are required to study the subject in more detail in order to rectify those problems. The platform acts as a feedback mechanism that informs the student just how much understanding is required for the project. This will deepen a student's understanding of the discipline.

Students begin to appreciate the deviation of practical implementations from ideal theory when they perceive results based on first-hand experience. Paper assignments rarely encourage students to think about real-world improvisations and their implications. With a learning platform, students have an authoritative practical reference that supplements other references such as textbooks, lectures, and online material.

(5) Reusable in other classes to leverage scaffolding of knowledge

The continuous development of the learning platform in future classes encourages students to find solutions to a given problem using a new set of tools and knowledge that has bearings on past knowledge. Experience and knowledge gained from one project can be reused to form a knowledge scaffold from topic to topic or from class to class. Once a strong base is built, incorporating new knowledge onto an existing scaffold is easier.

(6) Flexible, expandable, and does not limit extensibility

It is absolutely essential for any learning platform to be flexible in terms of use and expandable in terms of functionality. These two criterions provide avenues for innovation and variations of design solutions that cannot otherwise be achieved by a strictly specialized platform or exercise.

With overly specialized platforms, students are limited in terms of the number of goals that can be achieved. Specialized hardware that only serves one purpose is an educational dead end. For example, a microcontroller circuit board that has every imaginable support part wired to the microcontroller itself leaves very few input/output ports for expansion by students. This will limit the extensibility of the learning platform by fixing the number of solution sets that is determined by the hardware. On the other hand, a more flexible platform can incorporate concept applications from future classes as it is reused to leverage scaffolding of knowledge.

A learning platform should be treated as an ongoing system that gives students the freedom to breed new ideas. It should be composed of configurable parts that students can fully customize. The building blocks of the platform must be understood by the students and be easy to form into something useful [2]. A sufficiently flexible platform would allow multiple solution paths to a given open-ended problem. The vast freedom of solutions closely resembles the real-world environment and encourages student creativity.

(7) Hands-on wherever possible

Students can gain a better "feel" of acquired knowledge when they apply in-class concepts to the platform using a hands-on learning style as a way of involving more human senses. From a psychological perspective, students have a higher likelihood of absorbing knowledge when more human senses are involved in the learning process. For example, a powerful way to teach new students about signal frequency is to show them what the signal looks like when measured by an oscilloscope, and what the signal sounds like when it is fed into a speaker. This provides visual and auditory stimuli that will increase retention of knowledge.

When students engage in hands-on work, the amount of material retained and the ability to integrate that knowledge is greatly improved when the course material relates to personal experience [2]. Studies have shown that students with practical experience scored better in the final exam than those without practical experience [17].

(8) Fun to use

An effective learning platform is fun to use. Fun instills in students the motivation to learn and explore, as opposed to making them feel as though they are relearning the wheel. The platform needs to be structured so that it makes learning the basics fun, exciting and engaging.

(9) Portable

A learning platform needs to be compact and lightweight. It should be portable so that students can quickly access the platform conveniently when a design inspiration strikes. Students can also bring it around to impress friends, demonstrate to family, and prove to future employers of their engineering skills.

(10) Inexpensive to purchase

Affordability is necessary for the purpose of education. When a learning platform is sufficiently low in price, every student is able to afford one. This encourages personal ownership of the learning platform and avoids congested sharing of a common platform.

If the platform is physical in nature, the parts that build the platform need to be easily reproduced for production and restoration purposes. Also, cheap replacement parts ease the burden on students who accidentally destroy them.

(11) Useful in a community of learners

A platform for learning enriches the learning experience for the entire *community of learners* that include students, parents, teachers, schoolchildren, enthusiasts and employers alike. Members of the community learn from each other, derive pleasure from the work being done, and give support back in return.

Learning platforms show inherent qualities in students who build them. These qualities capture the attention of other members in the community who, in turn, act as catalysts for the student's learning experience. This would reduce the burden on teachers by relinquishing the cultivation of student-learning morale to the community. Others can then help teach and motivate the students in their own ways. For instance, students who perform well would elicit praises from amazed parents. This will boost confidence and motivation.

The platforms also remain useful long after the lessons are over. They can be used to show creation and achievement. For example, employers tend to look for students who are passionate about interesting projects they have worked on in the past. To those employers, a learning platform that has accompanied a student throughout the entire curriculum represents many qualities, among which are the student's tenacity and dedication to an ongoing project. These are highly prized qualities valued in the job world.

3 FPGA LEARNING PLATFORM

3.1 Introduction of a new FPGA-based learning platform

An FPGA (Field Programmable Gate Array)-based learning platform that encompasses the attributes described in Chapter 2.2.2 is being introduced to undergraduate students. This platform is intended for an upper-level SoC (System-on-Chip) design course. The objectives of this course are to teach students about the organization of VLSI (Very Large-Scale Integration) systems, and to bring them through the design flow of building a chip. This exposes students to the different types of constraints involved, such as area and timing issues, which they need to consider. The FPGA provides programmable logic hardware for the learning platform to facilitate the course's objectives.

Currently, the teaching method used for this course is to have students develop virtual prototypes of their VLSI systems. Unfortunately, virtual prototypes only provide students with a feel of their designs from the perspective of the CAD (Computer-Aided Design) tools they use. There is no hardware realization to give students a practical use for their work of designing, simulating, and synthesizing the entire system.

The new platform can provide practical experience by allowing each student to program an FPGA circuit board with her or his design. Students can then see their work in action as the designs perform actual tasks while operating on the FPGA learning platform.

3.2 Features of FPGA learning platform

The FPGA learning platform is an electronic circuit board that is centered around an FPGA chip that utilizes a set of software tools. The hardware is designed to give students as much freedom of expansion as possible while providing some common components on the main board for students to quickly get started. The philosophy of this approach is to provide a bare bones system that students can fully customize as they see fit, and to avoid giving students every bell and whistle. This adheres to the ideals for effective learning platforms outlined by the Platform for Learning^{TM} concept.

If students require more functionality from the FPGA board, they can make use of a daughterboard that plugs on top of the main board. The daughterboard is a bare prototype PCB (Printed Circuit Board) that connects to the user I/O of the main board. Once connected, added components on the daughterboard essentially extend the circuitry of the main board. This motivates students to innovate their own designs on the daughterboard while driving them to do the significant work of adding components.

Xilinx®, the company that produces the FPGA, provides a free set of industry-grade software tools to be used with the FPGA. The software tools are bundled as a graphical IDE (Integrated Development Environment) known as Xilinx ISE^{TM}. The main tool of ISE is the Project Navigator. It houses all the other tools, and is used to structure the hierarchy of an FPGA design. The navigator also allows design constraints to be set. From the navigator, the PACE tool can be loaded to assign package pins that connect core I/O to external I/O. Then, the design is synthesized using the XST tool. Finally, the FPGA can be configured by programming the synthesized design with the iMPACT tool.

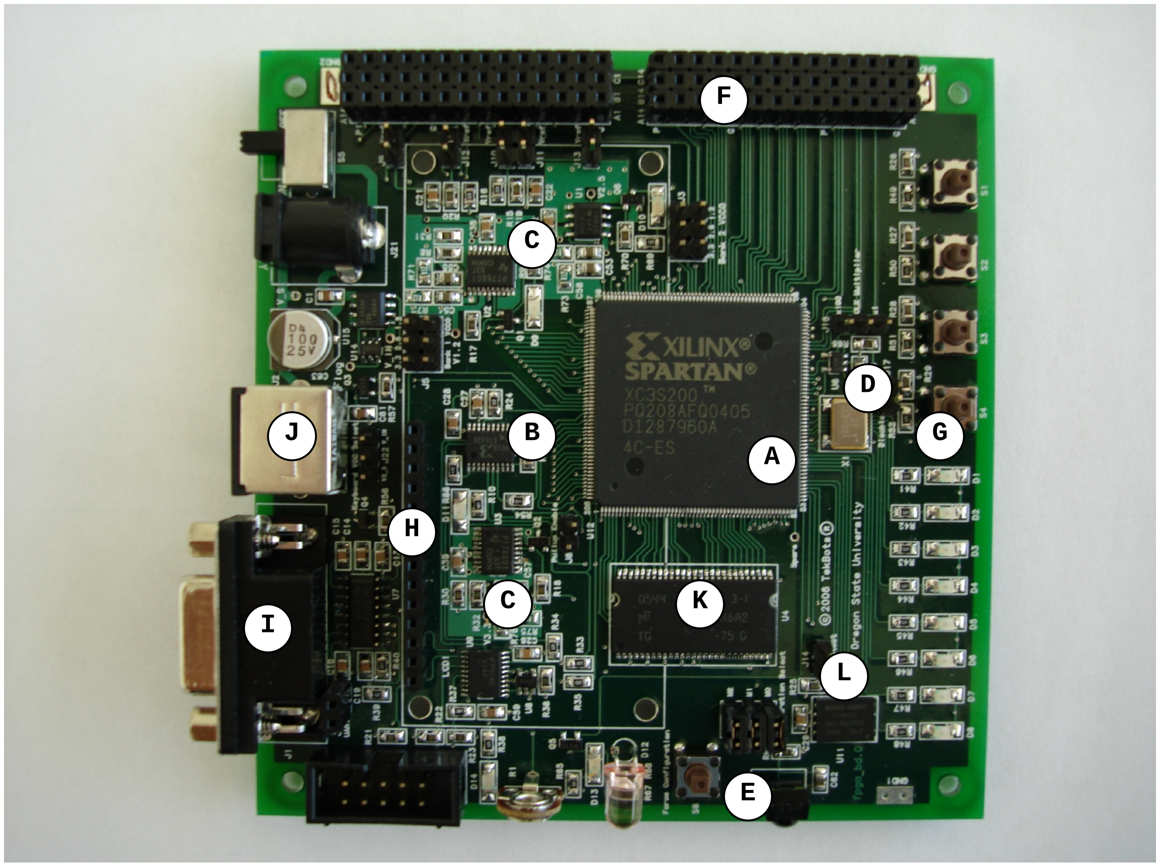


Fig. 1: FPGA learning platform board

Listed below are the hardware features of the FPGA board:

(A) FPGA as main processor

The FPGA chip used is the Xilinx Spartan-3 XC3S200 [19]. This chip provides a large 200K-gate design capacity, and is optimized for Xilinx RISC (Reduced Instruction Set Computer)-based cores such as the PicoBlaze microcontroller [20] and the MicroBlaze microprocessor [21] cores. Programming is achieved by downloading the configuration bitstream into the FPGA using JTAG (Joint Test Action Group) boundary scan.

(B) Flash PROM (Programmable Read-Only Memory)

A Xilinx XCF01S [22] flash PROM with a 1Mbit density is provided as a means to store a non-volatile configuration bitstream for the FPGA. Upon power up, the PROM will automatically program the FPGA with the bitstream. It will also appear as part of the JTAG scan chain for the FPGA board.

(C) Power regulators

The board supports 4 different voltage regulators with ratings of 1.2V, 2.5V, 3.3V, and 5.0V [23]. They supply power to the main board components and daughterboard expansion.

(D) Oscillators

A crystal oscillator drives the main clock signal for the FPGA chip that runs at 100MHz. Another oscillator [24] is provided for custom use. The output signal of the oscillator can be controlled by a jumper setting to produce 3 different frequencies from 607kHz to 6.07MHz.

(E) IR (InfraRed) transceiver

The board is equipped with an IR transmitter and an IR receiver. The IR transmitter is an IR LED pointing off the side of the board. The IR receiver is a miniature Vishay TSOP32138 [25] that is located on the same side of the board. The receiver is receptive towards IR signals with a carrier frequency of 38kHz. This frequency can be generated by dividing a 607kHz frequency (generated by the auxiliary oscillator) by 16.

(F) User I/O

The board has 84 user I/O pins located on the side. 64 are allocated for signal I/O, 12 are ground and power connections, and 4 are unconnected.

(G) LEDs (Light-Emitting Diodes) and pushbuttons

There are 8 LEDs and 4 pushbuttons on the board.

(H) LCD (Liquid Crystal Display) interface

The board provides 14 signal lines to interface with a 16x2 LCD that is provided with the learning platform. The interface could be extended to other types of LCDs with the common Hitachi HD44780 instruction set [26]. 10 of the 14 signal lines can also be used as general I/O.

(I) Serial port

A serial port allows serial communication using the RS232 protocol. The most common use of this type of communication is to send and receive data from a computer.

(J) PS/2 port

A PS/2 port is provided to enable interfacing with a keyboard, mouse, or any compliant devices. It supports both 3.3V and 5.0V operation.

(K) SDRAM (Synchronous Dynamic Random Access Memory)

The SDRAM chip on board is a single Micron MT48LC4M16A2 [27] with a capacity of 64M bits. This type of memory is volatile.

(L) EEPROM (Electrically Erasable Programmable Read-Only Memory)

The EEPROM chip on board is an Atmel AT45DB161B [28] with a capacity of 16M bits. This type of memory is non-volatile. Data is sent and retrieved from the EEPROM via SPI (Serial Peripheral Interface).

3.3 Advantages of FPGA learning platform

The advantages of using this FPGA platform corresponds to the attributes of effective learning platforms described in Chapter 2.2.2. Listed below are the advantages of using this platform:

(1) Can be used to teach a variety of topics

The FPGA board gives students great flexibility for systems design. It can be used to teach a wide variety of engineering topics. Some examples are provided below:

- Basic digital hardware
  - Combinatorial logic
  - Sequential logic
    - Mealy and Moore state machines
  - Logic fault detection
    - Stuck-at fault
    - Functional fault
- Core integration techniques
  - Microcontroller design
    - 3-stage pipeline implementation
    - PicoBlaze soft core with internal memory
  - Microprocessor design
    - 5-stage pipeline implementation
    - Superscalar implementation
    - Data bus construction
    - MicroBlaze soft core running uCLinux OS
- Mixed signal processing
  - DSP (Digital Signal Processing)
    - Digital multiplier for image processing
    - FIR (Finite Impulse Response) filtering
  - Analog / digital interaction
    - Signal generation and integration
    - Phase-locked loop

(2) <u>Reusable to leverage scaffolding of knowledge</u>

The FPGA board is great for linking together knowledge from previous classes. For example, the knowledge of how to use a microcontroller learned in a previous class can be reused to implement a soft core into the FPGA device. This allows the FPGA learning platform to build nicely upon the existing body of knowledge.

(3) <u>Affordable to students</u>

The FPGA platform will be available as an inexpensive alternative to available FPGA boards. Boards sold by commercial manufacturers are usually expensive, and their design specifics are often proprietary. Also, they usually leave few FPGA I/O ports for expansion, and incorporate too many advanced features that serve applications too specific for educational purposes.

(4) <u>Speedy design and testing for a low cost</u>

In order to implement a hardware realization of an ASIC (Application-Specific Integrated Circuit) design, a fabrication process is needed to create the chip. This process requires a lot of time (i.e., approximately 6 months or more) and money (i.e., in the order of thousands of US dollars). The FPGA, on the other hand, provides a hardware programmable chip that has already been fabricated. The implementation of hardware on an FPGA equates to the programming of the chip. This allows fast development turnaround time in the order of seconds. In addition, the cost of an FPGA chip is relatively low (i.e., below USD$10 when purchased in quantities of 100).

(5) <u>Portable</u>

With FPGA learning platforms, students can quickly implement and test their designs without requiring specialized development equipment. The software that is used to program the FPGA is free. Thus, students can use their platforms anywhere at anytime. The software tools are also widely adopted in industry, allowing the platform to be utilized in the commercial work environment.

(6) Hands-on to demonstrate details of concepts

The type of hands-on work involved with the FPGA learning platform requires students to dive into the intricacies of low-level hardware design. As such, the complexity of the FPGA device makes it inappropriate for use at introductory level courses [29]. With the FPGA, students can observe how "software" interacts with "hardware" at the device level. In the process of learning, they would also be exposed to the use of CAD tools, and HDLs (Hardware Design Languages) employed in industry.

(7) Very extensible

The FPGA learning platform is also very powerful in terms of expandability. Modern FPGA devices provide for large design capacity that is well within the boundaries of educational use. Often, students assume they must work with only one processor core. With an FPGA, this restriction has been loosened to expose students to multi-core applications on a chip. The end result would be the creation of more elaborate embedded systems that extend the functionality of the learning platform.

4 FPGA BOARD TEST ENVIRONMENT

4.1 FPGA core in a self-testable system

The flexibility of the FPGA platform introduced in the previous chapter allows for a different approach to be taken for testing the new board. The main processor of the FPGA board is "brainless" to begin with. The brains of the FPGA can then be configured with an intelligent core to test the on-board peripherals in order to produce a ***self-testable*** system. The board is able to test itself without requiring an additional tester built onto it. The self-test is achieved by employing the FPGA, the central part of the entire system to be tested, as the core component that facilitates the testing of the rest of the system. This method of testing is an alternative for exhaustive test methods that flood the board with test signals using a bed-of-nails type of setup.

An FPGA is inherently different from a microprocessor when used as the core component for a ***self-testable*** system. A microprocessor has a permanent core that only provides functionality general enough to cover a set of basic operations. An FPGA, however, can have its core optimized for application-specific tests. It can be customized for accuracy to implement timing-critical tests because of its logic-level architecture. It also allows multi-core implementations to be embedded on-chip, reducing its dependency on off-chip systems. For instance, an SPI core can be implemented seamlessly into the FPGA for components that require the interface. Conventional ASICs do not allow such pliability.

The on-board flash PROM that holds the bitstream configuration for the FPGA chip is initially empty. This is where the implemented core that performs the tests of the FPGA board components is stored. Whenever the board is powered up, the PROM will program the FPGA chip with the stored bitstream. This effectively provides the illusion that the FPGA device was tailored with a permanent core for testing purposes, and that the FPGA has "brains" by default.

In the learning environment, the content of a configured PROM is not intrusive. It does not get in the way of the students' FPGA designs. When students wish to implement their designs, they can bypass the PROM in the scan chain to program the FPGA directly. However, this type of programming is volatile. After a power reset, the FPGA board is flashed anew with the test suite. Students are, of course, given the choice of making their designs permanent by overwriting the PROM.

4.2 Development test suite for FPGA board

A major part of the work involved in this thesis is the development of the FPGA core that is used to test the FPGA board. This specialized core is able to test a variety of on-board components as well as student component modules and circuit designs. It has two objectives, which are to

- facilitate the development of the FPGA board while it is being manufactured.
- assist students with testing their FPGA designs in the learning environment using a cycle-accurate hardware testbench debugger (see Chapter 5).

The core provides a composition of tools to perform these objectives seamlessly. These tools are collectively known as the "development test suite". In particular, the development test suite for the FPGA learning platform is given the name "OMICRON".

The primary emphasis of the manufacturing tests conducted by OMICRON is mainly to verify solder connections of board components and traces that connect them together. As it is, the FPGA board has very few macro components. Most of the components connected to the FPGA are either interface ports or the user I/O. Available macro components such as the IR transceiver, SDRAM and EEPROM are tested for their basic functionalities.

The conducted tests are simple and quick. There is no need for elaborate tests that have results that are hard to decipher. We only need to know whether or not a problem exists with one of the board components. After that, further work to troubleshoot the exact problem is relatively straightforward.

OMICRON assumes a fully functional FPGA chip (and flash PROM) whereby each of the components connected to the FPGA is systematically tested. This is a reasonable assumption made in order to manage and contain the complexity of the test suite. Assuming otherwise would entail testing the core exhaustively, which is beyond the scope of objectives for this test suite.

In section 4.2.1, we shall explain the structure of OMICRON and how it is built internally to provide the test suite. Next, in section 4.2.2, we shall describe how OMICRON performs tests on the various peripherals on-board. Then, in section 4.2.3, the two different modes of operation for OMICRON are presented.

4.2.1 OMICRON development test suite implementation

The development test suite core, OMICRON, is essentially a large state machine implemented in the FPGA. It is implemented around the popular Xilinx PicoBlaze microcontroller core [20]. The hardware of OMICRON running within the FPGA is composed of the PicoBlaze core, instruction and data-type ROMs, output registers, an input MUX (multiplexer), and supplemental core modules. The software of OMICRON manages the hardware by supplying PicoBlaze with instructions to control the entire state machine. OMICRON is written entirely in VHDL (for the hardware) and PicoBlaze assembly (for the software).

Shown below is a block diagram of OMICRON's datapath layout:

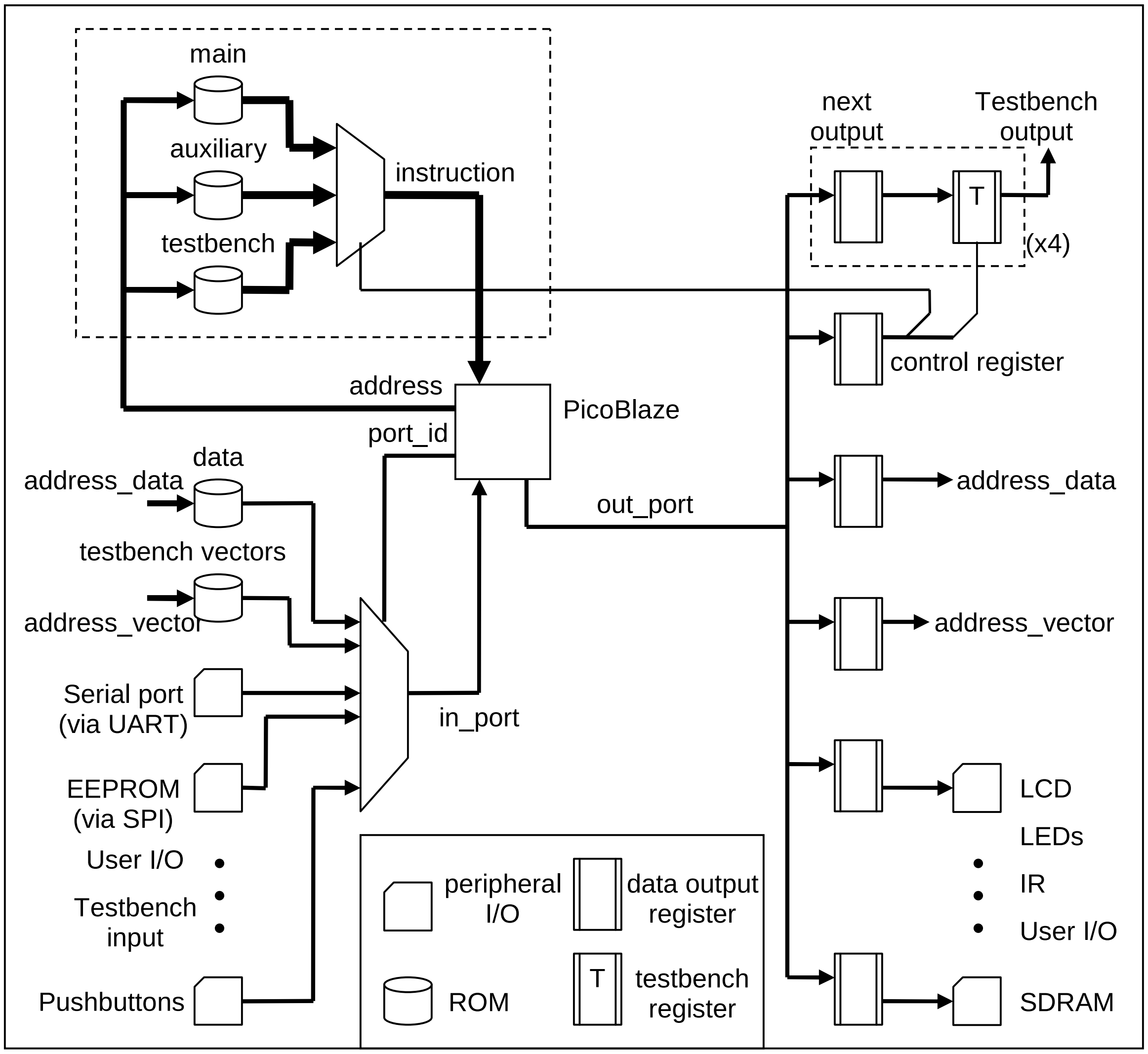


Fig. 2: Structure of OMICRON

Listed below are detailed descriptions of each part of the structure:

(1) PicoBlaze microcontroller core

PicoBlaze is an 8-bit microcontroller intended for use in Spartan-3 FPGA devices. It was chosen for its simple architecture which makes it a good candidate for applications requiring a complex, but non-time critical, state machine. The core is totally embedded within the target FPGA and requires no external support. It is supplied as a synthesizable VHDL macro that is handled by place-and-route tools to merge with the logic of a design.

The PicoBlaze core is optimized for low deployment cost and efficiency on Xilinx FPGAs. It only occupies 5% of the XC3S200 device capacity. Its small size allows other large designs to co-exist alongside it, which is ideal for the purpose of a testbench (more in Chapter 5). The performance of PicoBlaze is respectable at approximately 43 to 66 MIPS (Millions of Instructions Per Second) even with such size constraints [30].

All instructions under all conditions will execute in 2 clock cycles. The constant execution rate is of great value when determining the execution time of a program [30]. There are 57 types of instructions in the instruction set. CALL and RETURN instructions support a stack depth of 31 levels.

Internally, the PicoBlaze microcontroller provides 16 x 8-bit registers with 64 bytes of scratch-pad memory. It also provides abundant flexible I/O. From this, its basic functionality is easily extended and enhanced by connecting additional logic to the microcontroller's input and output ports.

Shown below is a diagram of PicoBlaze and its interface I/O:

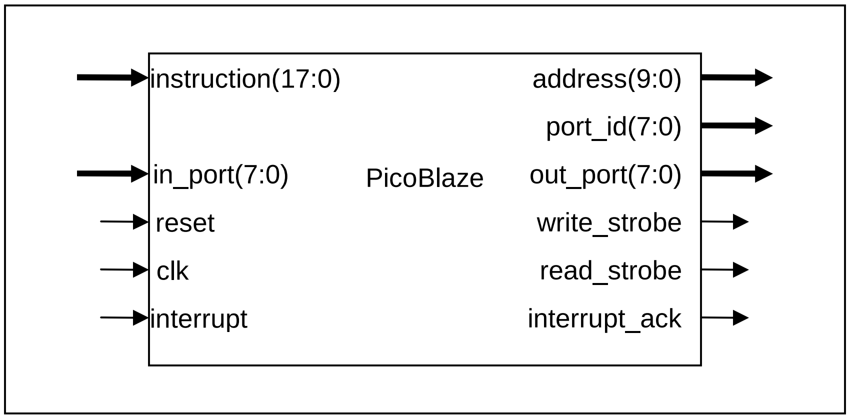


Fig. 3: PicoBlaze microcontroller core

As seen from the figure above, each PicoBlaze instruction is 18 bits long, and is addressed by 10 bits for a maximum of 1024 instructions in a single 18kbit ROM. Its input and output ports are 8 bits each, and are addressed by an 8-bit port_id to allow 256 locations of inputs and outputs. Read and write strobes are provided to inform an external module when a read or write operation is performed. A level-triggered interrupt pin is also provided.

(2) Microcontroller I/O

Due to the addressable nature of the I/O, special care is taken to create output registers that only capture PicoBlaze's output based on the port_id address. Shown below is the construction of a data output register:

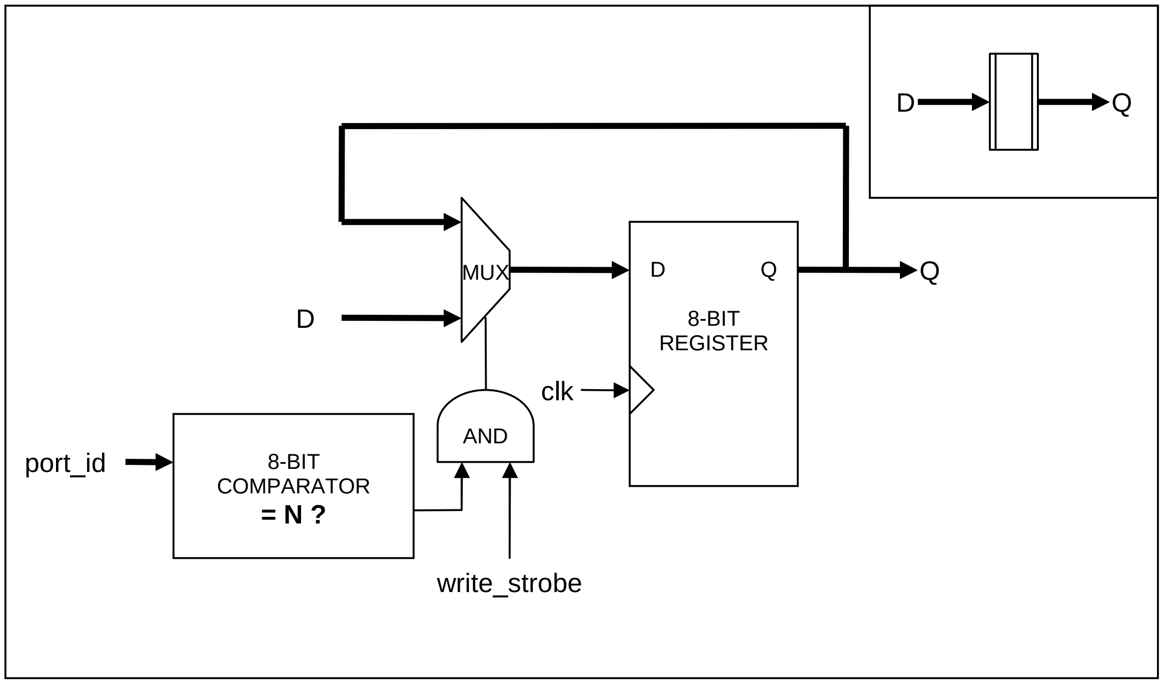


Fig. 4: OMICRON data output register

In a data output register, the port_id signal is checked to determine when it is appropriate to capture the incoming byte. An extra precaution is taken by only allowing a byte to be captured when there is a write strobe indicating a write operation. This is analogous to the operation of a data bus. The data bus is indicated by the out_port signal in Figure 2.

On the input side, a multiplexer is responsible for selecting which input byte to read. The selection is determined by the port_id signal. With the exception of the UART (Universal Asynchronous Receiver-Transmitter) modules, the read strobe is not utilized by OMICRON. This is because most of the incoming bytes originate from non-queued sources.

In Figure 2, output registers are used for control signals of the datapath. It is also used to send data to various peripherals. **For the sake of keeping the figure concise, not all peripherals are listed.** Likewise, the list of connected peripherals on the input side is shortened as indicated by the triple dots.

(3) Instruction ROMs

An assembler is provided with the PicoBlaze source files to compile assembly programs into synthesizable VHDL ROM modules. Each ROM module is allows 1024 x 18-bit instructions to be stored.

Unfortunately, OMICRON requires more than 1024 assembly instructions for its operation. For this reason, two extra instruction ROMs were created to supplement the main program ROM. The entire OMICRON program is partitioned into three instruction ROMs as follows:

§ Main program ROM

  The main ROM contains the root program execution for OMICRON. This includes code for the menu interface as well as the handling of RS232 terminal and non-terminal modes. This also includes a majority of the code for testing on-board devices (known as "devtests" or device tests). Devtests for the LEDs, pushbuttons, LCD, serial port, SDRAM, and EEPROM are found in this ROM.

§ Auxiliary program ROM

  The auxiliary ROM contains subroutines to construct and display strings on the RS232 terminal or the LCD. It also contains various subroutines for LCD operations such as displaying characters and clearing the display.

§ Testbench program ROM

The testbench ROM contains subroutines that define the operations of the cycle-accurate hardware testbench debugger. It also provides code space for custom subroutines that build upon OMICRON's existing subroutines using the PicoBlaze instruction set. Devtests for the user I/O, PS/2 port, and IR transreceiver are found in this ROM.

OMICRON extends the number of program instructions beyond 1024 by means of a soft switch [31] to switch seamlessly between the three instruction ROMs while the root program is still in execution. The outputs from the three ROMs are fed into a multiplexer. One of the outputs is then selected as the current instruction by a control signal originating from a data output register.

(4) Data ROM

OMICRON utilizes a single 2048-byte ROM as its data ROM. The data ROM is used to provide OMICRON with its vocabulary of words. The words were compressed by hand and stored as most-used syllables to save on storage space. This allows OMICRON to efficiently generate over 140 English words from only 256 bytes used. The remaining 1792 bytes are left for future use by students.

A Perl script was written to generate data ROM modules as synthesizable VHDL macros. The script `data2rom.pl` takes a text file as an input and parses its content by performing an ASCII (American Standard Code for Information Interchange) to hex conversion. The result is filled into the form file `dataROM_form.vhd` to generate the macro.

(5) Testbench vector ROMs (optional)

Optional testbench vector ROMs can be plugged into OMICRON as a source of test vectors for the hardware testbench debugger. A more detailed explanation is provided in Chapter 5.

(6) <u>UART modules</u>

UART transmit and receive modules provided with the PicoBlaze package [32] are used to facilitate RS232 communications for the serial port. They provide the functionality of a simple UART transmitter and receiver with these fixed characteristics:

- 1 start bit
- 8 data bits (least significant bit first)
- No parity
- 1 stop bit

This pair of macros are highly optimized for Xilinx FPGAs. Both the macros occupy approximately 2% of the XC3S200 device capacity.

(7) <u>SPI core</u>

An SPI core is required to provide the interface between OMICRON and the EEPROM. The core will operate as the master in SPI mode 3 whereby the data is ordered most significant bit first. The main clock of 100MHz is divided down by 8 to produce a 12.5MHz SPI clock that is well within the requirements of the EEPROM.

(8) <u>Clock divider (not shown in Figure 2)</u>

The signal from the auxiliary oscillator is fed through a clock divider to produce a 38kHz frequency signal. This is required as a carrier frequency for testing the IR transceiver. The 38kHz signal is achieved by dividing a 607kHz clock signal generated by the oscillator by 16.

(9) <u>Testbench output registers</u>

These registers facilitate the output functionality of the hardware debugger. A more detailed explanation is provided in Chapter 5.

4.2.2 How OMICRON performs tests on FPGA board

Theoretically, module testing involves the application of test vectors to a tested component in order to induce an observable result. That result is then used to assess whether or not the test was successful. In a self-testable system, the core (assumed to be fully functional) is responsible for testing the components around it. However, for that core to receive the results of its tests, some sort of result feedback mechanism must be employed.

In the case of the FPGA board, most of its components provide some means of feeding signals back into OMICRON. For instance, the user I/O pins can be connected in pairs to form feedback loops. On the other hand, the IR transceiver can be used to send and then receive an IR signal. Some components require human feedback for test confirmation. For example, a pushbutton test relies on the user to push the button and visually confirm an actuated result.

In the list below, we shall describe the methods used by OMICRON to test each peripheral of the FPGA board. The entire implementation that was laid out in the previous section 4.2.1 enables OMICRON to probe traces and to execute test algorithms on components that are external to the FPGA.

(1) Serial port test

The transmit and receive capabilities of the serial port is tested by means of a serial communication between OMICRON and a computer running an RS232-capable terminal.

To test the transmission of data, a simple ASCII string is sent to the terminal by OMICRON. The user would have to visually confirm that the string was successfully sent, indicating that the TX (transmit) trace is good.

To test the receiving of data, OMICRON will continously listen for any incoming serial transmission from the computer. In this test, anything that is typed on the keyboard will be sent via the terminal to OMICRON. Incoming bytes will be displayed on the LCD as they are received, indicating that the RX (receive) trace is good. During this time also, LEDs would alternately flash as an additional indicator, providing backup for a faulty LCD.

Of course, if commands can be issued and information can be received via the terminal in the first place, then the serial port is in full working order.

(2) <u>LCD test</u>

To test the LCD, a range of ASCII characters is displayed for every character of every line on the LCD. This will test the integrity of the LCD interface traces. Block characters can also be displayed to detect dead pixels on the LCD. Visual confirmation of the test is required by the user. At any time, the LCD can be cleared for a new test.

The serial port data receive test can also be used to test the LCD. Characters sent from the keyboard are displayed on the LCD, allowing the user to enter custom messages.

(3) <u>LEDs test</u>

A binary counter is activated on LEDs to flash them in an incremental manner. Visual confirmation is required by the user.

The serial port data receive test can also be used to test LEDs. The LEDs will toggle flash whenever a byte is received from the keyboard.

(4) <u>Pushbuttons test</u>

Pushbuttons will be tested for button push events. Whenever a push is made, a message to indicate which button was pushed will either appear on the terminal or the LCD, depending on the mode of operation (see next section 4.2.3). LEDs will also toggle accordingly. Visual confirmation is required by the user.

(5) IR transceiver test

The IR transceiver is tested by first transmitting a signal via the transmitter, and then receiving it with the receiver. This action forms a feedback mechanism between the two devices that allows OMICRON to test both devices simultaneously. OMICRON is able to determine the success of this test on its own. User feedback is not required.

A valid IR signal is distinguised from other signals, which exist naturally in the environment, by its carrier frequency. The 38kHz carrier signal provided by the clock divider (see previous section 4.2.1) is used to modulate a valid signal. This signal is continously pulsed in bursts of 16 periods with a 37.5% duty cycle. On the receiver side, the incoming signal is verified at critical points to ensure that the original signal is correctly received.

It is advisable to avoid testing the IR transceiver under direct exposure of fluorescent light that can drown the signal. It is also advisable to provide a reflective surface, such as a sheet of paper, to reflect the IR signal back to the receiver.

(6) User I/O test

The 64 general I/O pins available on-board are grouped into 2 virtual ports of 32 pins each. These virtual ports are physically connected together in pin-wise pairs to form feedback loops to the FPGA.

To test the user I/O, each virtual port takes its turn to become an input and an output. While one virtual port acts an input with pull-ups activated, the other port act as an output that drives the former input port low. When the entire input port is detected as being driven low, the test is successful for one way. Then, the virtual ports interchange their input/output roles and this process is repeated again. OMICRON reports that the whole test passes only when it is successful for both ways.

(7) PS/2 port test

This test is similar to the user I/O test except that it only involves 2 pins. First, the clock and data lines are physically connected together to form a feedback loop. Then, with pull-ups activated, each line takes its turn to become an input and an output. The current line acting as an output will try to drive the input line with a low signal, and vice versa. If low signals are detected both ways, the test is successful.

(8) SDRAM test

The process of testing the SDRAM involves writing some data to a memory address location, and then immediately reading the data back from the same address location. If the data sent and received do not match, OMICRON will report the failure by displaying data value and address location where it failed. This process is repeated until the end of every test stage.

There are two stages for the SDRAM test. The first stage is to test the data bus. This is achieved by first fixing the memory address location to 0. Then, the data lines are tested with walking-ones and walking-zeroes configurations. With these configurations, it is easy to point out which data bus lines are faulty once errors do occur.

```
00000001      11111110
00000010      11111101
00000100      11111011
00001000      11110111
00010000      11101111
00100000      11011111
01000000      10111111
10000000      01111111
```

Fig. 5: 8-bit walking-ones/zeroes example

The second stage is to test the address bus. For this stage, an exhaustive test is performed on every memory address location with pseudo-random data values. It is important to cover every address location to confirm the operation of all memory cells.

(9) <u>EEPROM test</u>

The testing process for the EEPROM is the same as for the SDRAM. It uses the same SDRAM first stage to test the data bus. However, it uses a different approach to test the address bus. Unlike the SDRAM, exhaustive testing for every address location is not recommended for the non-volatile EEPROM due to its limited number of allowable rewrites. Instead, walking-ones and walking-zeroes configurations are used in place of the SDRAM exhaustive approach to select the address locations. Those locations are then tested with pseudo-random data.

4.2.3 OMICRON modes of operation

Tests performed by OMICRON are manually executed by the user. Once a test is performed, OMICRON will return the results of the test to the user and await further instructions. OMICRON offers two modes of operation to interact with the user:

(1) <u>Terminal (TERM) mode</u>

For this mode, a serial cable connection between the FPGA board and a computer is required. Communication is established to OMICRON by opening a serial link using an RS232-capable terminal on the computer. The communication setting of the serial link is configured at a baud rate of 38400.

Once communication is established, the user is provided with a CLI (Command Line Interface) to issue commands to OMICRON and receive results on the terminal. On the CLI, a menu interface is provided to organize the various test commands. At any point, the user can view the current menu and its list of commands by issuing a question mark "?" as the command. To go back a menu level, a "G" command can be issued.

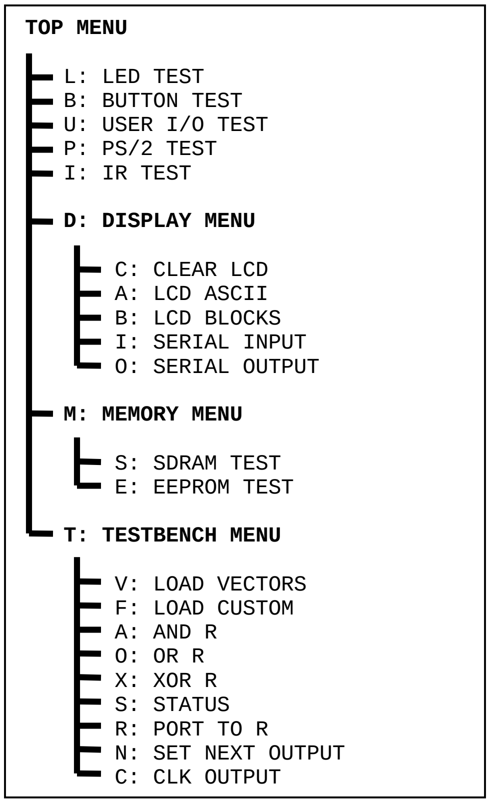


Fig. 6: OMICRON terminal menu hierarchy

With the exception of the TESTBENCH menu, all tests are executed with one-letter commands for simplicity. From the figure above, the alphabet indicated before each test is the CLI command for that particular test. For example, to test the SDRAM, a user currently in the TOP menu would perform this series of keystrokes:

(i) Type "M" and hit <Enter> to get into the MEMORY menu

(ii) Type "S" and hit <Enter> to start the SDRAM test

The TERM mode is the preferred mode of operation because OMICRON can be more verbose with the results it reports back to the user. However, if the serial port of the FPGA board fails for any reason, OMICRON cannot proceed with using serial communications for user interaction. Should this occur, OMICRON has the capability to fall back on a bare-bones mode for interaction.

(2) Non-terminal (NOTERM) mode

This is a stripped-down interactive mode where an LCD is used as the main display for test results and the pushbuttons are used to select test options. It should be noted that, even in this mode, OMICRON will still try to send verbose results via serial communication. Hence, NOTERM mode can in fact co-exist with TERM mode.

By default, OMICRON starts in TERM mode. In order to switch to NOTERM mode, the user just has to press the pushbuttons 3 times (any button will do). Once in NOTERM mode, the two pushbuttons closest to the LEDs are used to move backwards and forward through the available options that are presented on the LCD. The pushbutton that is furthest away from the LEDs is used to select or end the current option. Shown below are the options for NOTERM mode:

```
0000  ENTER TERM MODE
0001  LED TEST
0010  BUTTON TEST
0011  USER I/O TEST
0100  PS/2 TEST
0101  IR TEST
0110  LCD ASCII
0111  LCD BLOCKS
1000  SERIAL INPUT
1001  SERIAL OUTPUT
1010  SDRAM TEST
1011  EEPROM TEST
```

Fig. 7: OMICRON non-terminal options

In the figure above, the binary codes indicate how four LEDs are lighted for their respective options. For example, when the option "LCD BLOCKS" is selected, the 3 LEDs closest to the pushbuttons are turned on. This enables the LEDs to be used as a secondary display in the event that the LCD display is faulty. The other four LEDs are used to indicate results of the pushbutton test, and to also indicate test success or failure.

The "CLEAR LCD" command is not included in this mode because the LCD will be cleared automatically as it is used. In addition, the entire TESTBENCH menu is removed because NOTERM mode is too restrictive to support it.

4.3 Post-manufacturing requirements and test flow using OMICRON

Naturally, the development test suite is intended for users who require the FPGA board to be tested. These users fall either into the category of learners or manufacturers. Learners, such as students and educators, expand the use of the board in an educational context. Thus, they require an informative and detailed yet concise user guide that documents all the aspects of the learning platform explained in this thesis, including the development test suite itself. **An initial version of this user guide is provided in the Appendix.**

Manufacturers, on the other hand, are mainly interested in the peripheral testing aspect of the test suite. The information they require would be strictly on the test flow for the FPGA board. For this purpose, a step-by-step instructional guide is provided to facilitate the deployment of the development test suite in the manufacturing environment. This guide is briefly described at the end of this section.

Before manufacturing test work can begin, certain post-manufacturing requirements need to be fulfilled. The following are equipment requirements for a single test station:

- A computer equipped with both parallel and serial ports is required. On this computer, the Xilinx iMPACT tool and an RS232-capable terminal program are required to be installed. The parallel port is used to program new FPGA boards that come off the assembly line via a programming dongle. The serial port is used to access OMICRON in terminal mode using a serial cable.
- An LCD with at least 2 lines of 16 characters is needed for testing the LCD interface and for accessing OMICRON in non-terminal mode.
- A user I/O feedback connector is required to provide feedback connections for the user I/O test. The connector should plug and disconnect easily to speed up testing.

- A loopback wire is required to connect the data and clock lines of the PS/2 port for testing.
- A sheet of paper is used to reflect an IR signal back into the receiver for the IR transceiver test.

On the manufacturing floor, lightly skilled technicians are responsible for testing newly assembled boards. They are required to possess computer skills and a basic understanding of electronics.

The time allocated for running through the whole test suite to test a single FPGA board is approximated at 4 minutes. This includes the time to flash the PROM, assemble connectors and issue test commands as well as making quick troubleshooting decisions.

The technicians will be briefed on how to test new boards with OMICRON. Listed below is an introduction outline of the instructional guide on how to work the development test suite:

Before testing, make sure:

A1. Board power adaptor is connected to power source.

Serial cable is connected to serial port.

Programming dongle is connected to parallel port.

A2. Computer is booted up.

A3. Xilinx iMPACT tool is running with boundary scan chain detected.

To detect scan chain, perform steps A1 & A2 before clicking on "Next" twice in iMPACT. The same scan chain can be reused over and over again.

For each device on scan chain, select `omicron.mcs` as file to be programmed.

A4. Terminal program is running on serial port at 38400 baud rate, configured with 8 data bits, no parity, 1 stop bit.

When a new board is received:

B1. Set board down with pushbuttons pointing upwards from the table.

B2. Connect power adaptor to power input.

Connect serial cable to serial port.

Connect programming dongle to programming input.

Connect LCD.

Connect feedback connector to user I/O.

Connect loopback wire to PS/2 port.

Make sure jumpers are connected on PROM configuration pins.

B3. Power up board.

B4. Program PROM using iMPACT with `omicron.mcs` bitstream. The PROM is the first scan chain device.

Once the PROM has been successfully programmed, the technician can proceed to perform tests on the FPGA board using OMICRON. The results of each test is recorded and notes are made if necessary.

If the serial port test fails, the technician is instructed to switch OMICRON over to non-terminal mode. This allows the test flow to proceed without using the terminal program. If the LCD does not work in non-terminal mode, the technician is instructed to rely on LEDs as indicators. These alternative methods for using OMICRON effectively remove test flow hindrances even when the two primary displays are faulty.

5 OMICRON CYCLE-ACCURATE TESTBENCH DEBUGGER

5.1 Description of OMICRON testbench debugger

In the process of developing the test suite, OMICRON was expanded to include a *cycle-accurate hardware testbench debugger*. This debugger functions as a general-purpose digital hardware tester that can assist students in testing designs for the FPGA learning platform.

The testbench debugger is able to test modules implemented within the FPGA as well as circuits on a daughterboard connected to the FPGA. The flexibility of the FPGA helps to provide a conducive test environment where signals of a tested module can be easily accessed and probed by the tester.

Test results derived by the debugger are directly extracted from hardware. This makes the testbench debugger a valuable tool to supplement available software simulations in order to verify the functionalities of synthesized modules. The debugger can also provide test simulations for physical circuits connected to the user I/O that cannot otherwise be achieved with software simulators. Besides testing, the debugger can be used as a learning tool to teach basic digital logic. In section 5.4, we compare the testbench debugger with an alternative platform to teach students about logic fault detection.

A test operation is performed by connecting input and output test signals from the debugger to the output and input ports of a module, respectively, as shown below:

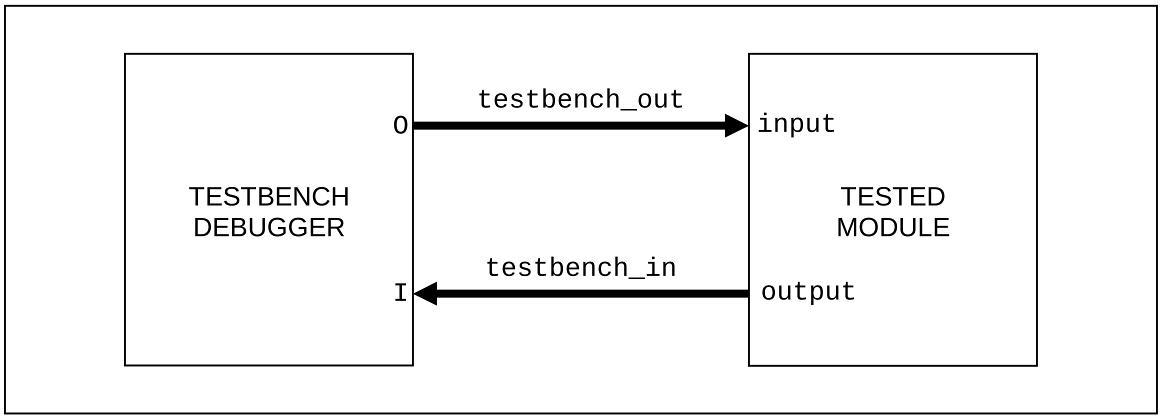


Fig. 8: Testbench debugger operation

Before an FPGA module can be tested, it must be included as a component to be synthesized with the OMICRON core. To do this, the student is responsible for hooking up the necessary test signals to the I/O of the module. It is important to note that the module I/O is not only limited to its main input and output ports. The FPGA freely allows internal signals of the module to be accessed for testing. If the tested component is external to the FPGA, then the internal test signals are assigned to the appropriate user I/O pins. Test vectors are then applied to the inputs of the tested module while the returned output signals are simultaneously captured. In section 5.3, we shall describe ways of applying these test vectors.

The debugger builds upon existing OMICRON subroutines that allow students to interact with it and receive test results via a serial terminal. The presence of terminal interaction, however, does not allow timing-critical tests to be performed, since OMICRON will also be responsible for handling terminal subroutines in addition to the actual test subroutines. The multi-tasking work causes OMICRON be inconsistent in delivering test vectors. Hence, the debugger can only guarantee cycle accuracy in terms of timing the test vectors. For most sequential-logic systems, cycle-accurate test vectors are sufficient for functional testing.

5.2 Testbench debugger implementation

The OMICRON testbench debugger provides, by default, eight 8-bit ports to test a component. Four of the ports act as test input ports (labeled I1 to I4) while the other four act as test output ports (labeled O1 to O4). Vital to the debugger's user interactivity are four 8-bit intermediate vectors, known as next output signals (labeled N1 to N4). In addition, an 8-bit temporary register "R" is provided.

The input ports are responsible for capturing the output signals of the tested module. The output ports are responsible for supplying the tested module with input signals in order to excite the module. The next output signals are used as placeholders to decide what the follow-up output port signals are going to assume on the next "clock" operation. Lastly, the R register is used to perform logic operations on the existing vectors recorded by the debugger in order to provide new values for next output signals.

As shown in Figure 2, the test input ports are implemented as inputs into the input MUX that is fed into the PicoBlaze core. In the upper right corner of the same figure, four data output registers are allocated for the next output signals. The outputs of those registers are fed as inputs to four testbench output registers. This setup allows the next output registers to be configured an arbitrary amount of times before finally having their outputs committed as testbench outputs. As soon as the next output signals are committed, all test output signals are changed (or "clocked") simultaneously.

Shown below is the construction of a testbench output register:

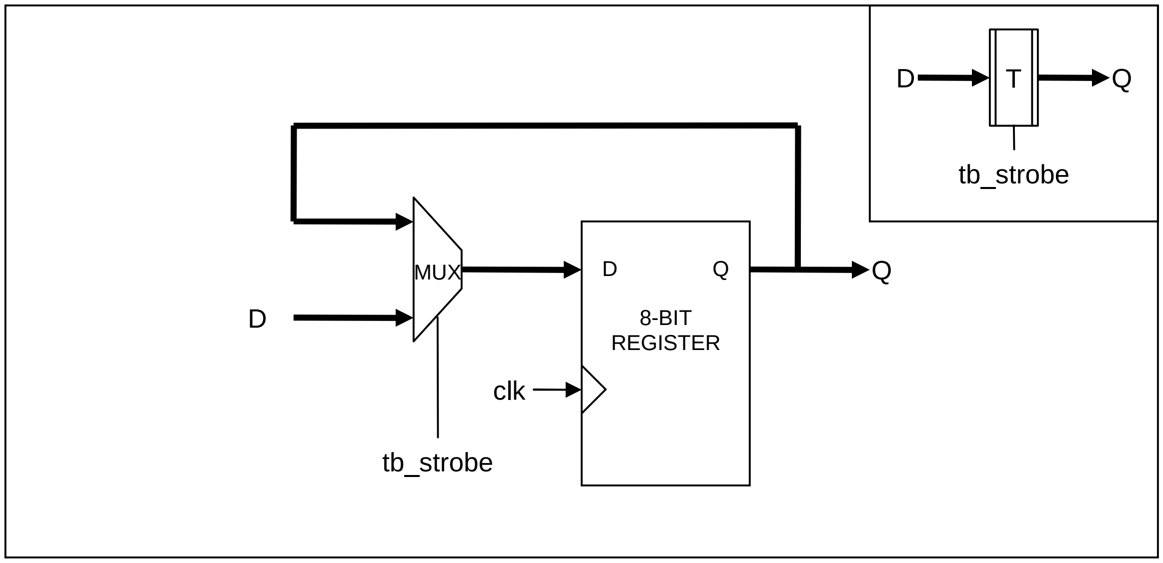


Fig. 9: OMICRON testbench output register

In a testbench output register, the tb_strobe signal is controlled by the debugger to reject or allow the incoming signal to be latched at the output. This ensures simultaneous clocking of the test output registers.

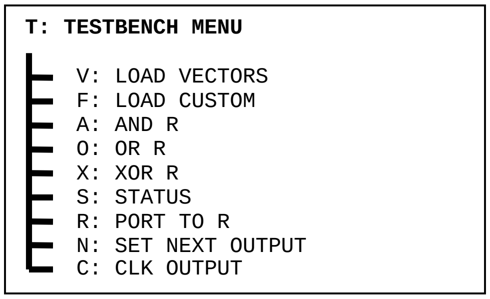
In the testbench debugging mode, an 18kbit ROM is used to supply the debugger with testbench vectors. This vector ROM is connected as an input to OMICRON. As test vectors are read from the ROM, they are redirected as next outputs to be committed to the testbench output registers. The generation and usage of this vector ROM is elaborated in section 5.3.2.

5.3 Module debugging using OMICRON

Once a module is connected to the test I/O of the hardware debugger, it is ready for testing. Testing is achieved by first accessing OMICRON in terminal mode. In the command line interface of the terminal, the commands to issue and manipulate test vectors can be found in the TESTBENCH menu. These commands are explained in section 5.3.1. From the same menu, a command can be issued to start the testbench debugging mode where test vectors are applied continuously as test outputs. A unique test flow associated with this mode is decribed in section 5.3.2.

5.3.1 Using command line interface

When the TESTBENCH menu is accessed, the following command options are offered:



```
T: TESTBENCH MENU

    V: LOAD VECTORS
    F: LOAD CUSTOM
    A: AND R
    O: OR R
    X: XOR R
    S: STATUS
    R: PORT TO R
    N: SET NEXT OUTPUT
    C: CLK OUTPUT
```

Fig. 10: OMICRON testbench menu commands

All of these options revolve around the STATUS command. This is the command where OMICRON displays the status of every test I/O port, including the current values for the next output signals.

When the STATUS command is issued, an output like this is produced:

```
++++++++++++++++++++++++++++++++++++++++++
I1 : DE [11011110]
I2 : DE [11011110]
I3 : 76 [01110110]
I4 : 14 [00010100]

O1 : 02 [00000010] <- N1 : 03 [00000011]
O2 : DA [11011010] <- N2 : DA [11011010]
O3 : 56 [01010110] <- N3 : 56 [01010110]
O4 : 9D [10011101] <- N4 : 9D [10011101]

 R : F5 [11110101]
==========================================
```

Fig. 11: Status of testbench debugger

From the example above, we can see that the two output ports of the tested module connected to I1 and I2 are currently holding the same byte value of "DE" (in hex). The next output signals are currently set to values that are ready to be committed as testbench outputs. For instance, when the outputs are "clocked" with the CLK OUTPUT command, only the first output (O1) changes its value from "02" to "03". The other outputs (O2 to O4) induce no change from the committed next outputs (N2 to N4).

A next output can be manually set with the SET NEXT OUTPUT command by either specifying a new value for it (e.g., issuing "N3 9A") or setting it to be equal to the R register content (e.g., issuing "N3" without a hex value).

The contents of the R register can be changed with the PORT TO R command (e.g., "R 5B"). The value of any port can also be copied to the R register with the same command. For example, to copy the fourth debugger input to the R register, an "R I4" command is issued. After that, the contents of the register can be modified with any of the logic operation commands. For example, with the status as shown in Figure 11, issuing an XOR command "X 60" on the R register changes its contents from "F5" to "95".

The LOAD CUSTOM command executes a custom subroutine defined in the testbench program ROM. Students can customize this subroutine to perform other functions that build upon existing OMICRON capabilities.

Finally, the LOAD VECTORS command begins the testbench debugging mode. The entire use of this mode is expounded in the next section 5.3.2.

5.3.2 Using testbench vectors

Using the CLI to manipulate test vectors and step through an entire test sequence is useful for small test cases. However, to be effective in producing elaborate functional simulations, a large quantity of vectors need to be utilized to test a module.

For this purpose, the debugger supports a testbench mode in which vectors are continuously applied to the test outputs in order to collect a range of results, much like how a software testbench works. These results can then be converted into a form that can be displayed as a graphical waveform by simulation software.

In this section, we shall define the test flow of using testbench vectors to test a module by walking through an example. The test flow was designed to be used in conjunction with the ModelSim® simulation software by Mentor Graphics® [33].

We begin with a module that we wish to test:

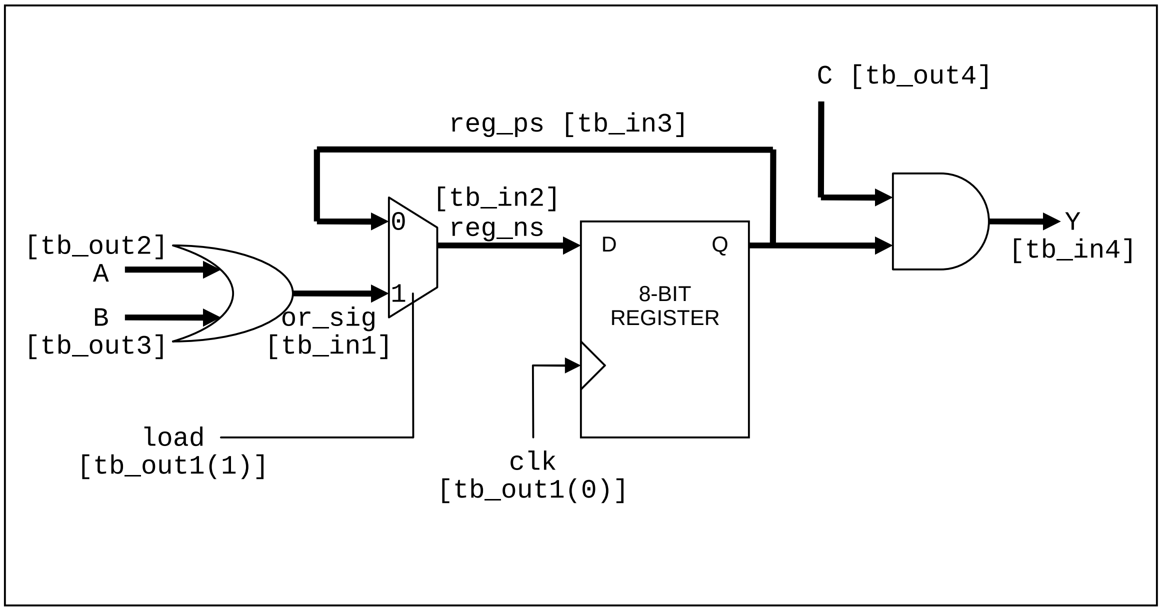


Fig. 12: Tested module example

The figure above shows a Mealy sequential-logic system that has three 8-bit input signals (`A`, `B`, and `C`) with a clock input (`clk`) and a control signal (`load`). There is a single 8-bit output (`Y`).

A bitwise-OR operation is performed on both `A` and `B` inputs and their OR-gate output `or_sig` is fed into a MUX. The `load` signal is then used as a synchronous control to enable or disable the register from latching on to `or_sig`. Finally, the output of the register is bitwise-ANDed with the `C` input to produce output `Y`.

The signal names that are not in the square brackets are the original signal names given to the module. The original signals will be assigned to corresponding testbench I/O signals (with names in the square brackets). Testbench outputs are connected to module inputs, and vice versa. For testbench output `tb_out1`, only two of its eight bits were used.

Once we are certain of how we would test our given module, we are ready to begin the test flow of employing vectors for the testbench. The progression of the flow is shown in the figure below:

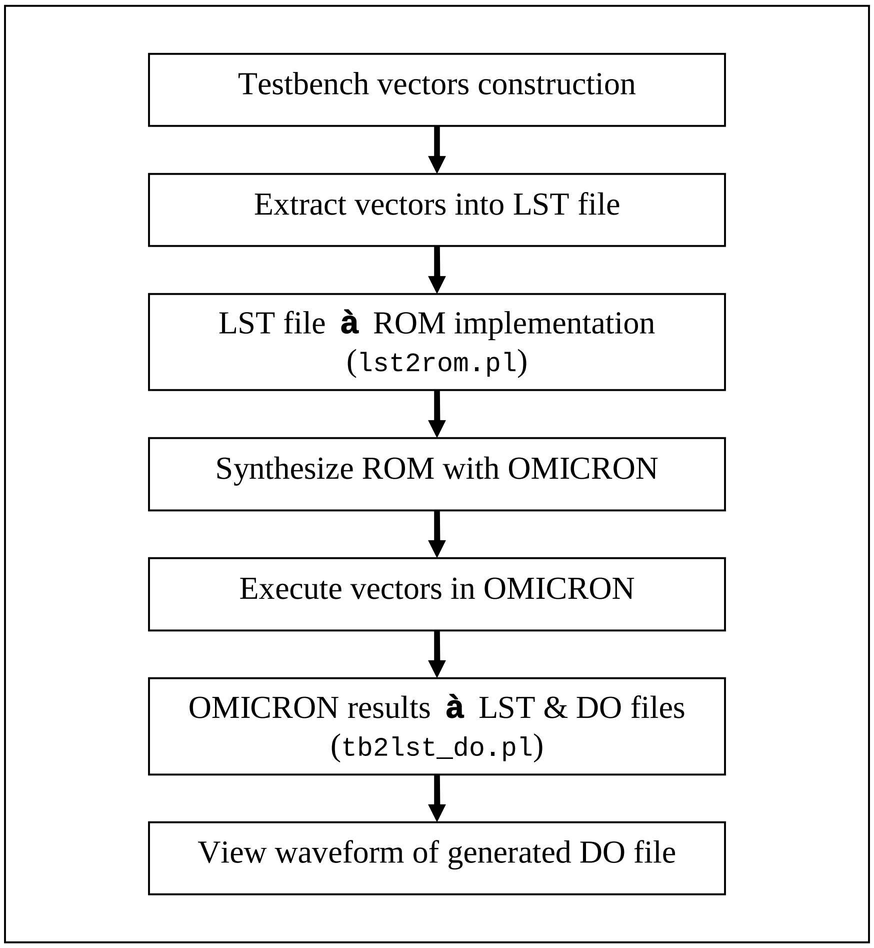

Testbench vectors construction
Extract vectors into LST file
LST file à ROM implementation (lst2rom.pl)
Synthesize ROM with OMICRON
Execute vectors in OMICRON
OMICRON results à LST & DO files (tb2lst_do.pl)
View waveform of generated DO file

Fig. 13: Testbench vector test flow

(1) Testbench vectors construction

Before we can use test bit-vectors for our testbench, we need to construct them first. ModelSim offers a comprehensive scripting language to describe vectors that are used for simulations within its environment.

Presumably, we would have loaded a design of the module given in Figure 12 into the ModelSim environment. Then, we would perform a simulation test on the design using vectors scripted in a macro DO file. Shown below is the macro script that we use to test our module:

```
view signals

view wave
add wave: /testmodule/*

force -freeze /testmodule/clk 0 0, 1 {50 ns} -r 100
force -freeze /testmodule/load 1 0
force -freeze /testmodule/A 00000000 0
force -freeze /testmodule/B 00000000 0
force -freeze /testmodule/C 00000000 0
run 100

force -freeze /testmodule/A 01010110 0
force -freeze /testmodule/B 01110010 0
force -freeze /testmodule/C 11010101 0
run 100

force -freeze /testmodule/A 11011010 0
force -freeze /testmodule/B 01010110 0
force -freeze /testmodule/C 10011101 0
run 100

force -freeze /testmodule/A 11110000 0
force -freeze /testmodule/B 00001110 0
force -freeze /testmodule/C 00111111 0
run 100
```

Fig. 14: DO file (`testmodule.do`) to create vectors and generate waveform

In the DO file above, we define some input signals to test the module over four clock cycles. When we execute the DO file within ModelSim, a graphical waveform of the simulation is generated as shown on the next page.

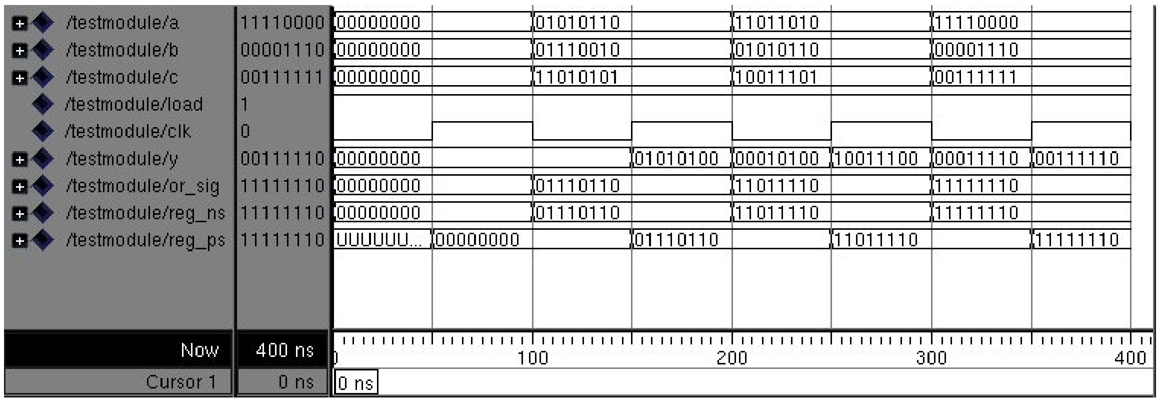
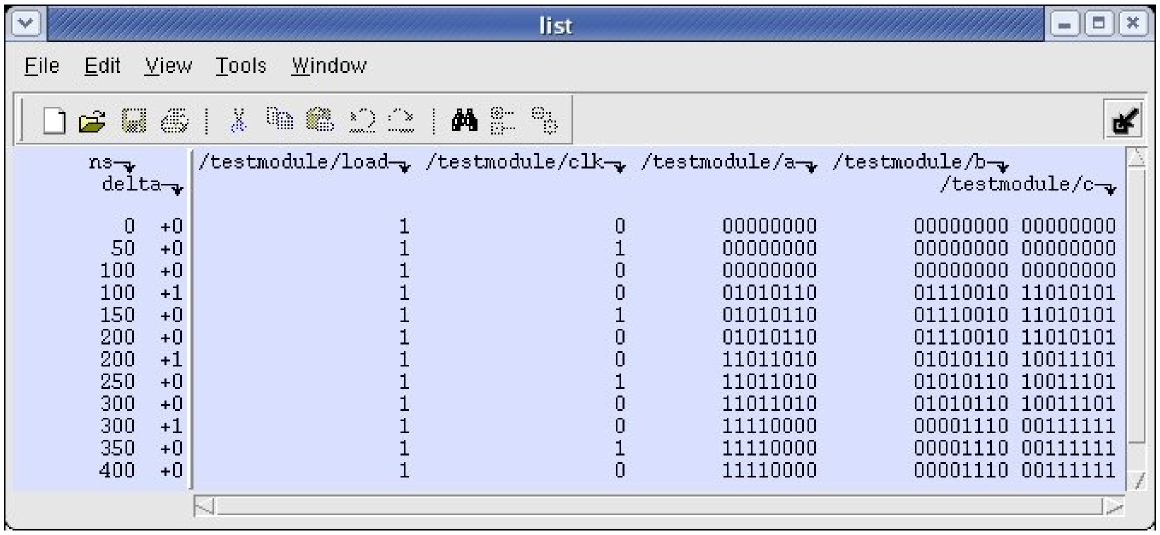


Fig. 15: ModelSim waveform of original simulation

Signals shown in a waveform can easily be viewed and distinguished, which makes this graphical representation of signals ideal in a test environment. Once we have a waveform of the original simulation for comparison, we can extract the vectors for use in OMICRON.

(2) <u>Extract vectors into LST file</u>

The waveform shows how vectors of the module signals change over time. These signals can be extracted and stored as arrays of vectors that are arranged according to time. To do this in ModelSim, we simply drag and drop input signals from the "signal" window into the "list" window so that they are arranged as shown in the figure on the next page.



```
       ns      /testmodule/load  /testmodule/clk  /testmodule/a  /testmodule/b
      delta                                                                /testmodule/c

   0  +0                      1                0       00000000       00000000 00000000
  50  +0                      1                1       00000000       00000000 00000000
 100  +0                      1                0       00000000       00000000 00000000
 100  +1                      1                0       01010110       01110010 11010101
 150  +0                      1                1       01010110       01110010 11010101
 200  +0                      1                0       01010110       01110010 11010101
 200  +1                      1                0       11011010       01010110 10011101
 250  +0                      1                1       11011010       01010110 10011101
 300  +0                      1                0       11011010       01010110 10011101
 300  +1                      1                0       11110000       00001110 00111111
 350  +0                      1                1       11110000       00001110 00111111
 400  +0                      1                0       11110000       00001110 00111111
```

Fig. 16: ModelSim extracted list window

From the figure above, 12 rows of vectors were extracted from the original simulation that was performed on the tested module. Out of the 12 rows, 3 of them are deltas which indicate several vector changes have taken place within a single time slot.

The extracted vectors are saved into the tabular LST file `omi_vector.lst`:

```
         ns       /testmodule/load   /testmodule/clk   /testmodule/a   /testmodule/b
          delta                                                               /testmodule/c
          0  +0                  1                 0        00000000        00000000 00000000
         50  +0                  1                 1        00000000        00000000 00000000
        100  +0                  1                 0        00000000        00000000 00000000
        100  +1                  1                 0        01010110        01110010 11010101
        150  +0                  1                 1        01010110        01110010 11010101
        200  +0                  1                 0        01010110        01110010 11010101
        200  +1                  1                 0        11011010        01010110 10011101
        250  +0                  1                 1        11011010        01010110 10011101
        300  +0                  1                 0        11011010        01010110 10011101
        300  +1                  1                 0        11110000        00001110 00111111
        350  +0                  1                 1        11110000        00001110 00111111
        400  +0                  1                 0        11110000        00001110 00111111
```

Fig. 17: LST file (`omi_vector.lst`) of extracted vectors

(3) <u>LST file to ROM implementation using `lst2rom.pl` Perl script</u>

For OMICRON to use extracted vectors, the LST file must be converted to a ROM implementation that is used as a repository for the test vectors. This is achieved by running the `lst2rom.pl` Perl script with `omi_vector.lst` as the input file. The script then uses a VHDL form file `vectorROM_form.vhd` to generate an FPGA-synthesizable macro as shown in Figure 18 on the next page.

During conversion, the script parses the rows of vectors of the LST file into word lengths of 32 bits to match 32 bits of the testbench output. The most significant vector byte is assigned to `tb_out1`. This is followed by the second byte being assigned to `tb_out2` and so on.

If the number of vector bits per row is less than 32, "0" bits are padded on the left to fill the gap. In the case of `omi_vector.lst`, six "0" bits were added on the left, fixing the six significant bits of `tb_out1` as zeroes. If there is more than 32 bits per row, the script fails with an error.

When delta rows exist, the script would only parse the last delta row with the latest vectors. Delta cycles are not used by the testbench debugger because the test outputs need to be simultaneously clocked (see section 5.2). This effectively reduces the number of testbench vector rows of `omi_vector.lst` down to 9.

The script marks the end of the list of vectors with a "1" bit. In Figure 18, "INITP_00" has a "1" on the ninth bit from the right to indicate nine stored vectors.

Each ROM can hold 512 cycles of testbench output vectors. Should the input LST file contain more than 512 effective vector rows, additional ROMs are generated by the script to hold more vectors (e.g., `omi_vector1.vhd`, `omi_vector2.vhd`, etc.).

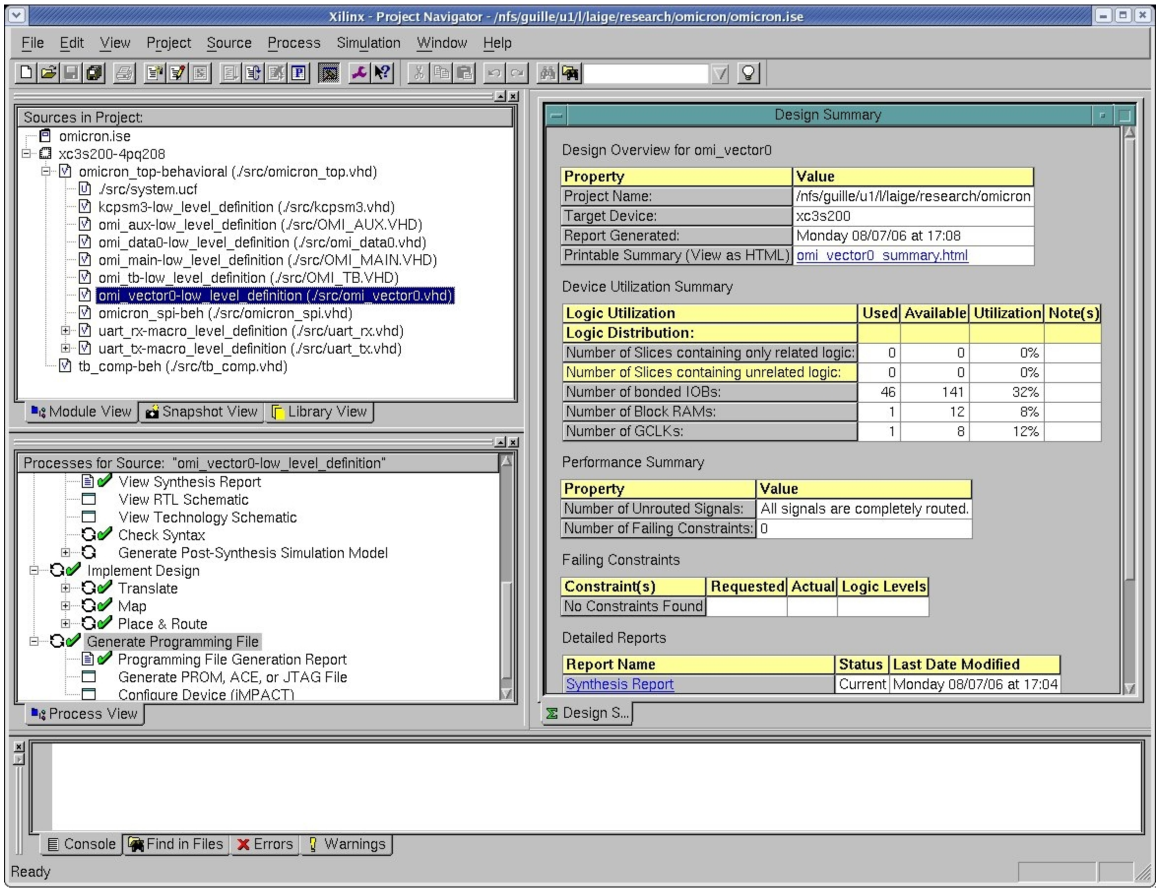
```
--
-- Definition of a single port ROM for OMICRON testbench vectors defined by omi_vector.lst
--
-- Generated by lst2rom.pl [8/7/2006 0:32]
--
-- Standard IEEE libraries
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
--
-- The Unisim Library is used to define Xilinx primitives. It is also used during
-- simulation. The source can be viewed at %XILINX%\vhdl\src\unisims\unisim_VCOMP.vhd
--
library unisim;
use unisim.vcomponents.all;
--
--
entity omi_vector0 is
  port(
        address : in  std_logic_vector(8 downto 0);
        data    : out std_logic_vector(35 downto 0);
        clk     : in  std_logic
      );
end entity omi_vector0;
--
architecture low_level_definition of omi_vector0 is
--
-- Attributes to define ROM contents during implementation synthesis.
-- The information is repeated in the generic map for functional simulation
--

[attributes removed to keep file short]

--
begin
  -- Instantiate the Xilinx primitive for a block RAM.
  ram_512_x_36: RAMB16_S36
  --synthesis translate_off
  -- INIT values repeated to define contents for functional simulation.
  generic map( INIT       => X"000000000",
               SRVAL      => X"000000000",
               WRITE_MODE => "WRITE_FIRST",
               INIT_00    => X"03F00E3F02F00E3F03DA569D02DA569D035672D5025672D50300000002000000",
               INIT_01    => X"0000000000000000000000000000000000000000000000000000000002F00E3F",

               [...]

               INIT_3F    => X"0000000000000000000000000000000000000000000000000000000000000000",
               INITP_00   => X"0000000000000000000000000000000000000000000000000000000100000000",

               [...]

               INITP_07   => X"0000000000000000000000000000000000000000000000000000000000000000"
             )
  --synthesis translate_on
  port map( DO   => data(31 downto 0),
            DOP  => data(35 downto 32),
            ADDR => address,
            CLK  => clk,
            DI   => X"00000000",
            DIP  => X"0",
            EN   => '1',
            SSR  => '0',
            WE   => '0'
          );
--
end architecture low_level_definition;
```

Fig. 18: VHDL file (`omi_vector0.vhd`) of ROM implementation

(4) Synthesize ROM with OMICRON

The ROM implementation `omi_vector0.vhd` is included as a component of the OMICRON system. Once it is synthesized and implemented into the FPGA, OMICRON is able to access the vectors for testbench debugging. Figure 19 shows the Xilinx ISE window where `omi_vector0.vhd` is included in OMICRON.

At this time, we include the tested module into OMICRON and implement it as was shown in Figure 12.


Xilinx - Project Navigator - /nfs/guille/u1/l/laige/research/omicron/omicron.ise
File Edit View Project Source Process Simulation Window Help
Module View
Snapshot View
Library View
Process View
Design S...
Console
Find in Files
Errors
Warnings
Ready

Sources in Project:
- omicron.ise
- xc3s200-4pq208
  - omicron_top-behavioral (./src/omicron_top.vhd)
    - ./src/system.ucf
    - kcpsm3-low_level_definition (./src/kcpsm3.vhd)
    - omi_aux-low_level_definition (./src/OMI_AUX.VHD)
    - omi_data0-low_level_definition (./src/omi_data0.vhd)
    - omi_main-low_level_definition (./src/OMI_MAIN.VHD)
    - omi_tb-low_level_definition (./src/OMI_TB.VHD)
    - omi_vector0-low_level_definition (./src/omi_vector0.vhd)
    - omicron_spi-beh (./src/omicron_spi.vhd)
    - uart_rx-macro_level_definition (./src/uart_rx.vhd)
    - uart_tx-macro_level_definition (./src/uart_tx.vhd)
  - tb_comp-beh (./src/tb_comp.vhd)

Processes for Source: "omi_vector0-low_level_definition"
- View Synthesis Report
- View RTL Schematic
- View Technology Schematic
- Check Syntax
- Generate Post-Synthesis Simulation Model
- Implement Design
  - Translate
  - Map
  - Place & Route
- Generate Programming File
  - Programming File Generation Report
  - Generate PROM, ACE, or JTAG File
  - Configure Device (iMPACT)

Design Summary

Design Overview for omi_vector0

Property	Value
Project Name:	/nfs/guille/u1/l/laige/research/omicron
Target Device:	xc3s200
Report Generated:	Monday 08/07/06 at 17:08
Printable Summary (View as HTML)	omi_vector0_summary.html

Device Utilization Summary

Logic Utilization	Used	Available	Utilization	Note(s)
Logic Distribution:				
Number of Slices containing only related logic:	0	0	0%	
Number of Slices containing unrelated logic:	0	0	0%	
Number of bonded IOBs:	46	141	32%	
Number of Block RAMs:	1	12	8%	
Number of GCLKs:	1	8	12%	

Performance Summary

Property	Value
Number of Unrouted Signals:	All signals are completely routed.
Number of Failing Constraints:	0

Failing Constraints

Constraint(s)	Requested	Actual	Logic Levels
No Constraints Found			

Detailed Reports

Report Name	Status	Last Date Modified
Synthesis Report	Current	Monday 08/07/06 at 17:04

Fig. 19: Synthesize ROM implementation with OMICRON

Currently, OMICRON supports up to a maximum of 128 vector ROMs to be chained together, thus providing as much as 65536 cycles of testbench vectors.

(5) Execute vectors in OMICRON

As mentioned in section 5.3.1, testbench debugging is started with the LOAD VECTORS command. As soon as the testbench starts, vectors incrementally read from ROM are continuously sent to the testbench outputs to test the implemented module.

This is done by first assigning newly read vectors as next outputs. Then, a STATUS command is performed to display the values of the test signals. Finally, a CLK OUTPUT command is issued to commit the current next outputs before reading in a new set of next outputs.

The results generated by the testbench debugger are saved from the terminal into the file `debug.tb` as shown in Figure 20 on the next page. These results resemble the output of many STATUS commands (see section 5.3.1). Each status block represents one testbench vector cycle and is numbered beginning from "00000".

```
00000
+++++++++++++++++++++++++++++++++++++++++
I1 : 00 [00000000]
I2 : 00 [00000000]
I3 : 00 [00000000]
I4 : 00 [00000000]

O1 : 00 [00000000] <- N1 : 02 [00000010]
O2 : 00 [00000000] <- N2 : 00 [00000000]
O3 : 00 [00000000] <- N3 : 00 [00000000]
O4 : 00 [00000000] <- N4 : 00 [00000000]

 R : 00 [00000000]
=========================================

00001
+++++++++++++++++++++++++++++++++++++++++
I1 : 00 [00000000]
I2 : 00 [00000000]
I3 : 00 [00000000]
I4 : 00 [00000000]

O1 : 02 [00000010] <- N1 : 03 [00000011]
O2 : 00 [00000000] <- N2 : 00 [00000000]
O3 : 00 [00000000] <- N3 : 00 [00000000]
O4 : 00 [00000000] <- N4 : 00 [00000000]

 R : 00 [00000000]
=========================================

00002
+++++++++++++++++++++++++++++++++++++++++
I1 : 00 [00000000]
I2 : 00 [00000000]
I3 : 00 [00000000]
I4 : 00 [00000000]

O1 : 03 [00000011] <- N1 : 02 [00000010]
O2 : 00 [00000000] <- N2 : 56 [01010110]
O3 : 00 [00000000] <- N3 : 72 [01110010]
O4 : 00 [00000000] <- N4 : D5 [11010101]

 R : 00 [00000000]
=========================================

00003
+++++++++++++++++++++++++++++++++++++++++
I1 : 76 [01110110]
I2 : 76 [01110110]
I3 : 00 [00000000]
I4 : 00 [00000000]

O1 : 02 [00000010] <- N1 : 03 [00000011]
O2 : 56 [01010110] <- N2 : 56 [01010110]
O3 : 72 [01110010] <- N3 : 72 [01110010]
O4 : D5 [11010101] <- N4 : D5 [11010101]

 R : 00 [00000000]
=========================================

00004
+++++++++++++++++++++++++++++++++++++++++
I1 : 76 [01110110]
I2 : 76 [01110110]
I3 : 76 [01110110]
I4 : 54 [01010100]

O1 : 03 [00000011] <- N1 : 02 [00000010]
O2 : 56 [01010110] <- N2 : DA [11011010]
O3 : 72 [01110010] <- N3 : 56 [01010110]
O4 : D5 [11010101] <- N4 : 9D [10011101]

 R : 00 [00000000]
=========================================
```

```
00005
+++++++++++++++++++++++++++++++++++++++++
I1 : DE [11011110]
I2 : DE [11011110]
I3 : 76 [01110110]
I4 : 14 [00010100]

O1 : 02 [00000010] <- N1 : 03 [00000011]
O2 : DA [11011010] <- N2 : DA [11011010]
O3 : 56 [01010110] <- N3 : 56 [01010110]
O4 : 9D [10011101] <- N4 : 9D [10011101]

 R : 00 [00000000]
=========================================

00006
+++++++++++++++++++++++++++++++++++++++++
I1 : DE [11011110]
I2 : DE [11011110]
I3 : DE [11011110]
I4 : 9C [10011100]

O1 : 03 [00000011] <- N1 : 02 [00000010]
O2 : DA [11011010] <- N2 : F0 [11110000]
O3 : 56 [01010110] <- N3 : 0E [00001110]
O4 : 9D [10011101] <- N4 : 3F [00111111]

 R : 00 [00000000]
=========================================

00007
+++++++++++++++++++++++++++++++++++++++++
I1 : FE [11111110]
I2 : FE [11111110]
I3 : DE [11011110]
I4 : 1E [00011110]

O1 : 02 [00000010] <- N1 : 03 [00000011]
O2 : F0 [11110000] <- N2 : F0 [11110000]
O3 : 0E [00001110] <- N3 : 0E [00001110]
O4 : 3F [00111111] <- N4 : 3F [00111111]

 R : 00 [00000000]
=========================================

00008
+++++++++++++++++++++++++++++++++++++++++
I1 : FE [11111110]
I2 : FE [11111110]
I3 : FE [11111110]
I4 : 3E [00111110]

O1 : 03 [00000011] <- N1 : 02 [00000010]
O2 : F0 [11110000] <- N2 : F0 [11110000]
O3 : 0E [00001110] <- N3 : 0E [00001110]
O4 : 3F [00111111] <- N4 : 3F [00111111]

 R : 00 [00000000]
=========================================

00009
+++++++++++++++++++++++++++++++++++++++++
I1 : FE [11111110]
I2 : FE [11111110]
I3 : FE [11111110]
I4 : 3E [00111110]

O1 : 02 [00000010] <- N1 : 02 [00000010]
O2 : F0 [11110000] <- N2 : F0 [11110000]
O3 : 0E [00001110] <- N3 : 0E [00001110]
O4 : 3F [00111111] <- N4 : 3F [00111111]

 R : 00 [00000000]
=========================================
```

Fig. 20: Saved testbench debugger output (`debug.tb`)

(6) <u>OMICRON results to LST & DO files using `tb2lst_do.pl` Perl script</u>

At this point, the hardware testbench is completed. The results from OMICRON can be converted back into vectors that can be read by simulation software. This is achieved by using the `tb2lst_do.pl` script to convert the text TB file `debug.tb` into two formats:

(i) <u>LST file `debug.lst`</u>

This is the same format used by ModelSim and other CAD tools to store bit vectors of signals.

```
ns       /omicron/tb_out1
 delta            /omicron/tb_out2
                           /omicron/tb_out3
                                    /omicron/tb_out4
                                              /omicron/tb_in1
                                                       /omicron/tb_in2
                                                                /omicron/tb_in3
                                                                         /omicron/tb_in4
 0  +0           00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
10  +0           00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000
20  +0           00000011 00000000 00000000 00000000 00000000 00000000 00000000 00000000
30  +0           00000010 01010110 01110010 11010101 01110110 01110110 00000000 00000000
40  +0           00000011 01010110 01110010 11010101 01110110 01110110 01110110 01010100
50  +0           00000010 11011010 01010110 10011101 11011110 11011110 01110110 00010100
60  +0           00000011 11011010 01010110 10011101 11011110 11011110 11011110 10011100
70  +0           00000010 11110000 00001110 00111111 11111110 11111110 11011110 00011110
80  +0           00000011 11110000 00001110 00111111 11111110 11111110 11111110 00111110
90  +0           00000010 11110000 00001110 00111111 11111110 11111110 11111110 00111110
```

Fig. 21: LST file (`debug.lst`) of testbench debugger output

(ii) <u>Macro DO file `tb_wave.do`</u>

The DO file `tb_wave.do` allows ModelSim to generate a waveform of the testbench debugger signals by means of an OMICRON mock system. The mock system is an empty shell that is used for the purpose of holding and reassigning signals.

```
view signals
view wave
add wave: /omicron/*

force -freeze /omicron/tb_out1 00000000 0
force -freeze /omicron/tb_out2 00000000 0
force -freeze /omicron/tb_out3 00000000 0
force -freeze /omicron/tb_out4 00000000 0
force -freeze /omicron/tb_in1  00000000 0
force -freeze /omicron/tb_in2  00000000 0
force -freeze /omicron/tb_in3  00000000 0
force -freeze /omicron/tb_in4  00000000 0
run 10

force -freeze /omicron/tb_out1 00000010 0
force -freeze /omicron/tb_out2 00000000 0
force -freeze /omicron/tb_out3 00000000 0
force -freeze /omicron/tb_out4 00000000 0
force -freeze /omicron/tb_in1  00000000 0
force -freeze /omicron/tb_in2  00000000 0
force -freeze /omicron/tb_in3  00000000 0
force -freeze /omicron/tb_in4  00000000 0
run 10

force -freeze /omicron/tb_out1 00000011 0
force -freeze /omicron/tb_out2 00000000 0
force -freeze /omicron/tb_out3 00000000 0
force -freeze /omicron/tb_out4 00000000 0
force -freeze /omicron/tb_in1  00000000 0
force -freeze /omicron/tb_in2  00000000 0
force -freeze /omicron/tb_in3  00000000 0
force -freeze /omicron/tb_in4  00000000 0
run 10

force -freeze /omicron/tb_out1 00000010 0
force -freeze /omicron/tb_out2 01010110 0
force -freeze /omicron/tb_out3 01110010 0
force -freeze /omicron/tb_out4 11010101 0
force -freeze /omicron/tb_in1  01110110 0
force -freeze /omicron/tb_in2  01110110 0
force -freeze /omicron/tb_in3  00000000 0
force -freeze /omicron/tb_in4  00000000 0
run 10

force -freeze /omicron/tb_out1 00000011 0
force -freeze /omicron/tb_out2 01010110 0
force -freeze /omicron/tb_out3 01110010 0
force -freeze /omicron/tb_out4 11010101 0
force -freeze /omicron/tb_in1  01110110 0
force -freeze /omicron/tb_in2  01110110 0
force -freeze /omicron/tb_in3  01110110 0
force -freeze /omicron/tb_in4  01010100 0
run 10
```

```
force -freeze /omicron/tb_out1 00000010 0
force -freeze /omicron/tb_out2 11011010 0
force -freeze /omicron/tb_out3 01010110 0
force -freeze /omicron/tb_out4 10011101 0
force -freeze /omicron/tb_in1  11011110 0
force -freeze /omicron/tb_in2  11011110 0
force -freeze /omicron/tb_in3  01110110 0
force -freeze /omicron/tb_in4  00010100 0
run 10

force -freeze /omicron/tb_out1 00000011 0
force -freeze /omicron/tb_out2 11011010 0
force -freeze /omicron/tb_out3 01010110 0
force -freeze /omicron/tb_out4 10011101 0
force -freeze /omicron/tb_in1  11011110 0
force -freeze /omicron/tb_in2  11011110 0
force -freeze /omicron/tb_in3  11011110 0
force -freeze /omicron/tb_in4  10011100 0
run 10

force -freeze /omicron/tb_out1 00000010 0
force -freeze /omicron/tb_out2 11110000 0
force -freeze /omicron/tb_out3 00001110 0
force -freeze /omicron/tb_out4 00111111 0
force -freeze /omicron/tb_in1  11111110 0
force -freeze /omicron/tb_in2  11111110 0
force -freeze /omicron/tb_in3  11011110 0
force -freeze /omicron/tb_in4  00011110 0
run 10

force -freeze /omicron/tb_out1 00000011 0
force -freeze /omicron/tb_out2 11110000 0
force -freeze /omicron/tb_out3 00001110 0
force -freeze /omicron/tb_out4 00111111 0
force -freeze /omicron/tb_in1  11111110 0
force -freeze /omicron/tb_in2  11111110 0
force -freeze /omicron/tb_in3  11111110 0
force -freeze /omicron/tb_in4  00111110 0
run 10

force -freeze /omicron/tb_out1 00000010 0
force -freeze /omicron/tb_out2 11110000 0
force -freeze /omicron/tb_out3 00001110 0
force -freeze /omicron/tb_out4 00111111 0
force -freeze /omicron/tb_in1  11111110 0
force -freeze /omicron/tb_in2  11111110 0
force -freeze /omicron/tb_in3  11111110 0
force -freeze /omicron/tb_in4  00111110 0
run 10
```

Fig. 22: DO file (`tb_wave.do`) to generate waveform of testbench debugger signals

Since the script is not able to recognize how the original signals of the tested module were assigned to the testbench signals, the mock system allows the original signals to be extracted from the testbench signals for the purpose of viewing their waveforms.

```
--
-- This is a mock system to mimic the OMICRON testbench.
-- A behavioral compile of this system is required to display the waveform
-- defined by the ModelSim macro file tb_wave.do that is generated from
-- the tb2lst_do.pl script.
--
-- Additional signals can be defined and assigned to the existing inputs
-- and outputs to make the output waveform clearer to understand from the
-- perspective of the module that was tested by the testbench.
--

library ieee;
use ieee.std_logic_1164.all;

entity omicron is
  port(
        tb_in1  : in     std_logic_vector(7 downto 0);
        tb_in2  : in     std_logic_vector(7 downto 0);
        tb_in3  : in     std_logic_vector(7 downto 0);
        tb_in4  : in     std_logic_vector(7 downto 0);
        tb_out1 : buffer std_logic_vector(7 downto 0);
        tb_out2 : buffer std_logic_vector(7 downto 0);
        tb_out3 : buffer std_logic_vector(7 downto 0);
        tb_out4 : buffer std_logic_vector(7 downto 0)
      );
end entity omicron;

architecture mock of omicron is

  -----------------------------
  -- define original signals --
  -----------------------------
  signal A      : std_logic_vector(7 downto 0);
  signal B      : std_logic_vector(7 downto 0);
  signal C      : std_logic_vector(7 downto 0);
  signal load   : std_logic;
  signal clk    : std_logic;
  signal Y      : std_logic_vector(7 downto 0);
  signal or_sig : std_logic_vector(7 downto 0);
  signal reg_ns : std_logic_vector(7 downto 0);
  signal reg_ps : std_logic_vector(7 downto 0);

begin

  -------------------------------------------------------------
  -- connect original signals to corresponding testbench I/O --
  -------------------------------------------------------------
  A <= tb_out2;
  B <= tb_out3;
  C <= tb_out4;
  load <= tb_out1(1);
  clk <= tb_out1(0);

  Y <= tb_in4;
  or_sig <= tb_in1;
  reg_ns <= tb_in2;
  reg_ps <= tb_in3;

end architecture mock;
```

Fig. 23: VHDL file (`omicron.vhd`) of OMICRON mock system to view waveform

In `omicron.vhd`, the original signals were redefined and connected to their corresponding testbench I/O in order to compare how the tested module functioned in hardware as compared to the software simulation.

(7) View waveform of generated DO file

The waveform of the hardware testbench results is compared to the original software simulation:

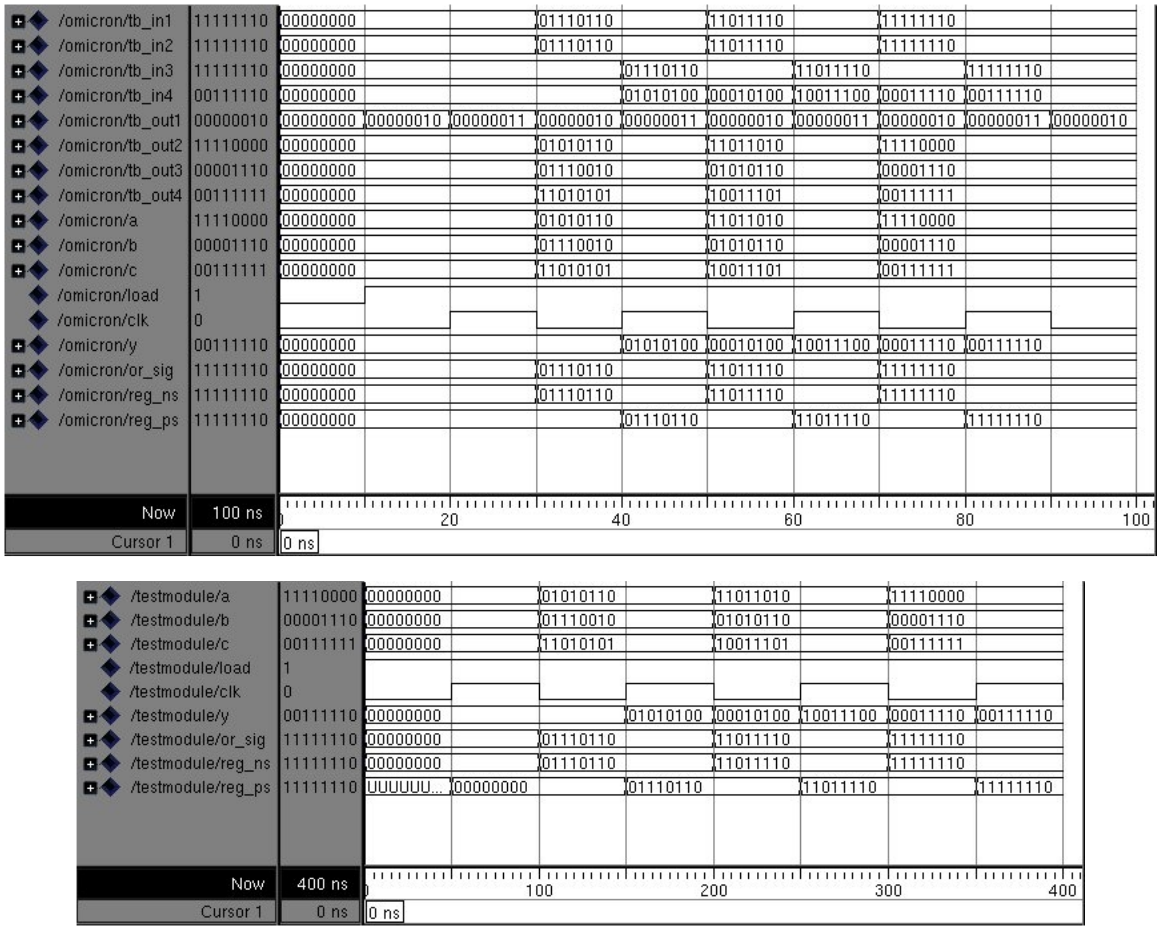


Fig. 24: ModelSim waveform of mock system vs. original simulation (bottom)

The identical results of the original signals confirm that the operation of the tested module in hardware functions as it was simulated in ModelSim. Upon closer inspection, there are 3 minor differences to be noted:

- The hardware testbench results contain 2 extra cycles that are essentially the starting and ending cycles of the software simulation. This appears as a result of the debugger having to establish next output signals before committing those signals as outputs.

- § The `reg_ps` signal of the software simulation begins as undefined. For the hardware testbench, it begins as zero. This is due to the FPGA hardware initialization of resetting signals to zeroes.
- § The cycle time lengths for both waveforms are not the same. It does not matter for the hardware debugger since it can only guarantee cycle accuracy in terms of timing. The cycle length for the hardware simulation is fixed at 10ns by the `tb2lst_do.pl` script.

5.4 Comparison with work by Niggemeyer et. al.

In 2001, Niggemeyer et. al. presented a novel approach to education in manufacturing test and automatic test equipment [34]. This approach involved using an industry-class HP83000 IC tester to probe an FPGA chip that was configured to emulate various logic faults. The objective was to demonstrate fault detection techniques to graduate students.

The approach that Niggemeyer et. al. took was restricted to testing digital circuits only. This makes an interesting comparison with the hardware testbench debugger provided by OMICRON. Since testing only involved digital circuits, the HP83000 tester that was capable of performing mixed-signal probes only utilized part of its capabilities. OMICRON, however, takes a different approach by integrating the tester within the FPGA itself thus making full use of the FPGA's digital architecture and avoiding an external tester altogether. The absence of an external tester makes the FPGA learning platform with OMICRON a more portable and inexpensive solution to the bulky FPGA-and-IC tester combination.

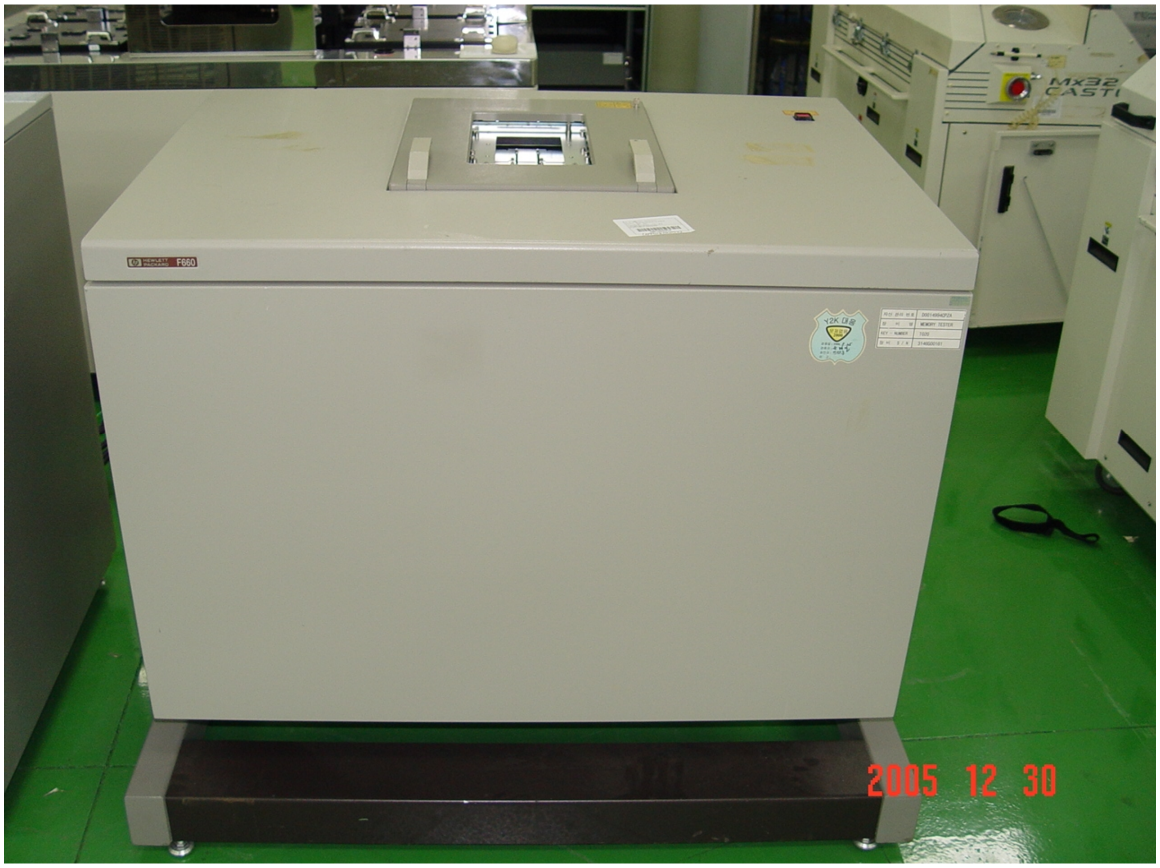


Fig. 25: HP83000 IC tester base

Even though the OMICRON debugger only performs cycle-accurate tests, it can do a majority of digital logic fault tests (with the exception of testing timing-dependent faults). This renders the IC tester's capability to capture timing-accurate test results as non-vital for the objective it was set out to do.

In the learning environment, the IC tester was housed in a laboratory where it was shared among students. This stifles the learning growth of students as they are not able to derive personal ownership from a single learning platform. Also, by isolating the platform, the involvement of the community of learners is severely reduced and this in turn diminishes the consolidation of the entire learning experience that is derived from the support of the community.

The FPGA-and-IC tester combination was targeted for a specific use. As a result, the FPGA device could not be applied elsewhere as it was already fixed on a specialized load board. This eliminated the possibility of carrying the platform over to another class in order to continue building on the existing scaffold of knowledge. On the other hand, the FPGA learning platform aims to preserve flexibility and general usefulness for it to be reapplied in any practical environment.

Shown below is a side-by-side comparison of the two platforms:

	FPGA Learning Platform with OMICRON	**Niggemeyer et. al. FPGA + IC tester [34]**
FPGA device	Xilinx XC3S200	Xilinx XC4005E
FPGA capacity	200k gates	9k gates
Logic tester	OMICRON with Xilinx PicoBlaze core (implemented into FPGA)	HP83000 mixed-signal IC tester (external of FPGA)
Support equipment	Computer, serial cable	Mainframe, testhead, testhead manipulator, cooling unit, workstation
I/O channels	64	64
Operating frequency	Cycle-accurate only	50MHz maximum
Requires load board?	No, all testing done within FPGA; external testing allowed via user I/O	Yes, specialized load board for tester built around FPGA

Table 2: Platforms to teach digital logic fault detection

6 CONCLUSION

This thesis provides an introduction to a new learning platform that is composed of an FPGA development board and a set of free CAD tools. This FPGA platform is the latest in a series of learning platforms developed at OSU. The platform is to be integrated into the coursework for senior undergraduate and graduate students. It incorporates the attributes that allow it to be effective as a learning platform.

These attributes were discovered, studied, and refined from the experience that was acquired from the history of the TekBots program. The first two chapters of this thesis define how these attributes can make an active change in the engineering learning environment.

When students purchase the new FPGA board, they receive a powerful tool that is not a toned-down evaluation-type board. The price of the FPGA board is affordable and very reasonable considering the fact that it incorporates features only found on higher priced FPGA development systems. Currently, each board is priced at approximately USD$85.

The FPGA board is more complex than any learning platform to date. Its main processor, the FPGA device, provides an unprecedented amount of flexibility. This allows an unconventional approach to be taken for testing newly assembled boards. In the fourth chapter of this thesis, we present a post-manufacturing test suite that relies on the FPGA as a tester core that is able to test the peripherals of the same board it is on.

An interactive set of tests was developed for the suite. These tests allow a technician on the manufacturing floor to quickly and efficiently determine a defective board component. Once post-manufacturing tests were completed, the test suites will remain in the FPGA boards and are passed on to students. This allows students to test their boards using the same tests that were used in production.

During the development of the test suite, an interesting feature was discovered and developed. This feature allows students to test their FPGA implementations as well as external circuit designs and compare them directly against an HDL simulation. The fifth chapter of this thesis provides details for a walkthrough on the usage of the hardware testbench debugger.

Currently, the hardware debugger that was developed can only perform cycle-accurate testing. As a suggestion for future work, the debugger may be modified such that it can perform timing-accurate tests. This would involve making use of a specialized core that could queue test vectors in a FIFO (First-In, First-Out) buffer. With this, timing of the testbench operations can be streamlined to a constant delta that is small enough for practical use.

Bibliography

[1] (2006) TekBots website, Oregon State Univ., Corvallis, OR. [Online]. Available: http://eecs.oregonstate.edu/education/tekbots.html

[2] R. Traylor, D. Heer, and T. Fiez, "Using an integrated platform for learning to reinvent engineering education," *IEEE Trans. Educ. (Special Issue on A Vision for ECE Education in 2013 and Beyond)*, vol. 46, pp. 409-419, Nov. 2003.

[3] D. Heer, R. Traylor, T. Thompson, and T. Fiez, "Enhancing the freshman and sophomore ECE student experience using a platform for learning," *IEEE Trans. Educ. (Special Issue on A Vision for ECE Education in 2013 and Beyond)*, vol. 46, pp. 434-443, Nov. 2003.

[4] L. Carley, and P. Khosla, *Experimental Context for Introduction to Electrical and Computer Engineering*. McGraw-Hill, 1997.

[5] L. Carley, and P. Khosla, *Introduction to Electrical and Computer Engineering Taught in Context (Second Edition)*. McGraw-Hill, 1998.

[6] L. Carley, P. Khosla, and R. Unetich, "Teaching 'introduction to electrical and computer engineering' in context," *Proc. IEEE*, vol. 88, pp. 8-22, Jan. 2000.

[7] T. Thompson, D. Heer, S. Brown, R. Traylor, and T. Fiez, "Educational design, evaluation, & development of platforms for learning," in *Proc. ASEE Frontiers in Education Conf.*, 2004.

[8] R. Brown, R. Lomax, G. Carichner, and A. Drake, "A microprocessor design project in an introductory VLSI course," *IEEE Trans. Educ.*, vol. 43, pp. 353-361, Aug. 2000.

[9] T. Clausen, "Academic excellence by the Telemark model of cooperative learning," in *Proc. ASEE Frontiers in Education Conf.*, 1997, pp. 57-61.

[10] (2006) LEGO main website. [Online]. Available: http://www.lego.com

[11] M. Jadud, “Teamstorms as a theory of instruction,” Dec. 1999. [Online]. Available: http://www.indiana.edu/~legobots/pdf/teamstorms_theory.pdf

[12] (2006) LEGO Mindstorms website. [Online]. Available: http://mindstorms.lego.com

[13] A. Ertas, T. Maxwell, V. Rainey, and M. Tanik, “Transformation of higher education: the transdisciplinary approach in engineering,” *IEEE Trans. Educ.*, vol. 46, pp. 289-295, May. 2003.

[14] S. Hadjerrouit, “Learner-centered web-based instruction in software engineering,” *IEEE Trans. Educ.*, vol. 48, pp. 99-104, Feb. 2005.

[15] G. Catalano, and K. Catalano, “Transformation: from teacher-centered to student-centered engineering education,” in *Proc. ASEE Frontiers in Education Conf.*, 1997, pp. 95-100.

[16] W. Clark, “Using multimedia and cooperative learning in and out of class,” in *Proc. ASEE Frontiers in Education Conf.*, 1997, pp. 48-52.

[17] N. Sarkar, “Teaching computer networking fundamentals using practical laboratory exercises,” *IEEE Trans. Educ.*, vol. 49, pp. 285-291, May. 2006.

[18] J. Stuart, “A method for teaching problem assessment,” in *Proc. ASEE Frontiers in Education Conf.*, 1997, pp. 83-87.

[19] Xilinx datasheet for “Spartan-3 FPGA family”. Document DS099, Ver. 2.1. Xilinx, Inc. Apr. 2006. [Online] Available: http://www.xilinx.com/bvdocs/publications/ds099.pdf

[20] Xilinx user guide for PicoBlaze 8-bit embedded microcontroller. Document UG129, Ver. 1.1.1. Xilinx, Inc. Nov. 2005. [Online] Available: http://www.xilinx.com/bvdocs/userguides/ug129.pdf

[21] Xilinx reference guide for “MicroBlaze processor”. Document UG081, Ver. 5.3. Xilinx, Inc. Oct. 2005. [Online] Available: http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf

[22] Xilinx datasheet for “Platform flash in-system programmable configuration PROMs”. Document DS123, Ver. 2.9. Xilinx, Inc. May. 2006. [Online] Available: http://www.xilinx.com/bvdocs/publications/ds123.pdf

[23] Texas Instruments datasheet for “Fast transient response, 1-A low-dropout voltage regulators”. Document SLVS211L. Texas Instruments, Inc. Jan. 2006. [Online] Available: http://focus.ti.com/lit/ds/symlink/tps76801.pdf

[24] Linear datasheet for “Low power, 1kHz to 20MHz resistor set SOT-23 oscillator”. Document 6900f. Linear Technology, Corp. Jan. 2006. [Online] Available: http://www.linear.com/pc/downloadDocument.do?navId=H0,C1,C1010,C1096,P2186,D1631

[25] Vishay datasheet for “IR receiver modules for remote control systems”. Document 82229, Ver. 1.2. Vishay Semiconductor, GmbH. Jan. 2005. [Online] Available: http://www.vishay.com/docs/82229/82229.pdf

[26] Hitachi datasheet for “HD44780U (LCD-II) dot matrix liquid crystal display controller/driver”. Document ADE-207-272(Z), Rev. 0.0. Hitachi, Ltd. Sep. 1999. [Online] Available: http://web.media.mit.edu/~ayah/documents/hd44780u.pdf

[27] Micron datasheet for “Synchronous DRAM (64Mb)”. Document 64MSDRAM_1.fm, Rev. L. Micron Technology, Inc. Jun. 2006. [Online] Available: http://download.micron.com/pdf/datasheets/dram/sdram/64MSDRAM.pdf

[28] Atmel datasheet for “16-megabit 2.5-volt only or 2.7-volt only DataFlash®”. Document 2224, Rev. 2224I-DFLSH-10/04. Atmel, Corp. Oct. 2004. [Online] Available: http://www.atmel.com/dyn/resources/prod_documents/doc2224.pdf

[29] M. Nixon, “On a programmable approach to introducing digital design,” *IEEE Trans. Educ.*, vol. 40, pp. 195-206, Aug. 1997.

[30] K. Chapman, “Xilinx PicoBlaze KCPSM3 manual,” Oct. 2003. [Online] Available: http://www.sysf.physto.se/~attila/ATLAS/JEM/JET_FPGA_JEM-1.0/VHDL/Backup_20050201/MC_Software/KCPSM3/KCPSM3_Manual.pdf

[31] Xilinx application note for “PicoBlaze 8-bit microcontroller for Virtex-E and Spartan-II/IIE devices”. Document XAPP213, Ver. 2.1. Xilinx, Inc. Feb. 2003. [Online] Available: http://www.xilinx.com/bvdocs/appnotes/xapp213.pdf

[32] K. Chapman, “Xilinx UART transmitter and receiver macros manual,” Jan. 2003. [Online] Available: http://www.coe.montana.edu/ee/rlarimer/ee467/Lab%207_2006/UART_Manual.pdf

[33] (2006) ModelSim – a comprehensive simulation and debug environment for complex ASIC and FPGA designs. [Online]. Available: http://www.model.com

[34] D. Niggemeyer, K. Stephano, and E. Rudnick, “Use of a field programmable gate array for education in manufacturing test and automatic test equipment,” *IEEE Trans. Educ.*, vol. 44, pp. 239-245, Aug. 2001.

Appendices

USER GUIDE FOR FPGA LEARNING PLATFORM

Version 0.1

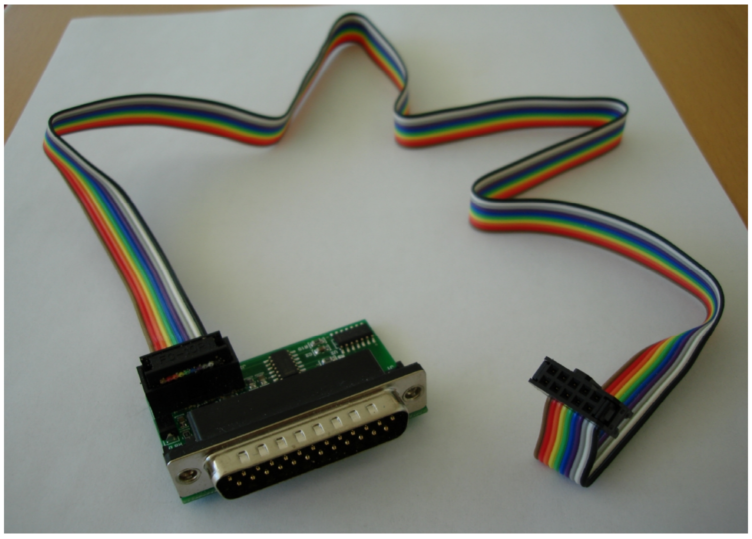
This user guide provides detailed and concise information regarding the features of the FPGA learning platform so that users can quickly become productive with it. Hardware features of the FPGA development board as well as the usage of the software CAD tools are presented.

The user guide also describes a development test suite, known as OMICRON, to test the peripherals of the FPGA board. In addition to testing on-board components, OMICRON can provide a hardware testbench debugger for the purpose of testing FPGA-implemented modules and circuit designs.

Requirements

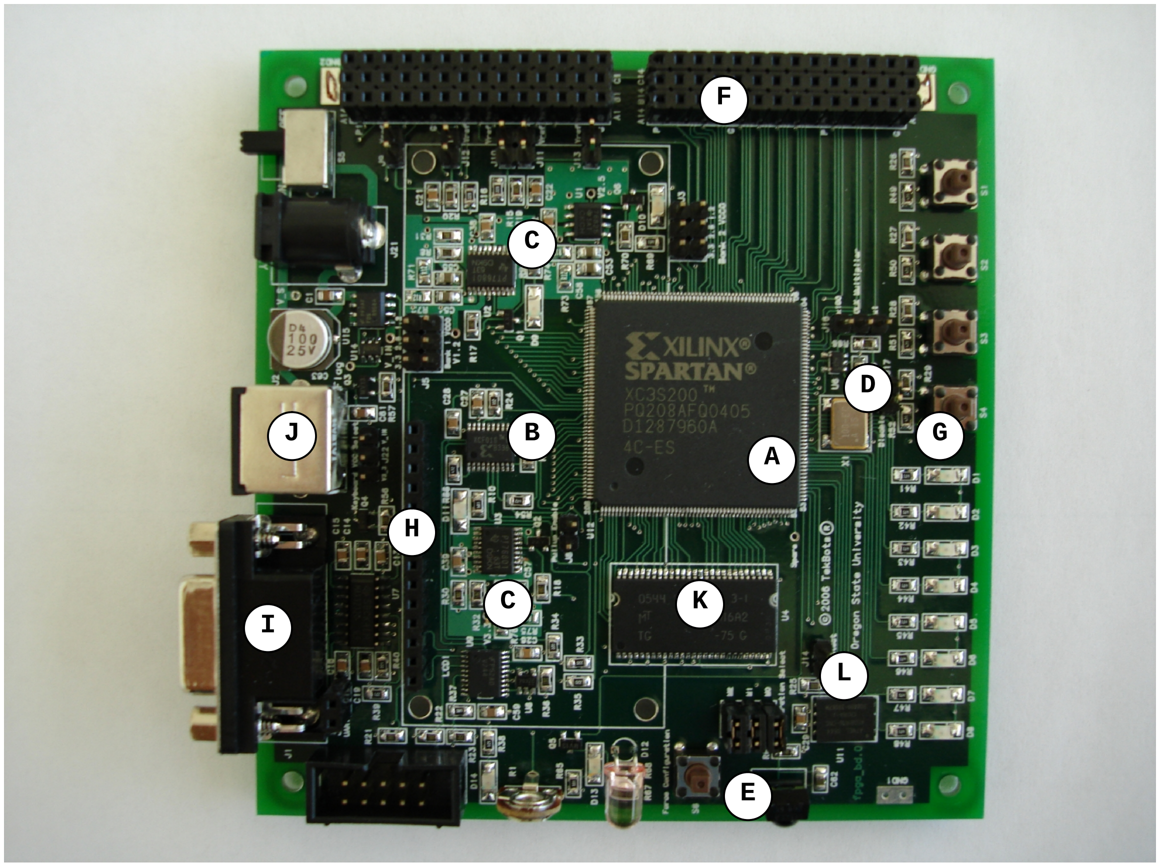
The following are required for development using the FPGA board:

- FPGA development board
- Computer
  - with **Xilinx ISE^{TM}** (version 7.1i or later) installed
  - with **parallel port**
  - preferably running on **Linux**-type OS
  - preferably with Internet access
- Power adaptor
  - 5V supply with approximately 1.5A rating
- TekBots® Xilinx programming dongle (picture shown below)



FPGA development board

In this section, hardware details of the FPGA development board are outlined. The various types of on-board components are described, and the URL links to their datasheets are provided. Pin assignments allow quick reference to see how the components are connected to the FPGA device.



(A) FPGA

- **Component**: Xilinx Spartan-3 XC3S200 [`U12`]
- **Datasheet**: `http://www.xilinx.com/bvdocs/publications/ds099.pdf`
- **Notes**
  - 200K-gate design capacity
  - 246Kbit RAM capacity

(B) Flash PROM

- **Component**: Xilinx XCF01S [`U10`]
- **Datasheet**: `http://www.xilinx.com/bvdocs/publications/ds123.pdf`
- **Notes**
  - Non-volatile storage for FPGA configuration bitstream
  - 1Mbit storage capacity
  - Make sure all jumpers on J7 are connected before programming PROM

(C) Power regulators

- **Components**: Texas Instruments TPS76801Q (PWP) & TPS76825 (D)
- **Datasheet**: `http://focus.ti.com/lit/ds/symlink/tps76801.pdf`
- **Notes**
  - Main supply at 5.0V/1.5A
  - Regulated by TPS76801Q
    - 1.2V/600mA supply [`U2`]
      - FPGA core power
    - 3.3V/600mA supply [`U3`]
      - Power for various on-board chips
      - FPGA I/O driver
  - Regulated by TPS76825
    - 2.5V/100mA supply [`U1`]
      - FPGA PLL & programming power

(D) Oscillators

- **Component**: Crystal oscillator [`X1`]
- **Notes**
  - 100MHz main clock for FPGA

- **Component**: Linear LTC6900 [`U6`]
- **Datasheet**: `http://www.linear.com/pc/downloadDocument.do?navId=H0,C1,C1010,C1096,P2186,D1631`
- **Pin assignment**

LTC6900	FPGA
5(OUT)	P80

- **Notes**
  - Auxiliary oscillator (optional clock source)
  - Frequency settings determined by jumper [`J16`]

Frequency	Divider setting	Pin 4 (DIV) connected to
60.76kHz	100	3.3V
607.6kHz	10	open
6.07MHz	1	ground

(E) <u>IR transceiver</u>

- **Component**: Fairchild QED123 [`D12`]
- **Datasheet**: `http://www.fairchildsemi.com/ds/QE%2FQED122.pdf`
- **Pin assignment**

QED123	FPGA
via gate of BSS138 FET	P78

- **Notes**
  - IR transmitter with pull-down FET [`Q5`]
  - **BSS138 datasheet**: `http://www.fairchildsemi.com/ds/BS/BSS138.pdf`

- **Component**: Vishay TSOP32138 [`U13`]
- **Datasheet**: `http://www.vishay.com/docs/82229/82229.pdf`
- **Pin assignment**

TSOP32138	FPGA
1(Vo)	P77

- **Notes**
  - IR receiver with 38kHz carrier frequency
  - Carrier frequency may be generated by dividing 607.6kHz auxiliary oscillator signal by 16

(F) <u>User I/O</u>

- **Pin assignments** (on next page) [`J19,J20`]
- **Notes**
  - 64 general I/O pins
  - VCCO jumper selects 1.2V, 2.5V, or 3.3V

User I/O	FPGA to	
	Banks 1 and 2	Banks 3 and 4
A1	ground	ground
A2	P133	P90
A3	P138	P95
A4	P141	P100
A5	Bank 2 VCCO [J3] select	3.3V
A6	P147	P107
A7	P150	P111
A8	P182	P115
A9	P178	P119
A10	ground	ground
A11	P171	P124
A12	P167	P128
A13	P162	unconnected
A14	Bank 1 VCCO [J5] select	3.3V
B1	P132	P87
B2	P135	P93
B3	P139	P96
B4	P143	P101
B5	P146	P106
B6	P148	P108
B7	P152	P113
B8	P181	P116
B9	P176	P120
B10	P172	P123
B11	P169	P125
B12	P166	P130
B13	P161	unconnected
B14	P155	unconnected
C1	Bank 2 VCCO [J3] select	3.3V
C2	P137	P94
C3	P140	P97
C4	P144	P102
C5	ground	ground
C6	P149	P109
C7	P154	P114
C8	P180	P117
C9	P175	P122
C10	Bank 1 VCCO [J5] select	3.3V
C11	P168	P126
C12	P165	P131
C13	P156	unconnected
C14	ground	ground

(G) LEDs and pushbuttons

- **Pin assignments**

LED	FPGA
D1	P76
D2	P74
D3	P72
D4	P71
D5	P68
D6	P67
D7	P65
D8	P64

Pushbutton	FPGA
S1	P86
S2	P85
S3	P83
S4	P81

- **Notes**
  - LEDs are active high and pushbuttons are active low

(H) LCD interface

- **Pin assignments** [`LCD1`]

LCD	FPGA
1(gnd)	ground
2(vcc)	5V
3(contrast)	10kΩ pot.
4(rs)	P199
5(r_wn)	ground
6(enable)	P198
7(d0)	P197
8(d1)	P196
9(d2)	P194
10(d3)	P191
11(d4)	P190
12(d5)	P189
13(d6)	P187
14(d7)	P185

- **Notes**
  - Based on Hitachi HD44780 instruction set
  - **HD44780 datasheet**: `http://web.media.mit.edu/~ayah/documents/hd44780u.pdf`

(I) Serial port

- **Pin assignments** [J1]

Serial port	FPGA
3(TXD)	P205
2(RXD)	P204

- **Notes**
  - § DTR (pin 4) connected to DSR (pin 6)
  - § RTS (pin 7) connected to CTS (pin 8)

(J) PS/2 port

- **Pin assignments** [J2]

PS/2 port	FPGA
1(DATA)	P184
5(CLOCK)	P183

- **Notes**
  - § Jumper [J22] select voltage operation at 3.3V or 5.0V

(K) SDRAM

- **Component**: Micron MT48LC4M16A2 [U4]
- **Datasheet**: `http://download.micron.com/pdf/datasheets/dram/sdram/64MSDRAM.pdf`
- **Pin assignments** (on next page)
- **Notes**
  - § 64Mbit storage capacity of volatile memory

SDRAM	FPGA
2(DQ0)	P2
4(DQ1)	P3
5(DQ2)	P4
7(DQ3)	P5
8(DQ4)	P7
10(DQ5)	P9
11(DQ6)	P10
13(DQ7)	P11
42(DQ8)	P21
44(DQ9)	P20
45(DQ10)	P19
47(DQ11)	P18
48(DQ12)	P16
50(DQ13)	P15
51(DQ14)	P13
53(DQ15)	P12
23(A0)	P45
24(A1)	P44
25(A2)	P43

SDRAM	FPGA
26(A3)	P42
29(A4)	P40
30(A5)	P39
31(A6)	P37
32(A7)	P36
33(A8)	P35
34(A9)	P34
22(A10)	P46
35(A11)	P33
20(BA0)	P50
21(BA1)	P48
19(CS_N)	P27
18(RAS_N)	P26
17(CAS_N)	P24
16(WE_N)	P22
37(CLKE)	P31
38(CLK)	P29
15(DQM_L)	P51
39(DQM_H)	P28

(L) EEPROM

- **Component**: Atmel AT45DB161B [U11]
- **Datasheet**: `http://www.atmel.com/dyn/resources/prod_documents/doc2224.pdf`
- **Pin assignments**

EEPROM	FPGA
1(SI)	P57
2(SCLK)	P58
3(RESET_N)	P61
4(CS_N)	P62
8(SO)	P63

- **Notes**
  - 16Mbit storage capacity of non-volatile memory
  - Requires SPI
  - Write protect WP_N enabled by jumper [J14]

FPGA development using Xilinx ISE™

The software used for FPGA development is **Xilinx ISE™**. ISE facilitates the development of FPGA designs by providing various CAD tools that are bundled within a graphical IDE. Shown below is a list of the tools and their descriptions:

- **Project Navigator**
  - Main tool of ISE
  - Houses all other tools by arranging them in order of design flow
  - Used to structure hierarchy of an FPGA design
  - Displays design results and summary
  - Displays RTL and Technology schematics
  - Able to launch **ModelSim™** for simulation
  - Provides text editor to edit source files
- **PACE** & **Constraints Editor**
  - Assign package pins that connect core I/O to external I/O
  - Allows area and timing constraints to be set
- **XST**
  - Used to synthesize FPGA design
  - **Floorplanner** tool used to tweak synthesized designs
  - **XPower** tool used to perform power analysis
- **iMPACT**
  - Generates PROM, ACE or JTAG file
  - Used to program design into FPGA

Shell environment setup on Linux

Before ISE can be used on a Linux workstation, the shell environment needs to be configured properly first. The main objective of this setup is to insert the **windrvr6** module into the Linux kernel. This module enables **iMPACT** to program the FPGA board via the parallel port.

If a precompiled module (with filename `windrvr6.ko`) was not provided, follow these steps below to compile the module:

- Download `http://www.jungo.com/download/WD702LN.tgz` into the home directory.
- Perform the commands below. Replace `<linux_dir>` with the appropriate Linux directory name derived from the `ls` command.

```
$ cd ~
$ mkdir isesetup
$ mv WD702LN.tgz isesetup
$ cd isesetup
$ tar –zxvf WD702LN.tgz
$ cd WinDriver
$ make
$ cd redist
$ ls
$ cp <linux_dir>/windrvr6.ko ~/isesetup
$ cd ~/isesetup
```

- When the commands above are successfully completed, there will be a module file `windrvr6.ko` in the directory `~/isesetup`.

If the precompiled module was provided, copy the module into the directory `~/isesetup`.

Once the file `windrvr6.ko is` in directory `~/isesetup`, perform the commands below *once* with **root** access:

```
$ cd ~/isesetup
$ insmod windrvr6.ko
$ mknod /dev/windrvr6 c $(grep windrvr6 /proc/devices | cut -f 1 -d " ") 0
$ chmod a+rw /dev/windrvr6
```

At this point, the module has been activated for the Linux kernel. You may begin to use iMPACT to program the FPGA board. The `WinDriver` directory may be safely deleted.

More detailed information for the shell environment setup can be found at:

`http://www.gentoo-wiki.com/HOWTO_Xilinx`

Composition of an FPGA design

A basic FPGA design within the ISE environment is composed of

- HDL (e.g., VHDL, Verilog, etc.) source files that define the hardware implementation
- a user constraints file (e.g., `top_level.ucf`) file that defines pin assignments and design constraints

As an example, given the following VHDL entity declaration:

```
entity module_top is
  port(
        signalA    : out  std_logic;
        signalB    : in   std_logic;
        databus    : out  std_logic_vector(3 downto 0)
      );
end entity module_top;
```

the corresponding `module_top.ucf` may look like this:

```
NET "signalA"        LOC = "P204";              # 1-bit signal
NET "signalB"        LOC = "P183" | PULLUP;     # signal with pull-up enabled
NET "databus<0>"     LOC = "P81";
NET "databus<1>"     LOC = "P83";
NET "databus<2>"     LOC = "P85";
NET "databus<3>"     LOC = "P86";                # 4-bit bus
TIMESPEC TS01 = FROM : PADS : TO : RAMS : 10 ns; # timing constraint
```

The user constraints file is read on a line-by-line basis. More settings can be stacked on a line using the pipe “|” symbol. The comment leader “#” works till the end of the line. Any number of spaces and tabs may be used to separate words.

Implementing a sample FPGA design (OMICRON) using ISE

In order to describe how designs are implemented for the FPGA using ISE, a walkthrough of how to implement a sample design is presented. The sample design is the OMICRON development test suite. Details on the usage of OMICRON are provided after this walkthrough.

- **Create a project directory**

  The first step to implementing an FPGA design is to create a project directory to store the design and work files that ISE uses. This is as simple as making an empty directory and placing the source files in there. Even though ISE provides the option of creating the project directory automatically, it is advisable to perform this step manually.

  ```
  $ mkdir omicron
  $ cd omicron
  ```

  Within the project directory, it is best to place the source files into a directory on its own. Create a directory called `src` to store all source files.

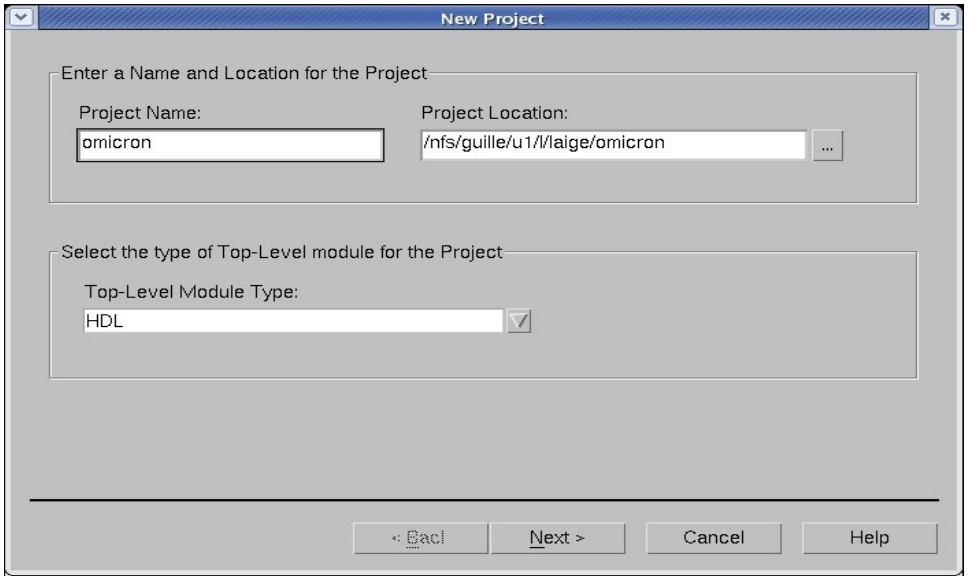
  ```
  $ mkdir src
  $ cp ~/omicron_pkg/* src
  $ cd src
  $ ls

  bbfifo_16x8.vhd  OMI_AUX.VHD      OMI_MAIN.VHD  uart_tx.vhd
  kcpsm3.vhd       omicron_spi.vhd  OMI_TB.VHD
  kcuart_rx.vhd    omicron_top.vhd  system.ucf
  kcuart_tx.vhd    omi_data0.vhd    uart_rx.vhd
  ```

- **Create new project in ISE**

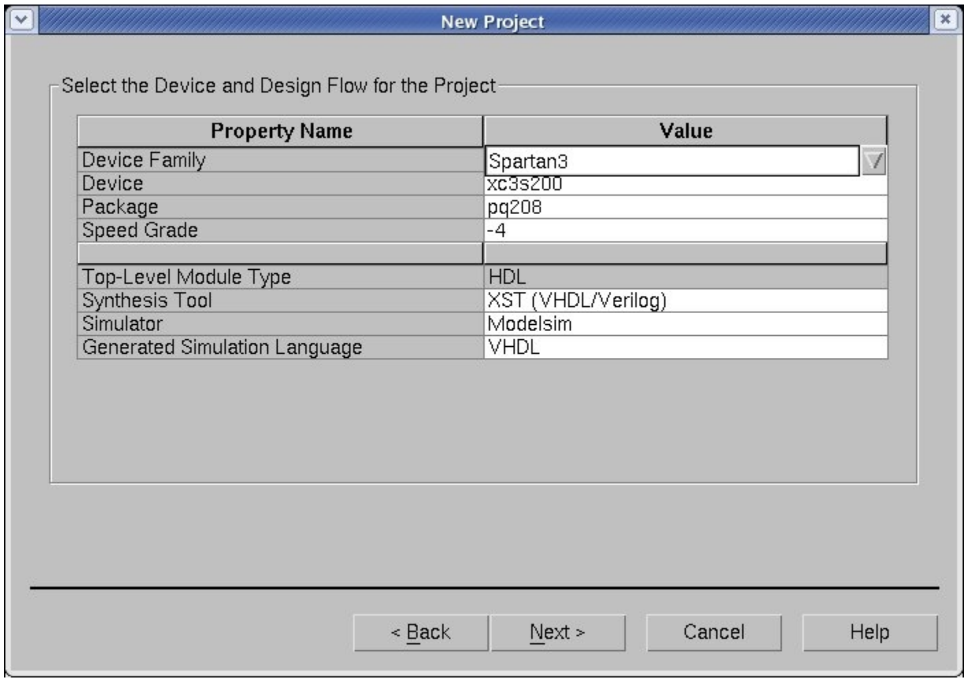
  Once the source files are in place, start ISE with

  ```
  $ ise
  ```

Once ISE is loaded, selecting **File à New Project** begins the project wizard. Enter the Project Name as "omicron", which is the same name of the project directory. Make sure that the Project Location corresponds to the project directory that was made. Top-level module type is selected as "HDL".


New Project
Enter a Name and Location for the Project
Project Name:
omicron
Project Location:
/nfs/guille/u1/l/laige/omicron
...
Select the type of Top-Level module for the Project
Top-Level Module Type:
HDL
< Back
Next >
Cancel
Help

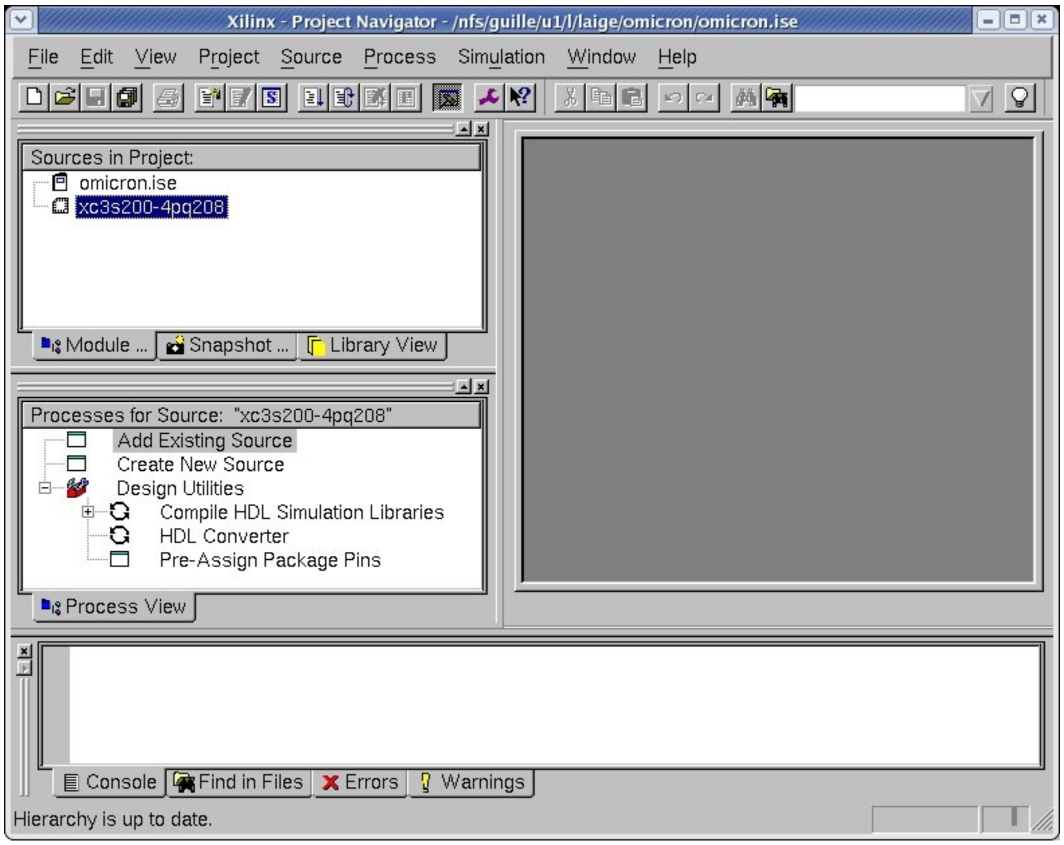
Clicking **Next** brings up the Device and Design Flow options. Select them as shown in the figure below. After this, click **Next** three times until the wizard offers to **Finish**, thus skipping the next few windows. Review the summary and click on **Finish** to end the wizard. Source files will be added later.



Property Name	Value
Device Family	Spartan3
Device	xc3s200
Package	pq208
Speed Grade	-4
Top-Level Module Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	Modelsim
Generated Simulation Language	VHDL

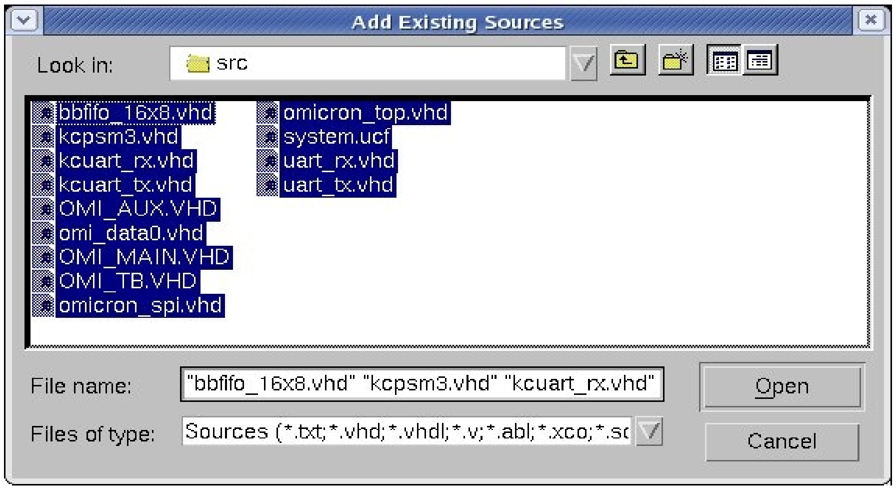
v Add source files

Once a new project is created, the Project Navigator window will look like this:



Hierarchy is up to date.

Proceed to add source files to the project by right-clicking on the *Module View* (top left sub-window with highlighted item) and selecting **Add Source**. Then, navigate to the `src` directory and select all design files to be added. This can be achieved with <Ctrl + A>, or <Ctrl/Shift + Left mouse-click> combinations.

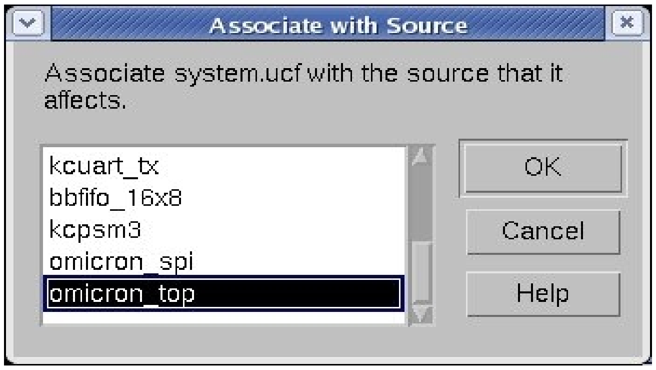


bbfifo_16x8.vhd
kcpsm3.vhd
kcuart_rx.vhd
kcuart_tx.vhd
OMI_AUX.VHD
omi_data0.vhd
OMI_MAIN.VHD
OMI_TB.VHD
omicron_spi.vhd
omicron_top.vhd
system.ucf
uart_rx.vhd
uart_tx.vhd

File name: "bbfifo_16x8.vhd" "kcpsm3.vhd" "kcuart_rx.vhd"

Files of type: Sources (*.txt;*.vhd;*.vhdl;*.v;*.abl;*.xco;*.sc

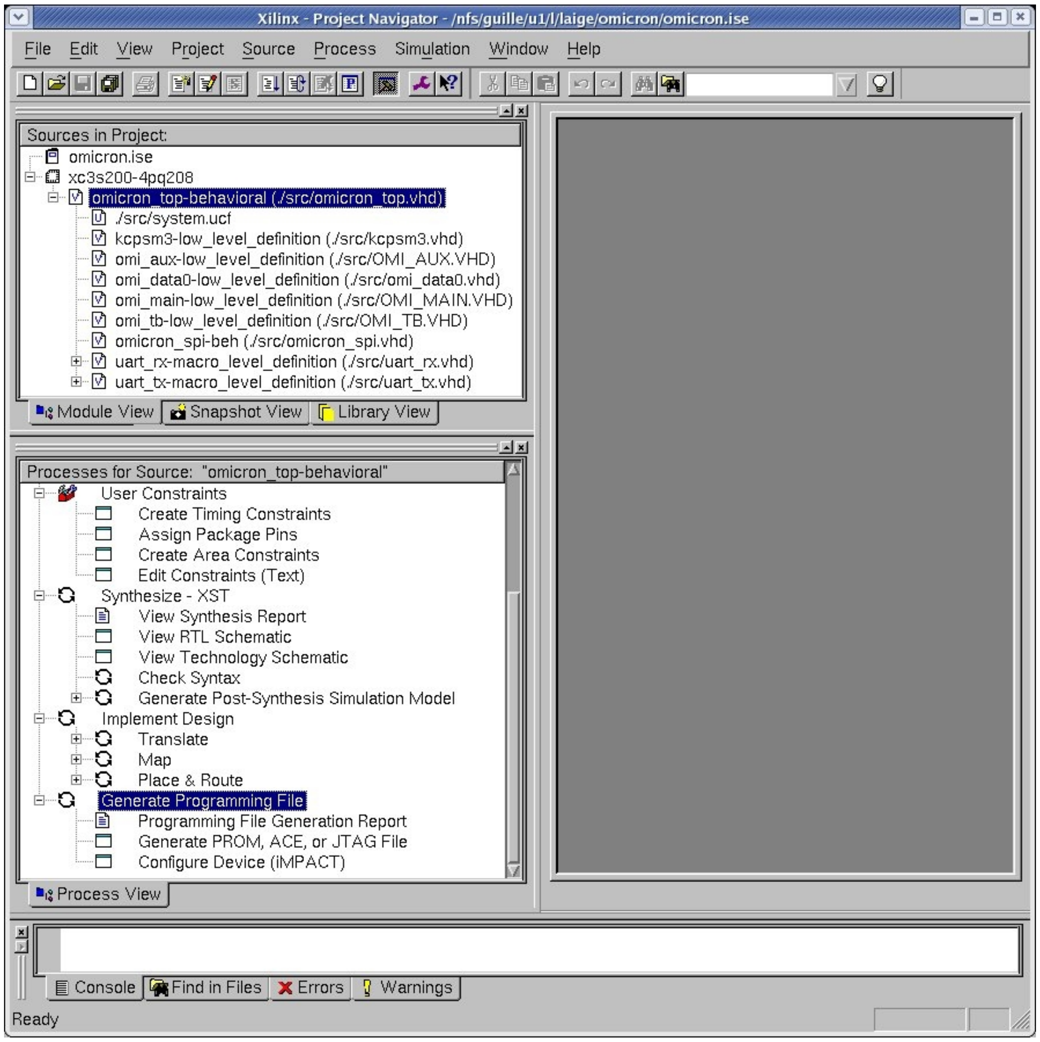
Click on **Open**. Then, for every prompt, confirm that the source files are “VHDL Design” files.



Associate system.ucf with the source that it affects.

kcuart_tx
bbfifo_16x8
kcpsm3
omicron_spi
omicron_top

Associate `system.ucf` with omicron_top. Once the design files are added to the project, the Project Navigator window will look like this:



Sources in Project:
omicron.ise
xc3s200-4pq208
omicron_top-behavioral (./src/omicron_top.vhd)
./src/system.ucf
kcpsm3-low_level_definition (./src/kcpsm3.vhd)
omi_aux-low_level_definition (./src/OMI_AUX.VHD)
omi_data0-low_level_definition (./src/omi_data0.vhd)
omi_main-low_level_definition (./src/OMI_MAIN.VHD)
omi_tb-low_level_definition (./src/OMI_TB.VHD)
omicron_spi-beh (./src/omicron_spi.vhd)
uart_rx-macro_level_definition (./src/uart_rx.vhd)
uart_tx-macro_level_definition (./src/uart_tx.vhd)

Processes for Source: "omicron_top-behavioral"
User Constraints
Create Timing Constraints
Assign Package Pins
Create Area Constraints
Edit Constraints (Text)
Synthesize - XST
View Synthesis Report
View RTL Schematic
View Technology Schematic
Check Syntax
Generate Post-Synthesis Simulation Model
Implement Design
Translate
Map
Place & Route
Generate Programming File
Programming File Generation Report
Generate PROM, ACE, or JTAG File
Configure Device (iMPACT)

- **Synthesize & Implement**

  The goal of synthesis and implementation is to generate a programming file, known as a configuration bitstream, which is downloaded into the FPGA. Synthesis involves compiling the HDL sources into a low-level form that can be translated and mapped into the actual FPGA hardware.

  To begin the implementation process, right-click on **Generate Programming File** in the *Process View* (bottom left sub-window with highlighted item) and click on **Rerun**. ISE will begin the entire design flow process from the top. During development, it is advisable to use **Rerun** instead of **Run** to ensure that design changes made anywhere in the flow are always committed.

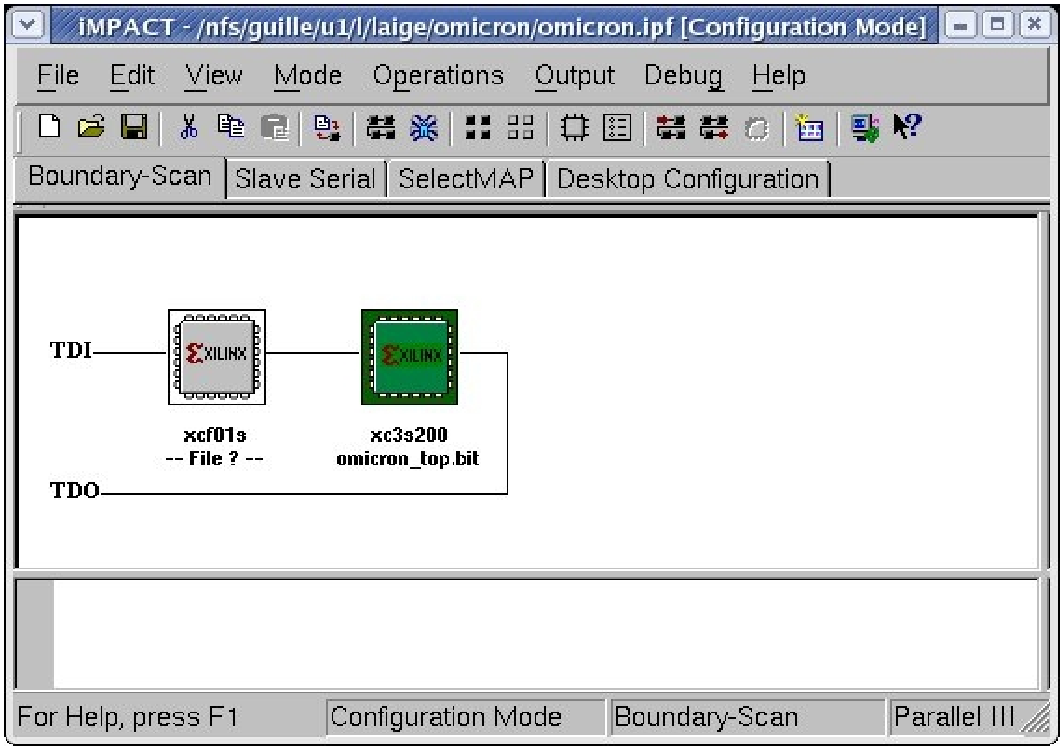
  Errors and warnings can be viewed by clicking on their respective tabs in the *Transcript View* (bottommost sub-window). If an error is encountered, the design process will halt. Warnings do not halt the process. It is advisable to fix as many warnings as possible. For OMICRON, there are unavoidable warnings that can be safely ignored.

- **Programming the FPGA**

  Once the design is successfully synthesized, a bitstream file `omicron_top.bit` will be generated. This is the file that needs to be downloaded into the FPGA.

  To achieve this, first make sure the programming dongle is connected to the FPGA board. Then, power up the board. Start iMPACT by right-clicking on **Configure Device (iMPACT)** and selecting **Open Without Updating**. Once loaded, iMPACT will bring up a wizard. Select **Boundary-Scan Mode** to configure the device and let it **Automatically connect to cable and identify Boundary-Scan chain**.

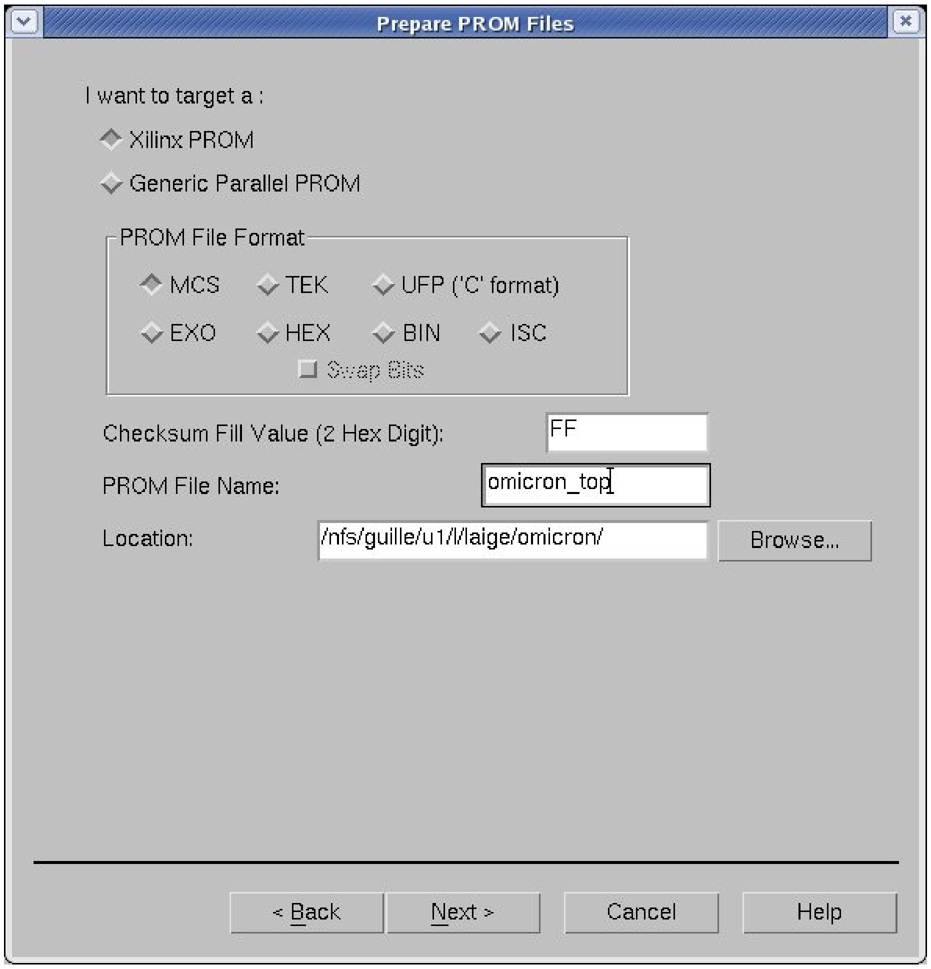
Two devices will be detected in the scan chain: (from TDI) the flash PROM **xcf01s** followed by the FPGA device **xc3s200**. For the PROM, click on **Cancel** (the PROM file can be generated later). For the FPGA, choose `omicron_top.bit` as the configuration file. The window will look like this:



The FPGA is now ready to be programmed! Right click on the FPGA device and select **Program**. Then, click **OK** to begin programming. Once finished, iMPACT will state that the **Programming Succeeded**.

A bitstream that is downloaded into the FPGA is volatile. Upon a power cycle, the bitstream is destroyed and the FPGA needs to be programmed again. To make the configuration permanent, the bitstream can be stored in the PROM so that the PROM injects the FPGA with the bitstream when the board is powered up.

Before a bitstream file is stored into the PROM, it needs to be converted into a PROM file. To generate a PROM file, go to **Mode à File Mode**. Then click on the **PROM Formatter** tab. Right-click in the window and select **Launch Wizard**. Enter the **PROM File Name** as "omicron_top". Leave the other options as shown below.



I want to target a :

[✓ Xilinx PROM]

Generic Parallel PROM

PROM File Format

[✓ MCS] TEK UFP ('C' format)

EXO HEX BIN ISC

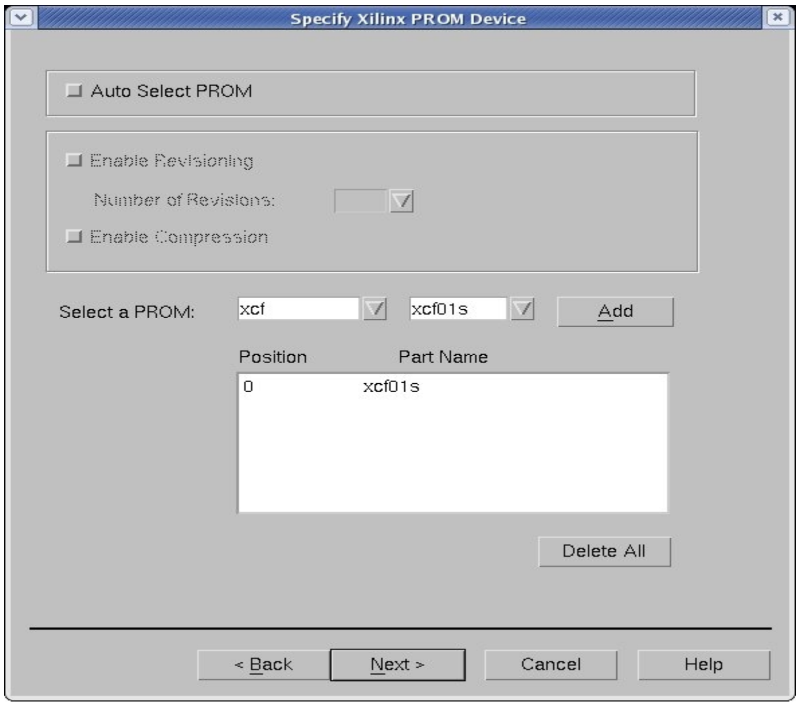
Swap Bits

Checksum Fill Value (2 Hex Digit): FF

PROM File Name: omicron_top

Location: /nfs/guille/u1/l/laige/omicron/

Click **Next**.

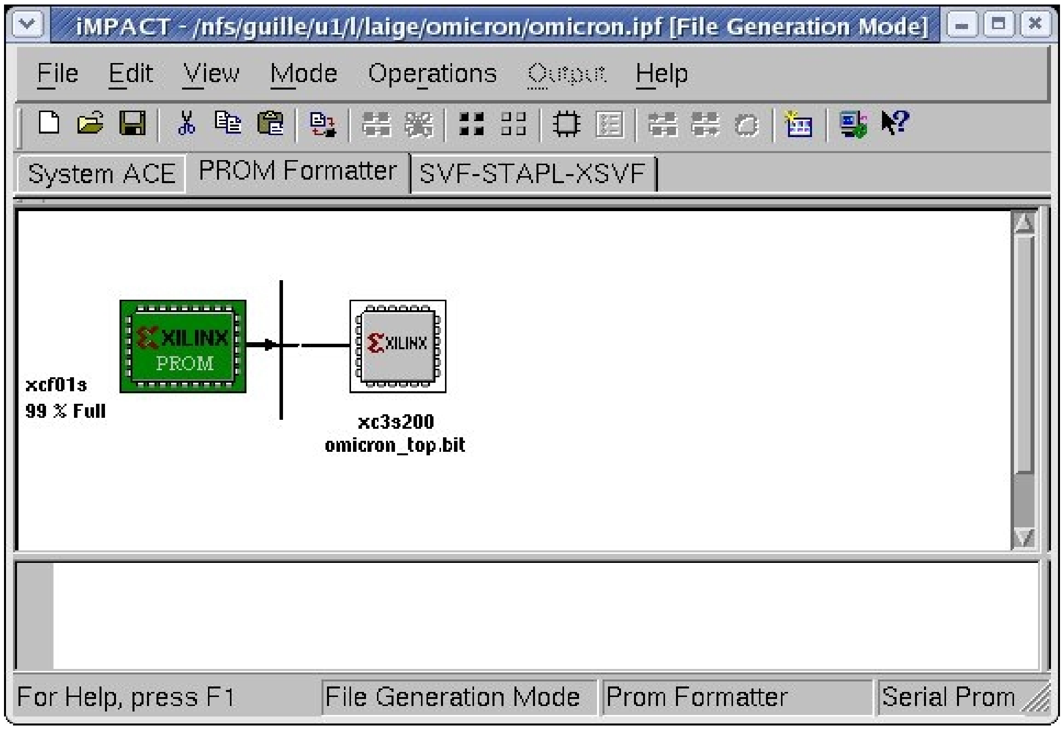


Select a PROM: xcf xcf01s

Position	Part Name
0	xcf01s

Click **Add** to select the PROM as **xcf01s**. Then, click **Next** twice.

In the *Add Device File* window, click on **Add File** and select `omicron_top.bit` as the file to be converted. Click **Finish** and select **Yes** to generate the file now. The iMPACT window will then look like this:



At this point, the PROM file `omicron_top.mcs` has been generated.

Switch back to *Configuration Mode* by selecting **Mode à Configuration Mode**. Right-click on the PROM and **Assign New Configuration File** as `omicron_top.mcs`.

Finally, follow the **same steps** to program the FPGA in order to program the PROM.

The provided walkthrough is by no means a complete guide. Many features of the tools are not explained, and the usage of a few tools (such as PACE and Floorplanner) have been left out. Please refer to the Xilinx documentation for more help on these tools.

Also, do not be afraid to experiment with different options. If an irrecoverable mistake was made, simply create a new project directory by copying over the source files to a new directory. Then, the original directory may be safely deleted.

Troubleshooting

The following are viable solutions to common errors that may occur:

v When programming the FPGA device, the following errors occur:

ERROR:Bitstream:2 – The input file "…msk" does not exist. Please check that the specified location is correct and that the bitstream was successfully created.

ERROR:iMPACT:123 – Mask file "…msk" is invalid.

ü Make sure the **Verify** radio button is not selected before clicking on **OK** to program the FPGA.

v When trying to program a device, iMPACT crashes quietly. Upon reloading iMPACT, the boundary scan chain cannot be detected because the parallel port is locked by the original iMPACT process.

ü Run the commands below in a non-root shell to clean up the lock. Note that the directory where iMPACT resides may be different.

```
$ cd /nfs/guille/a2/rh80apps/xilinx/8.1i/bin/lin
$ impact -batch
> setmode -bs
> cleancablelock
> setcable –p auto
> quit
```

Testing the FPGA board with OMICRON

OMICRON provides a test suite that comprehensively tests the FPGA development board. Once it is loaded into the FPGA, the board is able to perform self-tests to check the functionality of various on-board components.

Test requirements

These equipments are required for testing the FPGA board using OMICRON:

- Serial cable
- Computer
  - with **RS232-capable terminal program** installed (e.g., minicom, HyperTerminal)
  - able to perform communications via **serial port**
- LCD with 14-pin HD44780 interface
  - at least 2 lines with 16 characters
- User I/O feedback connector
  - provides feedback connections for user I/O test
- PS/2 port loopback wire
  - connects data and clock lines to form feedback
  - any hard breadboard-type wire will do
- Sheet of paper
  - provides reflective surface for IR signal feedback

The peripheral tests

Detailed below are the tests that OMICRON can perform:

- **Serial port test**
  - Uses serial communication to RS232-capable terminal
  - Transmit test: ASCII string sent to terminal
  - Receive test: Listen for serial transmission and display incoming bytes
    - Bytes generated from keyboard
    - LEDs flash alternately as additional indication

- **LCD test**
  - ASCII test: Display range of ASCII characters
  - Block test: Display block characters (to detect dead pixels)
  - Can use **Serial port test** to display custom message

- **LEDs test**
  - Test: Binary counter to flash LEDs

- **Pushbuttons test**
  - Test: Display which button pushed
  - LEDs toggle accordingly also

- **IR transceiver test**
  - Test: Transmit IR signal in pulses of 16 periods with 37.5% duty cycle; then verify received signal at critical points
  - Avoid testing under direct fluorescent light
  - Use sheet of paper to reflect IR signal

- **User I/O test**
  - Group 64 I/O pins into 2 virtual ports of 32 pins
  - Use feedback connector to connect both virtual ports in pin-wise pairs
  - Test: Virtual ports take turns to be either input (with pull-up enabled) or output; output port attempts to drive input port low
  - Successful drive to low both ways passes test

- **PS/2 port test**
  - Test: Same as **User I/O test** except only 2 pins (data & clock) are bidirectionally tested

- **SDRAM test**
  - Write data and read it back for test verification
  - If verification fails, display data value and address location
  - Data bus test: Fix memory address at 0
    Data lines verified with walking-ones/zeroes
  - Address bus test: Exhaustive verification on every memory address location using pseudo-random data

- **EEPROM test**
  - Same **Data bus test** as in **SDRAM test**
  - Address bus test: Select memory address location with walking-ones/zeroes
    Data lines verified with pseudo-random data

Interactive modes of operation

In order to perform peripheral tests on the FPGA board, test commands are issued by the user to OMICRON to be acted upon. Test results are then displayed either on the LCD or terminal screen depending on the type of operation mode.

Two modes of operation are provided for the user to interact with OMICRON:

- **Terminal (`TERM`) mode**
  - Utilizes serial cable connection on terminal program for user interactivity
    - Connect serial cable between serial ports of computer and FPGA board
    - Set for following communication settings
      - 38400 baud rate
      - 8 data bits
      - No parity bit
      - 1 stop bit
  - Menu-driven command line interface
    - Terminal menu hierarchy:

```
TOP MENU
┃
┣━ L: LED TEST
┣━ B: BUTTON TEST
┣━ U: USER I/O TEST
┣━ P: PS/2 TEST
┣━ I: IR TEST
┃
┣━ D: DISPLAY MENU
┃   ┃
┃   ┣━ C: CLEAR LCD
┃   ┣━ A: LCD ASCII
┃   ┣━ B: LCD BLOCKS
┃   ┣━ I: SERIAL INPUT
┃   ┗━ O: SERIAL OUTPUT
┃
┣━ M: MEMORY MENU
┃   ┃
┃   ┣━ S: SDRAM TEST
┃   ┗━ E: EEPROM TEST
┃
┗━ T: TESTBENCH MENU
    ┃
    ┣━ V: LOAD VECTORS
    ┣━ F: LOAD CUSTOM
    ┣━ A: AND R
    ┣━ O: OR R
    ┣━ X: XOR R
    ┣━ S: STATUS
    ┣━ R: PORT TO R
    ┣━ N: SET NEXT OUTPUT
    ┗━ C: CLK OUTPUT
```

    - To view current menu and its commands, issue "?" as the command
    - To go back a menu, issue "G" as the command

- **Non-terminal (`NOTERM`) mode**
  - Should serial port fail, OMICRON can fall back on NOTERM mode
    - Restrictive interactivity – no TESTBENCH menu commands
  - Access NOTERM mode by pressing pushbuttons 3 times
  - Controlled by pushbuttons
    - S4 to move option backwards
    - S3 to move option forward
    - S1 to select/cancel option
  - Non-terminal options:

```
0000  ENTER TERM MODE
0001  LED TEST
0010  BUTTON TEST
0011  USER I/O TEST
0100  PS/2 TEST
0101  IR TEST
0110  LCD ASCII
0111  LCD BLOCKS
1000  SERIAL INPUT
1001  SERIAL OUTPUT
1010  SDRAM TEST
1011  EEPROM TEST
```

  - LEDs indicate options when LCD is faulty
    - D4-D1 light up according to binary codes shown above
    - D8 lights up to show successful test
    - D7 lights up to show failed test
  - Can co-exist with TERM mode
    - OMICRON continues to send verbose results via serial port

§ OMICRON operating in NOTERM mode: