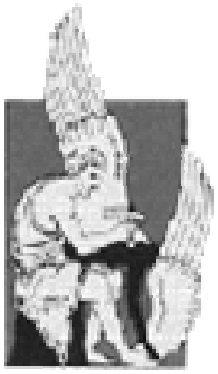

Technical University of Crete

Department of Electronic and Computer Engineering

Diploma Thesis

Development of a language and Universal Run-Time Environment for FPGA programming

Kyriakidis Thomas

Supervising Professor : Professor Apostolos Dollas

Thesis Committee : Associate Professor Kostantinos Kalaitzakis
Assistant Professor Dionysios Pnevmatikatos

March 2002

Αφιερώνεται στους γονείς μου και την οικογένειά μου...
Dedicated to my parents and my family…

Thomas Kyriakidis

I would like to thank

Professor Apostolos Dollas for his support throughout the whole duration of this thesis and for giving me the opportunity to be a member of Microprocessor and Hardware Laboratory (MHL).

The thesis committee, Associate Professor Kostantinos Kalaitzakis and Assistant Professor Dionisios Pnevmatikatos for their contribution to this thesis.

Markos Kimionis, member of the Technical Staff of the MHL for his support regarding technical matters.

Associate Professor Manolis Antonidakis for assisting us by lending a logic analyzer.

My friend and co-worker in this thesis Dionissis Efstathiou for his excellent work and co-operation that resulted in the success of the system.

All the undergraduate and graduate students for their valuable help.

And last, but not least, I would like to thank my parents and my friends for being there to support me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chapter 1

Introduction

Introduction

Reconfigurable logic is one of the most rapidly growing sectors of the semiconductor industry. FPGAs are becoming an important implementation technology, as the need for quick production of products with relatively fluid specifications is becoming more urgent. Time to market has become a very important factor. To shorten this time it important to shorten the validation and debugging times.

Design Simulation provides complete observability and controllability, but on the other hand execution times are very slow. Millions of simulated cycles can take hours or even days to run on a computer. This means longer verification and debugging times. Finally, simulation does not guarantee that the designs will operate as desired when they are loaded on an FPGA that is on a board because the simulated cases are by necessity limited.

Development boards have proven to be a reliable solution in such cases. They usually consist of a board with one or more FPGA of the same vendor and family and a User Interface, or implemented classes of a language (C/C++ or Java [1]). This solves many problems, but creates other. The developer is limited to testing his design on the board and on the specific FPGA that is on it. To test it on his own board, the developer has to buy a programmer. Programmers are usually vendor specific and quite expensive.

The purpose of this thesis is to develop a language and a Run-Time Environment for FPGA programming, that are vendor-independent. These along with a board containing an Atmel AVR [2] microprocessor provide a very good solution to all the problems mentioned above.

The language is used to create scripts that give the programmer essential information on how to perform the configuration or that can be used for test purposes.

A Graphical User Interface has been developed for writing, compiling and sending scripts to the programmer. The programmer parses the script, takes the necessary steps to execute it. This includes configuration, readback and testing.

The Run-Time Environment enables the programming of an FPGA by providing the necessary script or using one of those already created and giving the configuration bitstream file. The user can also create scripts that will be executed by the hardware for debugging purposes.

This thesis is divided in seven chapters and the appendix.

The second chapter discusses matters concerning FPGA configuration and related work. More specifically, the chapter contains explanations of the configuration modes, configuration pins and bitstreams. A reference is made to the programming of Xilinx’s [3] Spartan-II [4] and Virtex [5]

devices. And finally the PCI Pamette v1 [6], Xilinx's JBits API [7], Xilinx's MultiLinx Download Cable[8] and Altera's [9] MasterBlaster Serial/USB Communications Cable [10] are discussed.

The third chapter describes the architecture of the **Re**configurable logic **Run**-time environment (ReRun). The first two sections contain a brief description as well as the features and attributes. In the third section the structure and operating instructions are discussed. The fourth presents the instructions of the language, while the fifth describes the back-end.

The fourth chapter is dedicated to the language. The first two sections detail the lexical and syntactical analyzers. The third describes the contents of the symbol table. And finally, the fourth refers to the integrity checks and error messages.

The fifth chapter analyzes the Graphical User Interface developed. The first two sections discuss the classes created and the communication protocol. Finally, the third contains usage information for the GUI.

The sixth section is dedicated to presenting complete examples of PTL code. The three first scripts were used for programming FPGAs. The first is for a Xilinx XC3000, the second for a Xilinx XC4000 and the third for an Altera Flex 8000. The fourth example is a test script.

Finally, the seventh chapter contains conclusions and future work.

There are three Appendixes. The first contains a graphical representation of the language grammar. The second provides instructions on how to install ReRun and the third analyzes the instruction opcodes created by the ReRun compiler.

Chapter 2

FPGA Configuration and Related Work

Chapter 2

FPGA/CPLD Configuration Guidelines

This chapter contains information about the configuration process of FPGAs. The first section is a brief introduction to configuration. The second describes the configuration process in more detail. The third section analyzes the various configuration modes. The fourth contains the configuration pins and their description. Next, the fifth analyzes the configuration bitstreams and files. The sixth section refers to the configuration of Xilinx's Spartan-II and Virtex devices. The next four sections contain related work. Section seven describes the PCI Pammete v1. Section eight presents the Xilinx JBits API. The ninth and tenth sections describe Xilinx's MultiLinx Download Cable and MasterBlaster Serial/USB Communications Cable, respectively. Finally, the eleventh section contains a summary of this chapter.

2.1 Introduction

Configuration is the process of loading design-specific programming data into one or more FPGAs/CPLDs to define the functional operation of the internal blocks and their interconnections. An SRAM-based FPGA can be configured on its power-up or even on demand, depending on the architecture of the device. The reason we have to do this is because their configuration memory is generally volatile. That means that they lose their configuration if the power is turned off. The EEPROM based CPLDs can be programmed on demand and they keep their configuration data even after power-off. After configuration the device resets its registers, enables its I/O pins and begins normal operation as a logic device. This is called *User Mode*.

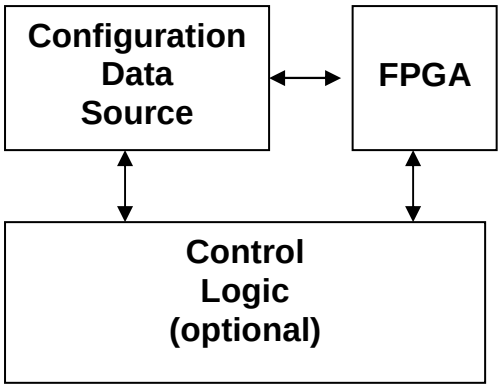

Figure 2.1 - A general block diagram for the configuration of an FPGA/CPLD

2.2 Configuration Process

Generally the configuration process is partitioned into several stages. Each stage is responsible for a specific task. For example the configuration starts with the device power-up which coincides with memory initialization in some SRAM based devices. Then the device enters programming mode, by activating the appropriate signals. The first stage of configuration, Configuration Memory Clear, is unpublished for Altera devices, while for Xilinx and Lucent [11] devices it is the stage in which the configuration memory is cleared. After that, the configuration data is loaded serially or in parallel. In the final stage, the device resets its registers, enables its I/O pins and begins operation as a logic device.
The Altera Company does not provide such information about its devices. The configuration process stages can be generally described as Power-Up → Configuration → Initialization → User Mode.
The configuration process for Xilinx devices consists of four stages:

- ✓ Configuration Memory Clear
- ✓ Initialization
- ✓ Load Configuration data
- ✓ Start-up

The full process for Xilinx devices is illustrated in Fig. 2.2. The first stage is *Configuration Memory Clear*. An internal circuit initializes the configuration logic. Then the V_{CC} reaches an operational level. When that is done a time delay occurs and during this delay the FPGA memory is cleared. During the second stage, *Initialization*, the initialization pin is released and the mode pins are sampled. *Loading Configuration Data*, which is the third stage, loads the device with the configuration bitstream. And finally, the fourth stage, *Start-Up*, prepares the device for normal operation. It releases or activates the configuration control signals and then the FPGA is active and functional with the loaded design.

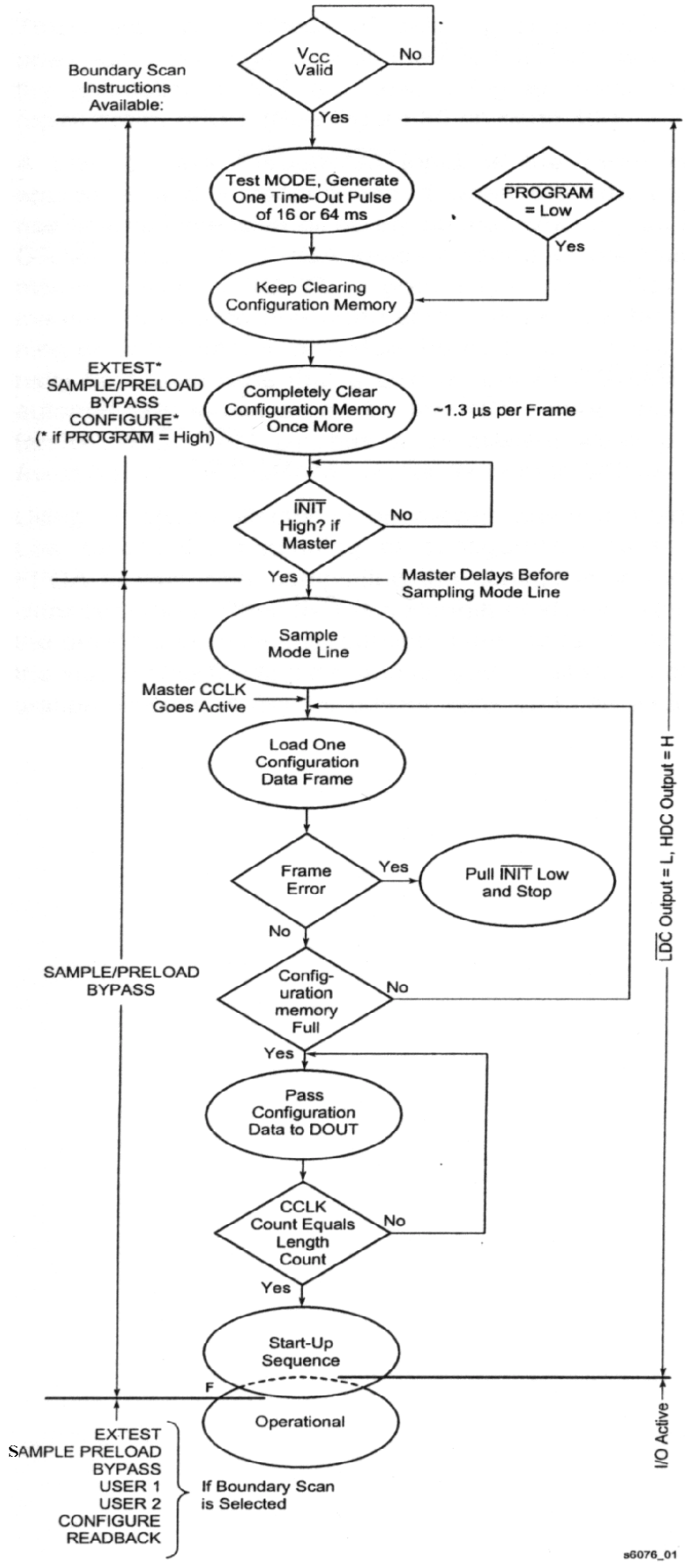

Figure 2.2 - Xilinx Configuration Process [9]

2.3 Configuration Modes

FPGAs/CPLDs can be configured using several schemes. Some modes configure the device using serial configuration data, while others use parallel. On many occasions the FPGA/CPLD produces by itself the control signals needed while in other modes these signals must be provided by external circuitry. When choosing a configuration mode, we must first consider the speed and configuration resources factors. Then we can choose a scheme that is supported by the device, as a device cannot always be configured in all modes. This section categorizes the configuration modes as **Master[12]/Active[13]**, **Slave[12]/Passive[13]** and **JTAG[12][13]**. The names Master and Slave are used by Xilinx and Lucent, while Active and Passive are used by Altera. The two first modes will be mentioned as Master and Slave for simplicity.

2.3.1 Master Modes

In Master configuration modes the device controls the entire configuration process and generates the synchronization and control signals necessary to configure and initialize itself from an external memory. These configuration modes can be used when fast time-to-market is an important factor in our design. They are easy and quick to implement and they require no external intelligence. A device in Master mode can be used in a daisy-chain to configure slave devices by providing the control signals. Finally, a master mode is ideal for automatic configuration at system power-up in most SRAM-based devices, although during an erroneous situation an external circuit must be present to issue reconfiguration. The Master modes are Master Serial and Master Parallel Up/Low and Down/High, which will be explained in the following sections.

2.3.1.1 Master Serial Mode

Master Serial Mode is supported by all FPGAs. As the name implies this mode uses a serial bitstream as a data source and data is loaded at a rate of 1 bit per configuration clock. Whether the MSB or LSB of each data byte is always written first to the data pin, depends on the manufacturing company. The configuration clock pin, which is driven by the target device (FPGA), clocks the sequential data bits from the configuration bitstream into the data pin. Since the target device is the one that controls the entire programming process, the

bitstream is typically stored in an EPROM. Figure 2.3 displays a block diagram for Master Serial Modes.

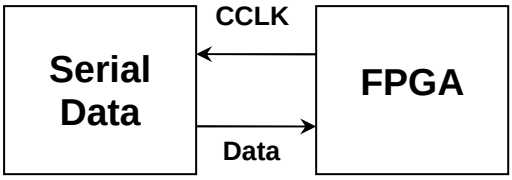

Figure 2.3 - Master Serial FPGA Configuration

2.3.1.2 Master Parallel Up/Low and Down/High Modes

In these modes the target device generates sequential addresses that drive the address inputs of an external PROM. The PROM then returns the byte-wide data to the configuration data input pins of the device. The HEX starting address is 00..0h and increases to a limit for Master Up mode, and it is xx..xh and decrements for Master Down. The limit varies depending on the device. This way they provide address compatibility for microprocessors which begin execution from opposite ends of memory. The device generates addresses until the pin indicating configuration completion, is released. The parallel modes simply activate an internal parallel-to-serial converter and then use the serial bitstream internally. In this mode the RDCLK/RCLK, a clock signal that is generated by dividing the configuration clock signal by eight (f_{RDCLK}= 8*f_{DCLK}), is used to frame the data bytes supplied by the external PROM. In both modes the configuration clock is generated internally and is used to serialize the incoming data bytes. On each pulse of the RDCLK (RCLK in Xilinx) signal, the byte is latched and the following 8 pulses on the configuration clock convert the 8-bit value into a serial data stream. The address generation starts when the signal used to indicate the status of the configuration process (i.e. DONE for Xilinx and CONF_DONE for Altera) is de-asserted.

2.3.2 Slave Modes

Slave Modes use external control logic to generate the configuration clock and allow Daisy-Chain configurations. It allows the FPGA to be configured using other logic devices such as microprocessors, or in a daisy-chain. The device is incorporated into a system with an intelligent host that controls the configuration process. The intelligent host transparently selects a serial or parallel data source and the data is presented to the device on a common data bus. Such systems can store the configuration data on a mass-storage device, such as a hard disk. This way, installing new configuration data becomes easier and the number of Integrated Circuits (ICs) required for a system is reduced. The two slave modes are Serial and Parallel:

2.3.2.1 Slave Serial Mode

Slave Serial Mode is supported by essentially all devices. It places, like all serial modes, the device configuration data at a rate of 1 bit at a time on the configuration data pin of the target device. Depending on the manufacturing company, either the LSB or the MSB is presented first. After all the data has been transferred, the configuration clock must be clocked a few additional times to initialize the device.

2.3.2.2 Slave Parallel /Express/SelectMAP Mode

This Mode is similar to Slave Serial, but the configuration data is loaded at a rate of 1 byte per configuration clock. Each byte is then serialized as described earlier in Master Parallel Mode. Slave Parallel Mode is used when speed is a factor. The Slave Parallel Mode differs from Peripheral Parallel Modes in that devices in this configuration scheme can not be serially daisy chained. At this point, it must be noted that Altera devices do not have a “pure” Slave Parallel Mode. Instead there are the Passive Parallel Synchronous (PPS) and Asynchronous (PPA) Modes. Due to the fact that these modes use an intelligent host (i.e. microprocessor) to control the configuration process they are considered Peripheral Modes and will be described in the Peripheral Modes Section. This is a convention made to categorize the configuration modes efficiently.

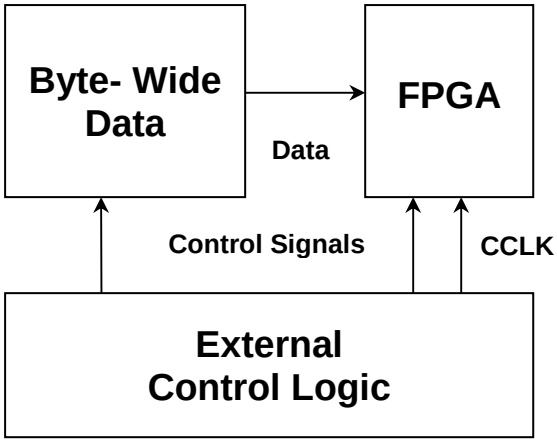

Figure 2.4 - Slave Parallel/ SelectMAP Configuration Mode

2.3.3 Peripheral Modes

Peripheral modes provide a simplified interface through which the device may be loaded bit or byte-wide, as a processor peripheral. Processor write cycles are decoded by controlling the Write Strobe and the Chip Select pins. These modes provide a pin indicating whether the target device is ready to receive the next bit or byte of configuration data. As with Master modes, Peripheral mode may also be used as a lead device for a daisy-chain of slave devices.

2.3.3.1 Serial Asynchronous (Altera Specific)

The microprocessor places a configuration bit on the configuration data input pin and uses the Write Strobe (WS) signal to write data to the device. On the next rising edge of the WS the device latches a bit of configuration data. Subsequently, the device drives the Ready/Busy signal to the appropriate level, indicating that it is processing the configuration data. The microprocessor can then perform other system functions while the device is processing the data bit. It can also monitor other control signals in order to send the next data bit, start initialization stage or restart configuration. An optional address decoder can control the device's Chip Select (CS) pins. This decoder allows the microprocessor to select the device by accessing a particular address, simplifying the configuration process. The microprocessor can control the CS signals directly. The device can process data internally

without the microprocessor. When it is ready to receive the next bit of configuration data, it inverts the Ready/Busy, causing the microprocessor to strobe it into the device.

2.3.3.2 Parallel Asynchronous

Parallel Asynchronous schemes are similar to Peripheral Serial Asynchronous. As the names imply, their main difference is that configuration data is loaded at a rate of one byte, instead of bit, at a time. In this mode the FPGA's internal oscillator generates a configuration clock burst signal used to time the byte-wide data. Asynchronous Mode uses the trailing edge of the logic AND condition of WS and one of the CS signals, as well as the AND of the Read Strobe (RS) and another of the CS signals to accept the data from a microprocessor bus. The Ready/Busy signal is inverted when a byte has been received and returns to its former level again when the byte-wide input buffer has transferred its information into the shift register and is ready to receive new data. This mode allows the RS signal to be strobed, causing the Ready/Busy signal to appear on one of the configuration data inputs pins.

2.3.3.3 Parallel Synchronous

In this mode the data can be driven directly onto a common data bus between the intelligent host and the device. The configuration control signals are connected to a port on the local host. The configuration clock can be driven from the system clock, but complete control over the interrupts is needed. Like in Master Parallel Up/Down, on the first rising clock edge a byte of configuration data is latched into the target device. The subsequent 8 falling clock edges serialize the data in the device. On the ninth rising clock edge the next byte is latched and serialized. The Ready/Busy pin indicates whether the device serializes data or is ready to receive the next byte.

2.3.4 JTAG

The *Bed of Nails* was the traditional method of testing electronic assemblies, but it has become obsolete due to smaller pin spacing and more sophisticated assembly methods (like surface mount technologies and multilayer boards). The Joint Test Action Group has developed a specification for boundary scan testing. The Boundary Scan Test (BST) is an

industry standard (IEEE 1149.1, or 1532) and it offers the capability to efficiently test components on PCBs with tight lead spacing. It can also test pin connections without using physical probes (like the Bed-of-Nails technique) and capture functional data while the device is operating normally. Another reason that this mode has gained popularity is due to its standardization and ability to program both FPGAs and CPLDs. Finally BST can be used to shift configuration data into the device. In this mode external logic is also required but this time to drive the JTAG specific pins, Test Data In (***TDI***), Test Mode Select (***TMS***) and Test Clock (***TCK***), and one optional the Test Reset (***TRST***). All other pins are tri-stated during JTAG configuration. JTAG configuration can start at any time, even during configuration through another mode. To avoid starting JTAG configuration accidentally, the JTAG pins should be kept stable during configuration of Altera devices and for Xilinx devices, at least one of the TCK, TDI, TMS should be kept High. The JTAG pins are described in Section *2.4 Configuration Pins*. To configure a single device in a JTAG chain, the software places all the other devices in BYPASS mode. JTAG testing can be performed before and after, but not during, configuration. The chip-wide reset and output enable pins do not affect JTAG boundary scan or programming operations. Toggling these pins does not affect JTAG operations. When designing a board for JTAG configuration, the regular configuration pins should be considered.

2.4 Configuration Pins

The configuration of FPGAs/CPLDs is performed through certain pins. During the configuration specific pins on the FPGA are used and these pins may act differently depending on the chosen mode (i.e. CCLK is an input in some modes and an output in some others). Finally one important note is that some pins are used in specific modes only (i.e. TDI, TMS, TCK are only used in JTAG). Also, most of the configuration pins, are not dedicated and reserved for configuration, so after the configuration process is complete, these pins can be used as user I/O. Each manufacturing company uses different names for the configuration pins, but although the names are different, there is a functional analogy among them.
This section gives a detailed description of the configuration pins and defines a universal pin naming, where this is feasible.

The **Mode Select** (**MS**) are the input pins and as denoted by their name they are used to select the configuration mode. The number of these pins varies, depending on the target device, from one up to three. It must also be mentioned that the encoding of a specific value in the MS pins is device dependent. If for example the value “010” implies Passive Serial configuration mode in one device, it does not necessarily mean that this applies to the rest. They are sampled before the start of the configuration and after the configuration process these pins can be used as user I/O.

The **Program** is an active-low configuration control input pin. A low transition forces the FPGA to reset. It is used to initiate the configuration process. When it goes high the FPGA begins configuration. All device I/Os go to tristate when the **Program** is de-asserted.

The **Configuration Clock (CCLK)** is either an input (i.e. Slave Modes) or an output (i.e. Master Modes). In Xilinx devices after configuration the **Configuration Clock** can be selected as a Readback Clock. In Altera, if selected from the software, this pin can be used as a user I/O.

The **DataIn** pin is the serial configuration data input receiving data on the rising edge of **Configuration Clock (CCLK)**, during the Serial Modes. During Parallel Modes, **DataIn** is the **DataIn0** input. In certain Peripheral configuration schemes, the **DataIn** pin represents the **Ready/nBusy** signal after the **RS** pin has been strobed. This is more convenient, for microprocessors, than using the **Ready/nBusy** pin. After configuration it is a user programmable I/O pin.

The **DataIn[7..0]** pins are the parallel configuration data input bus receiving data on the rising edge of **Configuration Clock (CCLK)**, during Parallel Modes. Each configuration data byte is serialized according to the specifications of the device. In some cases **DataIn0** is the MSB while in other cases it is LSB.

The **DataOut** pin during configuration is the serial output that can drive the **DataIn** of daisy-chained slave devices. Configuration Data appears on the **DataOut** pin after a specific number **CCLK** cycles. After configuration it is a user programmable I/O pin.

P_Done is an I/O signal. When used as an output the device drives this pin low before and during configuration. Once all data is loaded without error and the initialization cycle starts, the target device releases it. In other words, **P_Done** indicates the completion of the configuration process. When used as an input, a Low level on **P_Done** can be configured to delay the global logic initialization and the enabling of outputs.

nCS, CS, nWS, nRS are four inputs used in most Peripheral Modes. The chip is selected when **nCS** is Low and **CS** is High. After configuration these are user programmable I/O pins. If only

one chip select input is used the other must be tied to the active value (e.g., **nCS** can be tied to ground if **CS** is used). These two pins must be active during configuration and initialization.
The **nWS** and **nRS** pins should be mutually exclusive, but if both are Low simultaneously the Write Strobe overrides
The signal **nWS** is an active low Write Strobe input. A low-to-high transition causes the device to latch a bit or byte of data on the **DataIn** or **DataIn[7..0]** pins, respectively.
The signal **nRS** is an active low Read Strobe input. A low input on this pin directs the device to drive the **Ready/nBusy** (High if **Ready**, Low if **Busy**) signal on the **DataIn7** or **DataIn** pin and drives **DataIn[6..0]** High. If the **nRS** pin is not used, it should be tied high.
FPGA/CPLD vendors provide certain status pins that can be monitored in order to detect errors in the configuration process and observe the configuration progress.
For example the Altera **nSTATUS** pin is pulled low if an error occurs. If this is done by an external source, during configuration or initialization, the target device enters an error state. Driving this pin low after configuration does not affect the device. The **nSTATUS** pin can be used to indicate an error during configuration.
A similar pin is the bidirectional **INIT** pin in Xilinx devices. During configuration a Low on this output indicates that a configuration data error has occurred. This pin acts as an active Low open-drain output and is held low during the power stabilization and internal clearing of the configuration memory. As an active Low input it can be used to hold the FPGA in the internal WAIT state before the start of the configuration.
Similarities can also be observed between the **INIT** (Xilinx) and **INIT_DONE** (Altera). This is a status pin that indicates when the device has finished start-up and is in user mode. It drives low during configuration. Before and after configuration it is released and pulled to V_{CC} by an external pull-up resistor. Because **INIT_DONE** is tri-stated before configuration it is pulled high by the external pull-up resistor. Thus the monitoring circuit must be able to detect a low-to-high transition.
Finally there are the **TDI**, **TMS**, **TCK** and **TDO** for the JTAG. The **TDO** is the Test Data Out if Boundary Scan is used, if not it is a 3-state output after configuration is completed.
The **TDI**, **TCK** and **TMS** pins are the Test Data In, Test Clock and Test Mode Select respectively. They come directly from the pads, bypassing the IOBs. In some devices, once configuration is completed these pins become user programmable I/O. In some others, they can be used as user I/O but they must be kept stable before and during configuration, so as to prevent accidental loading of JTAG instructions.

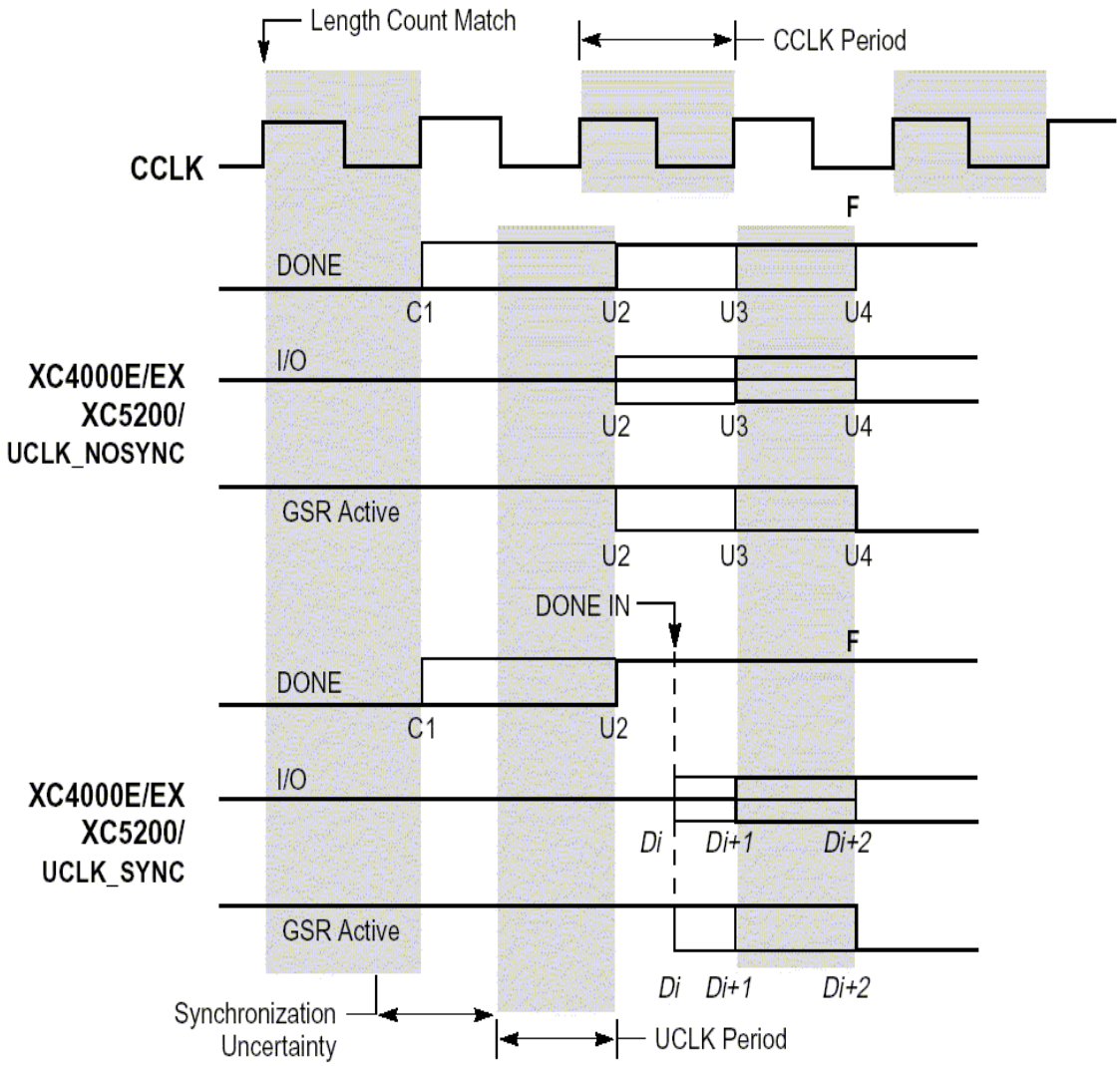

Figure 2.5 - Start-Up Timing for Xilinx XC4000/XC5200 devices [12]

2.5 Configuration Bitstream and Files

The Bitstream is a stream of bits that contains location information for logic on a device, that is, the placement of Configurable Logic Blocks (CLBs), Input/Output Blocks (IOBs), TBUFs, pins, and routing elements. Xilinx, unlike Altera, provides a substantial amount of information concerning the structure of its devices bitstream, therefore the main part of this section refers to Xilinx.

The bitstream includes empty placeholders that are filled with the logical states sent by the device during a readback. Only the memory elements, such as flip-flops, RAMs, and CLB outputs, are mapped to these placeholders, because their contents are likely to change from one state to another. When downloaded to a device, a bitstream configures the logic of a device and programs the device so that the states of that device can be read back. An example bitstream, for XC3000[14], and the switching characteristics are shown in Fig. 2.6 below.

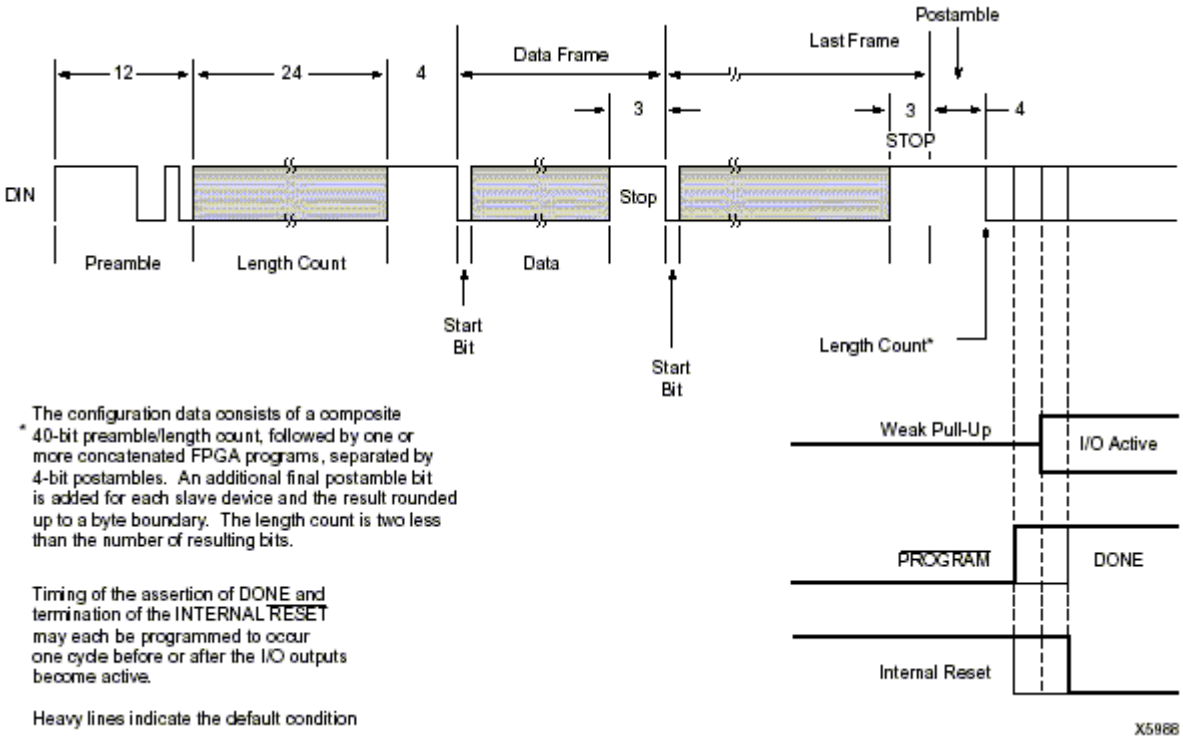

Figure 2.6 - Configuration and Start-up for XC3000 [9]

The bitstream format is very similar in Xilinx Families XC3000, XC4000[15], XC5200[16] and SPARTAN/XL. It starts with 8 Dummy or Fill bits and is followed by the preamble code which is 4 (0010 in XC3000, XC4000 and Spartan) or 8 bits (11110010 for XC5200 and Spartan XL Express Mode). The next part is the 24-bit length count. When configuration is initiated a counter in the FPGA is set to zero and begins to count the total number of configuration clock cycles applied to the device. The Configuration Loading process is completed when the current length count equals the loaded length count and when the required configuration program data frames have been written. The last part of the header is 4 fill or dummy bits or 8 in the XC5200 Family and none in the Spartan XL Express Mode. For

Spartan XL Express Mode the Length Count is not used by configuration logic and is considered fill bits. The next part of the bitstream is Data Frame which is divided in frames. The number and length of frames depends on the device. Each frame begins with a start field ranging from 1 bit for XC3000 to 8 bits in XC5200 and Spartan XL Express Mode. Its next field is the configuration data which varies in size from one device to another. And the frame ends with a Cyclic Redundancy Check (CRC) or a Constant Field Check which is 4 bits (XC3000 and Spartan XL in Express Mode do not support CRC). The Constant Field Check is used if the CRC error check is disabled. Detection of an error, as mentioned before, results in the suspension of data loading and the pulling down of the **INIT** pin.
Altera does not provide any information regarding the structure of its configuration files, so we are limited to a brief reference to the programming files. The following sections contain a description of the files used to program Altera and Xilinx FPGAs.

2.5.1 Bit File (.bit) - Xilinx

A Bit file is used to program a single FPGA. It is a binary file, which contains all the configuration information, as well as device specific information from other files. The Bit file is the standard bitstream file created.

2.5.2 Raw Bit File (.rbt) - Xilinx

A Raw Bit file is also used to program a single device, but it is an optional file. It is an ASCII version of the Bit file, containing ASCII ones and zeros. Another difference from the Bit file is that the header information is removed from the Raw Bit File.

2.5.3 Raw Binary File (.rbf) - Altera

The Raw Binary File is a binary file, containing configuration data. Data must be stored so that the least significant bit (LSB) of each data byte is loaded first. The converted image can be stored on a mass storage device. The microprocessor can read data from the binary file and load it into the device. A microprocessor can be used to perform real time conversion during configuration. In PPS and PPA configuration schemes, the target device receives its

information in parallel from the data bus, a data port on the microprocessor or some byte-wide channel. In PS and PSA the data is shifted in serially, LSB first.

2.5.4 Hexadecimal (Intel Format) File (.hex) - Altera

A Hex file is an ASCII file in the Intel Hex format. The file is used by third-party programmers to program Altera's serial configuration devices. Hex files are also used to program parallel configuration devices with third-party programming hardware. You can use parallel configuration devices in PPS, PPA or PSA configuration schemes, in which a microprocessor uses the parallel configuration device as the data configuration source.

2.5.5 Tabular Text File (.ttf) - Altera

The Tabular Text file is a tabular ASCII text file that provides a comma-separated version of the configuration data for the PPA, PPS, PSA and bit-wide PS configuration schemes. In some applications, the storage device containing the configuration data is neither dedicated to nor connected directly to the target device. For example, a configuration device can also contain executable code for a system and other data. The TTF allows you to include the configuration data as a part of the microprocessor's source code using the *include* or the *source* command. The microprocessor can still access this data from a configuration or mass storage device and load it into the target device. The TTF can be imported into nearly any assembly language or high level language compiler.

2.6 Spartan II and Virtex Configuration - Xilinx

2.6.1 Configuration Modes and Daisy-Chains

Spartan II and Virtex FPGAs can be configured in 8 different modes. There are four primary modes (Master Serial, Slave Serial, Slave Parallel (Spartan II) or SelectMAP (Virtex) and Boundary Scan), each with the option to have the user I/Os pulled up or floating during

configuration. If pull-ups are selected, they are only active during configuration. After configuration unused I/Os are de-asserted.

The **Serial Modes** perform essentially the same as those of previous FPGAs families.

In **Parallel Modes**, the Slave Parallel/SelectMAP are the 8-bit parallel mode for these devices that are similar to the Express Mode in XC4000XLA and Spartan/XL. As with these other device families, the D0 is considered the MSB. Spartan II and Virtex FPGAs do not have a Master Parallel mode, but can be configured using a parallel EPROM.

Spartan II and Virtex devices can be serially **Daisy-Chained** for configuration as all previous FPGA families. All devices must be in one of the serial modes. The Slave Parallel and SelectMAP modes do not support any serial daisy-chaining. Multiple devices can still be programmed using these modes in a parallel fashion.

Boundary Scan is always active from the moment of power-up, before, during and after configuration. Boundary Scan modes select the optional pull-ups and prevent configuration in any other modes.

2.6.2 Initialization and Timing

The initialization sequence is somewhat simpler in Spartan II and Virtex devices. Upon power-up the INIT signal is held low while the FPGA initializes the internal circuitry and clears the internal configuration memory. Configuration may not commence until this cycle is complete, indicated by the positive transition of INIT. Previous FPGAs families required an additional waiting period after INIT transitioning high before configuration could begin. Spartan II and Virtex devices do not require an additional waiting period after INIT transitioning high. As soon as this occurs, configuration may commence. The Spartan II and Virtex configuration logic does however require several CCLK transitions to initialize themselves. For this reason the configuration bitstream is padded with several dummy data words at the beginning of the stream.

2.6.3 Mixed Voltage Environments

Spartan II and Virtex FPGAs have separate voltage sources for the internal core circuitry (V_{CC} = 2.5V) and the I/O circuitry (SelectI/O). SelectI/O is separated into eight banks of I/O groups. Each bank may be configured with one of several I/O standards. Before and during

configuration all I/O banks are set for the LVTTL which requires an output voltage of 3.3V for normal operation.

2.6.4 BitGen Switches and Options

This section describes the new optional settings for bitstream generation that pertain only to Spartan II and Virtex devices. The new BitGen options found in the configuration options template of the Xilinx development software are listed in table 2.1 and described below.

Switch	Default Setting	Optional Setting
Readback	N/A	N/A
Config Rate (MHz) (nominal)	4	4, 5, 7, 8, 9, 10, 13, 15, 20, 26, 30, 34, 41, 45, 51, 55, 60
StartupClk	CCLK	UserClk, JtagClk
DONE_cycle	4	1, 2, 3, 5, 6
GTS_cycle	5	1, 2, 3, 4, 6, DONE
GSR_cycle	6	1, 2, 3, 4, 5, DONE
GWE_cycle	6	1, 2, 3, 4, 5, DONE
LCK_cycle	NoWait	0, 1, 2, 3, 4, 5, 6
Persist	No	Yes, No
DriveDONE	No	Yes
Donepipe	No	Yes
Security	None	Level1, Level2
UserID	FFFF FFFF	<hex string>(32-bit)
Gclkdel0	11111	<binary string>11111
Gclkdel1	11111	<binary string>
Gclkdel2	11111	<binary string>
Gclkdel3	11111	<binary string>

Table 2.1 Spartan II and Virtex BitGen Options

Note: Spartan II has the extra option KEEP for GTS, GSR and GWE cycles.

2.6.5 CCLK and LengthCount

Spartan II and Virtex FPGAs do not use any LengthCount number in the configuration bitstreams. The Start-Up sequence for these devices is controlled by a set of configuration commands that are embedded near the end of the configuration bitstream. Thus a free running oscillator can be used to drive the CCLK pin. Figure 2.7 below displays the default Start-up sequence timings for Virtex FPGAs, with the bold lines. This figure also applies to Spartan II FPGAs if we invert the GWE signal.

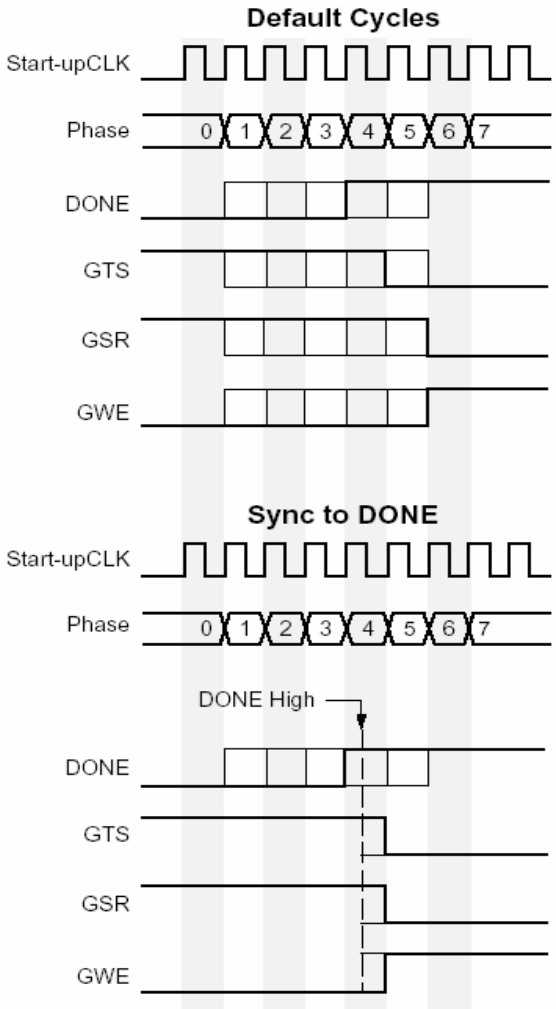

Figure 2.7 - Default Start-Up Sequence [17]

2.6.6 Configuration Pins

The configuration pins for Spartan II and Virtex are listed in Table 2.2 below.

Name	Direction	Driver Type	Description
Dedicated Pins			
CCLK	Input/Output	Active	Configuration clock. Output in Master mode.
$\overline{\text{PROGRAM}}$	Input	Active/ Open-Drain	Asynchronous reset to configuration logic.
DONE	Input/Output		Configuration status and start-up control.
M2, M1, M0	Input		Configuration mode selection.
TMS	Input		Boundary-scan tap controller.
TCK	Input		Boundary-scan clock.
TDI	Input		Boundary-scan data input
TDO	Output	Active	Boundary-scan data output.
Dual Function			
DIN (D0)	Input/Output	Active Bidirectional	Serial configuration data input.
D[0:7]	Input/Output	Active Bidirectional	Slave Parallel configuration data input, readback data output.
$\overline{\text{CS}}$	Input		Chip Select (Slave Parallel only).
$\overline{\text{WRITE}}$	Input		Active Low write select, read select (Slave Parallel only).
BUSY/ DOUT	Output	Open-Drain/Active	Busy/Ready status for Slave Parallel (open drain).
$\overline{\text{INIT}}$	Input/Output	Open-Drain	Serial configuration data output for serial

Table 2.2 Configuration Pins for Spartan II and Virtex

2.7 PCI Pamette v1

The PCI Pamette is manufactured by Compaq and is a generic PCI card based on reconfigurable logic.

2.7.1 The Hardware Architecture

The PCI Pamette has footprints for 5 Xilinx 4000 series FPGAs in PQ208 packages. One FPGA serves as the PCI interface (a 4010-E), while the other four (4044-XL) are organized in a simple two by two matrix. The FPGA implementing the PCI interface has a relatively

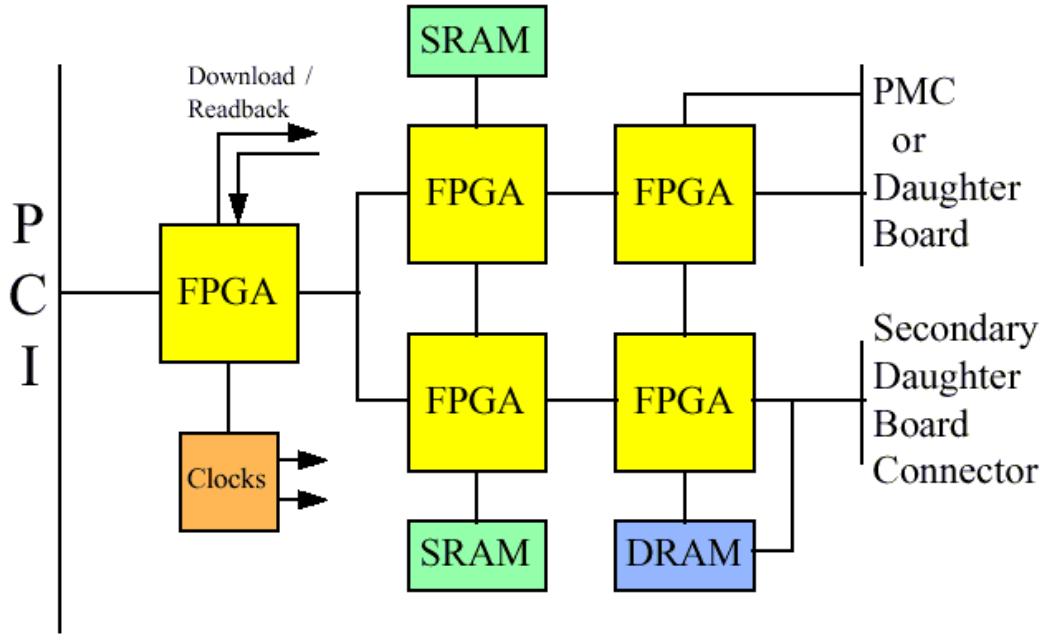

Figure 2.8 – PCI Pammete v1 Architecture [18]

fixed configuration, while the remaining four can be programmed with application specific configurations. As depicted in Fig.1, the front two user FPGAs connect to private scratchpad SRAMs and funnel data from the rear two user FPGAs to the interface FPGA. The rear two user FPGAs connect to the daughter board connectors and the DRAM SIMM sockets. The SRAM is divided in two banks of 16-bit wide 64k, and through the supplied connectors, industry standard 72-pin SIMM DRAM modules which permit from 4MB to 256MB of DRAM can be attached. The board has also a clock generation scheme.

The interface FPGA controls download and readback of the user area and generation of clocks. User FPGAs can be individually reconfigured without affecting other FPGAs, however when multiple FPGAs are reconfigured or their configurations are read, this is done in parallel.

2.7.2 Programming Tools

The PCI Pamette can be programmed using one of the following programming tools:

- Code-based module libraries from Compaq (PamDC[19])
- Tools from Xilinx, Inc.
- Any logic synthesis tools which generate the Xilinx .rbt raw bitstream format

Provided with the Pamette is a set of CAD tools called PamDC for implementing designs. PamDC is derived from Perle1DC, the CAD system of DECPeRLe-1. PamDC is a C++ class library which allows netlist descriptions to be embedded in user-written C++ code. The Pamette CAD provides support for attaching placement directives to nets at the C++ level. C++ classes are used to represent the hierarchy of a design, equivalent to blocks and sub-blocks in a schematic.
The Pamette design flow consists of writing a C++ program which, when compiled and run, produces a netlist. This netlist is then passed to the Xilinx backend tools to produce a Xilinx bitstream. Use of the CAD tools is not mandatory; any technique which generates a Xilinx bitstream can be used to configure the Pamette FPGAs.

2.7.3 Interface Modes

In firmware v2.0 four distinct interface modes are supported. These are selected through the <*PamRT.h*> function *PamSetMode* or *PamSetModeAndDelay* which set the appropriate value(s) in the decode register at address 0x30 in PCI Pamette memory space. The modes are

Static mode a simple low-performance interface that provides statically configured 16 bit paths to and from the user-area. The static mode is displayed in Fig.2.

Promiscuous mode transmits a selection of data and control values present on the PCI bus to the user-area. The flow of data is one-way. The name *promiscuous* is chosen by analogy with modes on some ethernet adapters which allow them to accept and observe all packets on the network.

Transaction mode is a high performance transaction oriented mode that supports both target and master operation. This is the preferred mode for all but the simplest designs.

Promiscuous Transaction mode combines the protocol state machine of Transaction mode with the trace collection capabilities of Promiscuous mode. It can aid in the debug of Transaction mode applications.

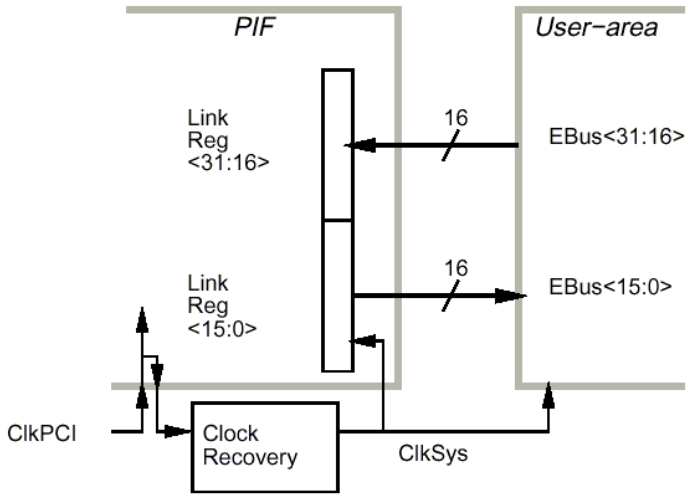

Figure 2.9 – Static Mode [18]

2.8 The JBits API - Xilinx

The JBits SDK contains a set of software tools and API's which let you create Xilinx XC4000 and Virtex bitstreams from Java code. All configurable resources of the device can be programmed individually through the software. Included is a graphical debugger called BoardScope, which allows you to view chip internals at runtime. BoardScope can also be used in simulator mode if hardware is unavailable.

2.8.1 The JBits Design Flow

The JBits API is based on the **J**ava **E**nvironment for **R**econfigurable **C**omputing for the XC6200 family of devices (JERC6K). JERC6K was also implemented completely in Java and provided fast compile times and supported dynamic reconfiguration.
JBits is a library which gives complete access to all of the configurable architectural features of the device. This library is pre-compiled Java classes, so the result is not a static configuration bitstream, but rather executable code. This code executes and supplies configuration control and data to the reconfigurable logic. The Design Flow for JBits is depicted in Fig. 2.10.

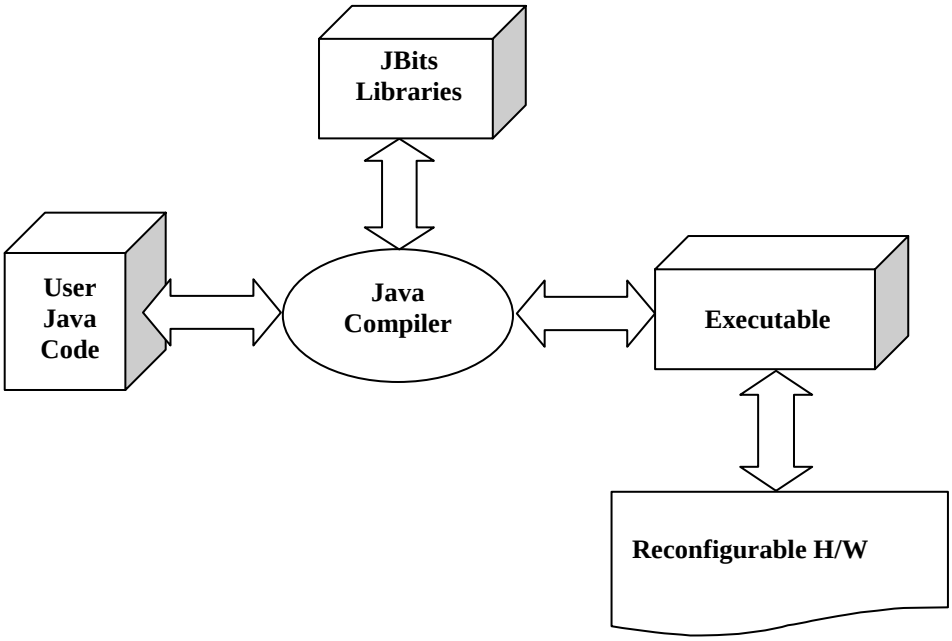

Figure 2.10 JBits Design Flow

2.8.2 The JBits System Design and BoardScope

The JBits System Design is illustrated in Figure 2.11. The User Java Code utilizes the JBits Interface to manipulate the configurable resources of the FPGA. The Bit Interface Level is called by function calls at the JBits Interface Level. At this point a single bit can be configured or cleared. The Bit Level Interface also interacts with the Bitstream class, which manages the device bitstream and provides support for reading and writing bitstreams to and from files. The Bitstream class can also take data read back from the device and map it the

underlying bitstream data format. Finally, the Core Library is a collection of Java classes which define macrocells or cores.

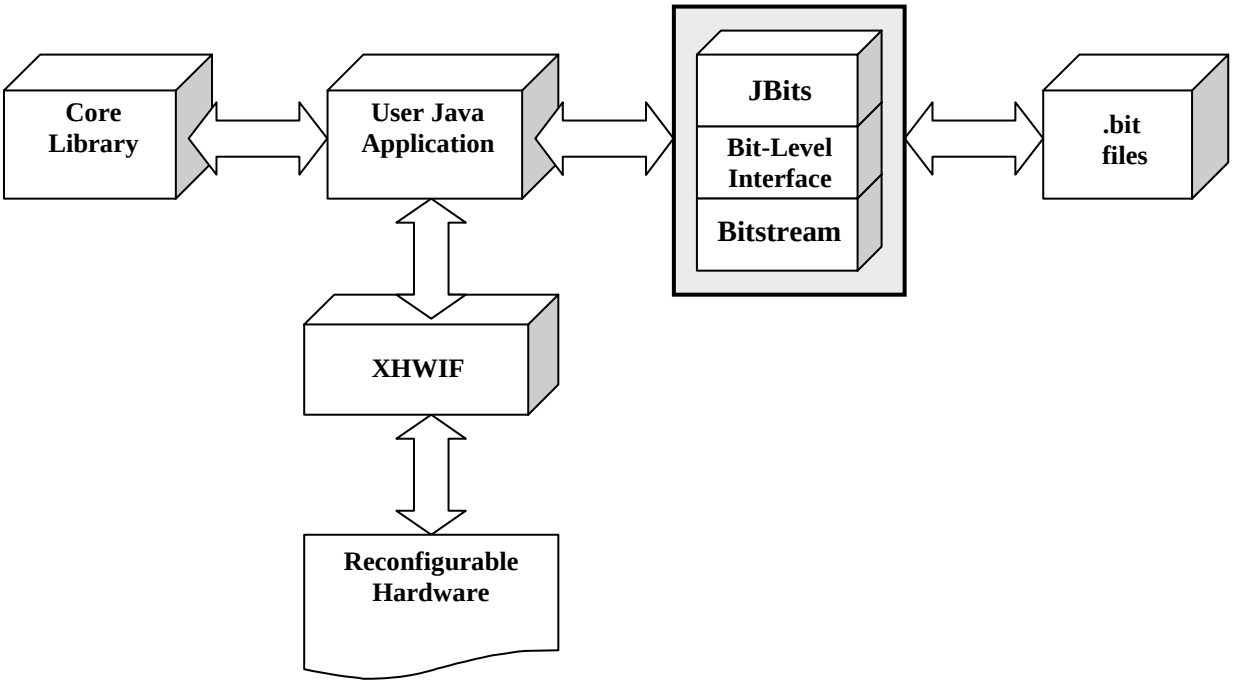

Figure 2.11 - The JBits System Design

BoardScope is a tool that enables the user to examine graphically the operation of FPGA circuits on any reconfigurable computing board. It can be used to verify the design's operation.

The JBits interface is used to access resources in the FPGA's bitstream. Then using XHWIF [20] the bitstreams are downloaded to configure the FPGAs, or readback to analyze them. BoardScope graphically displays the states of all CLB flip-flops for all computational FPGAs on the board. All the CLB flip flops can be examined in one view and flips flops changes from one state to the next can be observed. BoardScope also provides a more detailed view of the state of a Configurable Logic Block showing the look up table states, the X and Y flip-flop configuration and states, and the CLB's internal interconnect.

2.8.3 Limitations of JBits

The most important limitation of JBits is its manual nature that requires that everything is explicitly stated in the source code, including the routing. This need for explicit specification

of all resources makes the JBits interface more appropriate for structured circuits. Unstructured circuits such as random logic are not well suited for direct implementation in JBits applications.
JBits also requires that the user is familiar with the architecture. This makes it hard to use, because most users have never had the need of such details. It is also the greatest barrier to the widespread acceptance of JBits interface.
In addition, the JBits interface is at the most downstream end of the tool chain. While JBits API may make use of circuits produced by standard development tools, modification or reconfiguration of the circuit at the JBits level eliminates any possibility of using any analysis tools available to circuit designers further up the tool chain. One tool which appears to have at least partially offset the lack of analysis tools is the recent development of BoardScope.

2.9 MasterBlaster Serial/USB Communications Cable - Altera

The MasterBlaster Serial/USB Communications Cable is a typical Altera programmer that interfaces with an RS-232 serial or Universal Serial Bus (USB) port. It supports the SignalTap [21] embedded logic analyzer in the Quartus II software [22]. PC and UNIX users can configure every Altera FPGA/CPLD with the MasterBlaster. The power is received from 5.0-V or 3.3-V circuit boards, a DC power supply or 5.0-V from the USB cable.
The data is downloaded from the Quartus II development software or the MAX+PLUS II software [23] versions 9.3 and higher. The modes supported are Passive Serial and JTAG. Finally, a 10-pin circuit board connector is used.

2.10 MultiLinx Download Cable - Xilinx

The MultiLINX Cable is a peripheral hardware product released by Xilinx in 1999. This cable is primarily used for the purpose of downloading configuration and programming data to Xilinx FPGAs and CPLDs from a host computer to a users' target system.
The MultiLINX cable supports a USB interface. The MultiLINX Cable is also outfitted with all the appropriate flying leads for multiple configuration mode support, as well as supporting multiple readback modes such as verification, Capture, and the Virtex SelectMAP interface.

Finally, a noted feature of MultiLINX is that its internal hardware is upgraded via software, something that allows future expansion of cable features

2.11 Summary

In this chapter we discussed matters concerning the configuration of FPGAs and related work. The PCI Pammete v1 development board was presented. The Pamette design flow consists of writing a C++ program which, when compiled and run, produces a netlist. This netlist is then passed to the Xilinx backend tools to produce a Xilinx bitstream. This bitstream can then be downloaded to the FPGAs either by writing C++ code and including the libraries or by using a command prompt utiliy that requires writing the same program without including the libraries. The JBits API, is a powerful tool, but requires that the user is familiar with the FPGA architecture. This makes it hard to use and is probably the reason for not being widespread. MultiLinx and MasterBlaster are reliable solutions but they are vendor specific and as we will see in the next chapter, their prices are inhibiting.

Chapter 3

Architecture of ReRun

Chapter 3

Architecture of ReRun

This chapter describes the architecture of ReRun and its purpose, the development of a language and a universal run-time environment for FPGA programming. The language is named PTL and it is supplemented by a Graphical User Interface (GUI).

The first section contains a brief description of ReRun (**Re**configurable **Run**-Time Environment). The second section moves on to describing its features and attributes. The third analyzes its structure and gives an insight of the language and the GUI. In the fourth section the commands of the PTL are described. The fifth section makes a functional description of the back-end. Finally, the sixth section contains a summary of this chapter.

3.1 Brief Description

ReRun (**Re**configurable **Run**-Time Environment) is a system developed at the **M**icroprocessor and **H**ardware **L**aboratory (MHL) [24] of the Technical University of Crete [25]. It is the software part of a Universal Run-Time Environment for FPGA programming. As the title denotes during the implementation of ReRun a language was developed along with a Graphical User Interface. The language can be used to write scripts describing the configuration or testing process. The scripts are then passed through the compiler using the GUI. The compiler produces two files, one to be downloaded to the FPGA, the avr.dl and one to be used by the GUI, the props.gui. In case the script is intended for configuring an FPGA, the bitstream file produced by the Development Tools of the FPGA vendor, has to be provided by the user. Then through the GUI, the user can program and test any FPGA. The hardware platform of the Run-Time Environment is named Hardware Programmer and Tester (HPT) and was implemented by Dionissis Efstathiou. The software-hardware interface is established using the RS-232 serial port protocol.

3.2 Features and attributes

ReRun has a number of features that make it unique. First of all, it is *versatile* as it is a programmer and tester of FPGAs. It is also *vendor-independent* for it can be used for FPGAs of any vendor. Looking at Table 3.1, that lists FPGA programmers along with their prices, it is obvious that the cost of buying one is quite considerable, even more so if you require one programmer from each vendor. Besides being a many-in-one tool, ReRun is an *open system* with a cost which can be as little as $25.

Xilinx Cable	**Price**
MultiLinx Cable	$495
Parallel Cable III [26]	$95
Altera Cable	**Price**
MasterBlaster	$576
ByteBlaster [27]	$576

Table 3.1 FPGA Programmers Cost

Another notable feature is the *Graphical User Interface (GUI)*. A GUI is a fundamental aspect of the design of this system. It presents tasks visually, so that the tasks are easy to learn and prevent errors. It also combines functionally and usability. ReRun has a user-friendly GUI that allows the user to open, edit, compile and execute a script file, change the port settings, etc. A more thorough description of its features is made in Section 3.3.2.

The language has been developed using Flex [28] and Bison [29], while the GUI was implemented with Java. This has made ReRun portable and *platform-independent*. Java compilers do not produce native object code for a particular platform but rather ‘byte code’ instructions for the Java Virtual Machine (JVM) [30]. Making Java code work on a particular platform is then simply a matter of writing a byte code interpreter to simulate a JVM. What this means is that the same compiled byte code will run unmodified on any platform that supports Java. Flex and Bison on the other hand produce a C program, which with few modifications can become cross-platform.

The communication between hardware and software is achieved using the *serial port of a computer*. Thus it does not consume a lot of system resources, like for example a PCI card would. Moreover, it does not require the installation of hardware drivers specific to the

Operating System. A common problem for cards is that drivers for all operating systems are not available.

3.3 Structure

The Language and GUI are the most important parts of ReRun. Fig 3.1 is a graphical illustration of the structure. The tools used to implement the language, as well as the files produced by the PTL compiler are discussed in Section 4.3.1, while Section 4.3.2 describes the GUI in more detail.

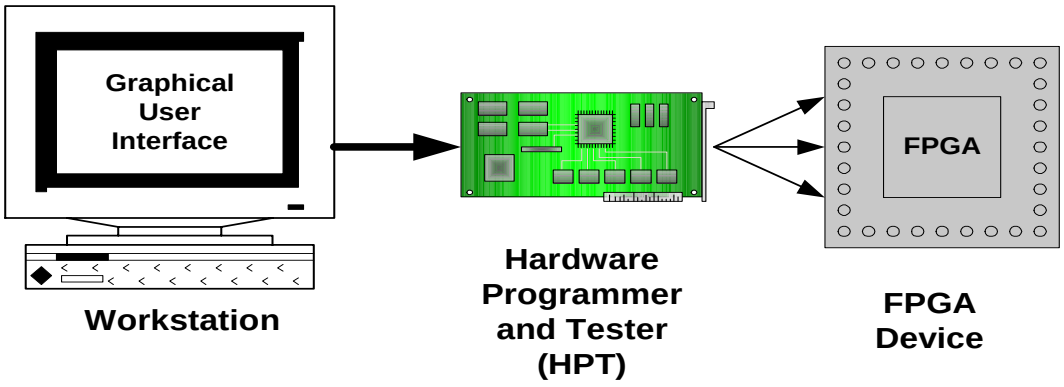

Figure 3.1 ReRun Structure

3.3.1 The Programming and Testing Language (PTL)

The PTL language was developed using GNU's **Flex** Lexical Analyzer Generator and **Bison** Syntactical Analyzer Generator. More specifically:

Flex is a tool for generating scanners: programs which recognized lexical patterns in text. Flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates as output a C source file, `lex.yy.c', which defines a routine `yylex()'. This file is compiled and linked with the `-lfl' library to produce an executable code. When the executable code is run, it analyzes its input for

occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Bison is a general-purpose parser generator that converts a grammar description for an LALR context-free grammar into a C program to parse that grammar.

Figure 3.2 depicts the steps taken to develop the compiler. First the input file containing the rules is read from Flex and the lex.yy.c file is generated. On the other hand the file expressing the grammar in Bison syntax is read from Bison and a filename.tab.c file is produced. Finally, the two files generated by Flex and Bison are compiled by a C/C++ compiler and the PTL Compiler is ready.

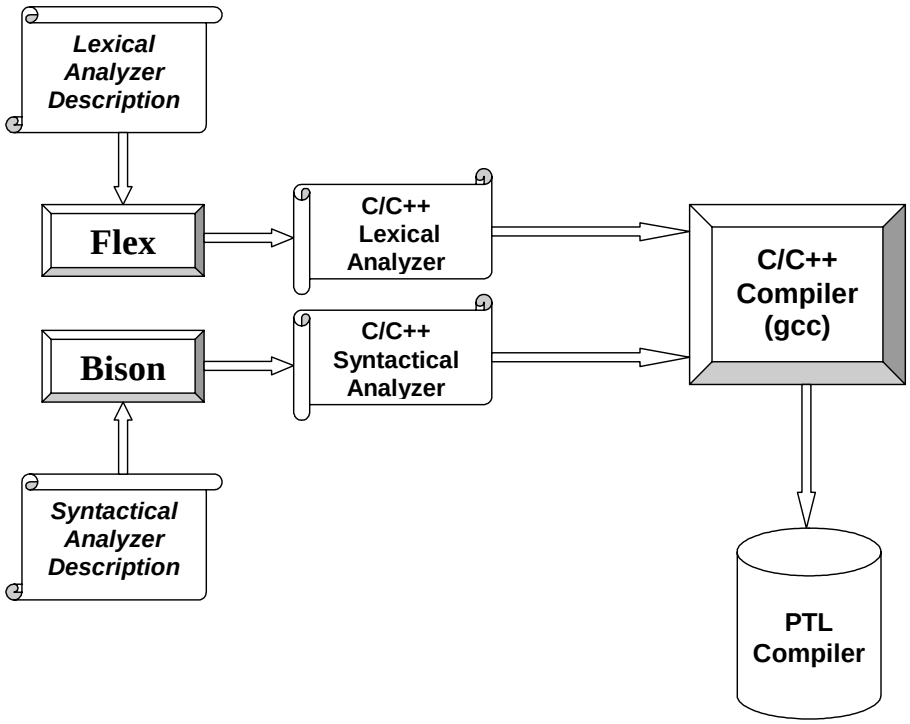

Figure 3.2 Compiler Development

The user can choose to compile a script from the command line or load the script using the GUI and compile it by pressing one button. Doing this will produce two files. The first, avr.dl, is the one that will be downloaded to and parsed by the HPT. The second file, props.gui, contains information that is utilized by the GUI. Specifically this information is the **Configuration Clock Frequency**, the **Voltage Source**, the **Signal and Static Mapping** and the **Number of Signals Used**. Section 3.4 provides more information concerning the language and presents its commands, while Chapter 4 delves into more technical details.

3.3.2 The Graphical User Interface

The GUI was implemented using the Java Programming Language and the Java Communications Application Programming Interface (API) [31]. This choice was made, taking into consideration two factors, the increasing usage of Java-based tools and technology in the Reconfigurable Logic Industry and the cross-platform operability of Java Applications.

Examples, of Java in Reconfigurable Logic are Xilinx's JBits API and JHDL [32]. JBits is a Java API that allows designers to write information directly to a Xilinx FPGA to carry out whatever customer logic operations were designed for it. It permits the modification of FPGA bitstreams and can be used to partially or fully reconfigure the internal logic of Xilinx's Virtex devices. On the other hand, JHDL is a design tool for Reconfigurable Systems, implemented as a set of Java class libraries, that allows the user to design the structure and layout of a circuit, debug the circuit in simulation, netlist and interface with back-end tools for synthesis, and so forth.

The Java Programming Language and Communications API can be used to write platform-independent applications. The Java Comm API used contains support for RS232 serial ports and IEEE 1284 parallel ports. More specifically, the user of the API can:

- ✓ Enumerate ports available on the system.
- ✓ Open and claim ownership of ports.
- ✓ Resolve port ownership contention between multiple applications.
- ✓ Perform asynchronous and synchronous I/O on ports.
- ✓ Receive Beans-style events describing communication port state changes.

A test was performed to ensure that the Java Comm API can be sent to and received from the HPT hex numbers ranging from 0x00 to 0xFF. The test was successful, thus no encoding was needed to ensure transmission of special characters through the serial port and this does not depend on the serial port driver.

The GUI was created using Sun's Forte™ for Java™ Community Edition Integrated Development Environment (IDE) [33]. The GUI is further analyzed in Chapter 5.

3.3.3 Operating Instructions

ReRun is not a completely independent Run-Time Environment. It requires that the user provides the configuration bitstream file. This file is produced by the development tools of the FPGA vendor. For Xilinx, the Bit file (.bit), described in Section 2.5.1 is required, while for Altera, the Raw Binary File (.rbf), described in Section 2.5.3 must be provided. This file is not produced automatically by the development tools, so the user must convert the bitstream produced to .rbf.

The user must also choose a script or write one in the GUI. The script must be compiled before it is downloaded to the HPT.

For programming scripts, the user must load the configuration bitstream file. When loaded, its size is displayed, to help the user calculate the load instruction's arguments.

For Xilinx devices, the Bit files contain a header which must be cut from the bitstream. To do this, the user must check the Process Bitstream checkbox. This option should be left unchecked for Altera devices or the device will not be configured.

To download a script to the FPGA, the user must connect the serial cable to the HPT. Then the appropriate set of cables must be connected to the FPGA and the system must be powered up. To connect these cables to the correct FPGA pins, the user must read the data sheet provided by the vendor.

During program scripts, the Clock is generated by the HPT, thus the user does not have to take any actions to create and map a signal for it in the script. On the other hand, in a test script, it must be implemented by the user, using the set instructions, which are described in the following section.

Finally, the configuration data is loaded to the device from a specific set of flying wires and the user only has to use the load instruction for the data to be loaded to the device. This means that no signal declaration is required for the data.

3.4 The Script Language

The Programming and Testing Language (PTL) is designed to support the programming and testing of FPGAs. It is interpreted to binary code that is sent through a port (the serial port in this implementation) to the Hardware Programmer and Tester (HPT). The HPT then executes the code and either programs a device or tests it. A PTL script consists of the header and the

main part of the script. The header is a series of declarations and the signal mappings. The main part is where the user types the code to be executed by the HPT.
Comments are supported by PTL. Line comments start with a double backslash (//) and stop at the end of the line. On the other hand, multi-line comments start with a backslash followed by an asterisk (/*) and terminate when an asterisk and a backslash are found (*/). Comments in comments are not allowed.
Finally, the PTL language supports instructions that are not implemented by the HPT. These instructions are optional and the user can write and execute scripts for the HPT without any problem.

3.4.1 Header

In the beginning of the script the user can write some comments about the FPGA. The next step is defining whether it is a program or test script. Then he can define the frequency of the configuration clock and the voltage source. These three parts are optional, but if used they must be defined in the specific order. Finally the header of the script consists of the declaration of integers, signals and statics and the mapping of the last two.
The commands that can be used in the header of the script are listed below, in the order they must be typed.

Comments (Optional)

Comments are used to provide information about the device to be programmed or tested. Below is an example of a comment:

manufacturer "Xilinx";
family "XC4000";
device "XC4044XLHQ208";

The comment statement, if present, requires all fields (manufacturer, family, device) to be specified. This feature is not utilized by the current version of the HPT.

Program or Test Script (Optional)

This part defines the type of the script. In case of a program script the user must type:

program “PM” ;

PM is the programming mode. Currently this can be *serial* or *parallel* and instructs the HPT about how to load the data, serial or parallel. To define that the script is a test script the user must type:

test;

This causes the HPT to use the port reserved for loading the configuration data, as inputs for testing.

LSB or MSB (Optional)

This part defines how the configuration data is loaded to the FPGA, Least Significant Bit (LSB) first or Most Significant Bit (MSB) first. Xilinx for example loads the configuration data MSB first, while Altera loads it LSB first. This command can be used only in program scripts. The syntax is

lsb;
or
msb;

Clock Declaration (Optional)

In this part the user can define the configuration clock rate or the test clock frequency. This command has two versions. The first is:

clk value (unit);

The value is an integer and the units are KHz, Khz, khz, MHz, Mhz, mhz. This feature is not utilized by the current version of the HPT.
The second version is:

clk low;
clk high;

This version tells the HPT that if, for example, *clk low* is selected, the configuration clock has no maximum low time, but there is a limit to the high time.

Operating Voltage Declaration (Optional)

It is well known, that FPGAs can operate at various voltage supplies. If the HPT can provide more than one voltage supply, the user can use the language to define the preferred value. The syntax is:
vs value (unit);

The value is an integer and the units are V, v, mV, mv. This feature is not utilized by the current version of the HPT.

Variable Declarations

The typical variable declarations are:

type id;
or
type id1,id2,…,idn;

The type can be int and signal and the name can be anything that is not a reserved word. The type int is the same as in C programming language. The signals declared here must be mapped in the Signal Mapping part.

Static Variables (Optional)

This feature can be used to hold certain signals at the specified value throughout the whole configuration or testing process. Its syntax is:

Static id 'state';

The statics declared must be mapped in the Signal and Static Mapping part of the script.

Signal and Static Mappings

In the signal mapping part of the script, the user must correspond the declared signals and statics to a specific cable of the HPT. To define that the signal or static is an output the user must use the symbol => and to define that it is an input the <=. Finally a static can not be defined to be an input. For example to correspond the signals mm0, mm1, init and prog to cables 0, 1, 2, and 3 as input, output, input and output respectively, the user must write:

map{
mm0 => 0;
mm1 <= 1;
init => 3;
prog <= 4;
}

A signal or static can not be mapped to more than one wire and vice versa.
Warning! The maximum number of signals that can be mapped is 23. The first 16 (0-15) can be inputs and outputs, while the last 8 (16-23) can only be inputs, to the HPT, in the case of a test script. In programming mode pins 16-23 are used for loading configuration data to the FPGA so they can not be used!

3.4.2 Main Program

This is the main part of the script and the code is written between a **start** – **end** statement. The language supports the following statements:

- Assignment
- For Loop
- Set
- Load (Byte, Kbyte)
- Readback (Byte, KByte)
- Get
- Nop
- Wait
- Compound Statement

Assignment

An integer variable can be assigned to an expression. An expression can be an integer variable, a number or a simple arithmetic operation such as addition, subtraction, multiplication and division. For example:
id1 = id1 + id2 + id3;
id2 =id3 + 15;

Compound Statement

Commands within a compound statement are executed simultaneously by the HPT. Only set statements are allowed and they must be two or more. For example to set the values of signals init, program and confdone to 1, 0, 1 you type:

{
set init ‘1’;
set prog ‘0’;
set confdone ‘1’;
}

For Loops

The For Loop can be used to repeat commands for a number of times equal to that specified by the user. It starts with the **for** keyword followed by a number and ends with an **endfor**:

for expression
…
endfor

The expression can be a number or a variable, denoting the number of iterations. The current maximum value for the *expression* is 256. Load and readback instructions can not be inside a for loop.

Get

This command returns the values of all the signals and statics declared by the user. The HPT checks them and sends their values in the appropriate form. This command consists of a single word, the word get followed by a number from 0 to 3 and a semicolon:

get 0;

Depending on the number, the HPT returns the values of all or some signals:
0 – Get data from all signals (Input and Output)
1 – Get data from signals mapped to pins 0-7.
2 – Get data from signals mapped to pins 8-15.
3 – Get data from signals mapped to pins 16-23.

Load

This command notifies the HPT that the next x Bytes or Kbytes of data sent through the serial port are from the bitstream file and must be loaded to the FPGA. Depending on the programming mode, the HPT loads the data in the device at a rate of 1 bit/clock cycle in serial modes or 1 byte/clock cycle in parallel modes. The data loaded by one load instruction can be up to 256 bytes or Kbytes. This means that to load a file of size 3,526 bytes, the user must type:

loadkb 3;
loadb 256;
loadb 198;

As mentioned above, the current limit for this instruction is 256.

Nop

This instruction can be used for adding delays. The argument defines how many nops will be sent to the HPT.

nop 5;

Readback

The readback command is similar to load but causes the HPT to send data read back from the FPGA, to the GUI. There are two versions of this command. The one specifies number of bytes to read and the other number of Kbytes. Its syntax is simple, it is the keyword readbackb or readbackkb followed by the number of bytes or Kbytes respectively and a semicolon:

readbackb 230;
readbbackkb 20;

Reverse

The reverse command toggles the state of a signal from input to output or vice versa. This is very useful in the case of input/output signals. Its syntax is the keyword reverse followed by the name of the signal:

reverse init;

This is command is not allowed for signals mapped to pins 16-23 and statics.

Set

This command is used to set the value of a signal to a specific state. Finally a static can not be set. For example:

set program '1' ;

Wait

The wait command causes the HPT to wait until a signal gets a specific value to continue normal operation.

wait init '1';

This feature is not utilized by the current version of the HPT.

3.4.3 Compiler Output

The compiler produces two output files. The one is the script that will be sent to the HPT and the other provides information to the GUI.

- **Script File**

The script file produced by the compiler is downloaded to the HPT. The HPT then, parses it and executes its commands.

- **Properties File**

This file contains information that is utilized by the GUI. Specifically, the **Configuration Clock**, the **Voltage Source**, the **Signals and Statics Mapping**, the **Number of Signals Used** as well as information about the **ports of the HPT** that are used.

3.5 Back-End

The Back-End of ReRun is the hardware platform developed by Dionissis Efstathiou. The platform is based on an AVR Microcontroller and communicates with the computer and GUI through the serial port using the RS-232 protocol. The HPT can be used for programming an FPGA device as well as test it.

To program a device, the user writes the appropriate script, compiles it, chooses the bitstream and presses the Start Button in the GUI. Then the compiled script and the bitstream are sent to the HPT through the serial port using the protocol developed. In the end the user receives a message that confirms the successful or unsuccessful configuration of the device.

In case of a test script, the user writes a script, compiles it and presses the Start Button. The script is then downloaded to the HPT using the same protocol. The HPT returns the values of the signals to the GUI, which stores them in a file and displays them in the form of arrays.

The crystal oscillator used for the AVR Microprocessor is 7.68 MHz.

The HPT can provide up to 24 signals to the user.

> **Caution!!** These signals are limited to 16 when in programming mode, because the 8 signals are used for loading configuration data to the FPGA. In test mode, these 8 signals can be used, but only as inputs to HPT.

The language is not AVR-specific, thus it is possible to develop another platform for programming and testing FPGAs, as long as it complies with the specifications set by the language.

3.6 Summary

In this chapter we referred to the architecture of ReRun. We saw a brief description of the system, as well as its features and attributes. Then we moved on to describing its structure, the tools used to create ReRun and some operating instructions. We presented the Programming and Testing Language (PTL) and its instructions. And finally we discussed the back-end of ReRun, also implemented at the Microprocessor and Hardware Laboratory, by Dionissis Efstathiou.

Chapter 4

Language

Chapter 4

Language

This chapter analyzes the language that was developed for ReRun in more depth. More specifically, the first two sections describe the Lexical and Syntactical Analyzers. The third section moves on to the Symbol Table. The fourth contains the type checking and error messages. And finally, the sixth section contains a summary of this chapter.

4.1 Lexical Analyzer

A lexical analyzer is an input procedure that reads blocks of input conforming to a specified set of patterns (tokens). A lexical analyzer reads from the current input (usually a file). It consumes characters until a complete token has been found. It then returns them to the Syntactical Analyzer. Usual tokens are keywords, variable identifiers, delimiters and operands. The procedure followed to build the lexical analyzer is depicted in Fig. 4.1.

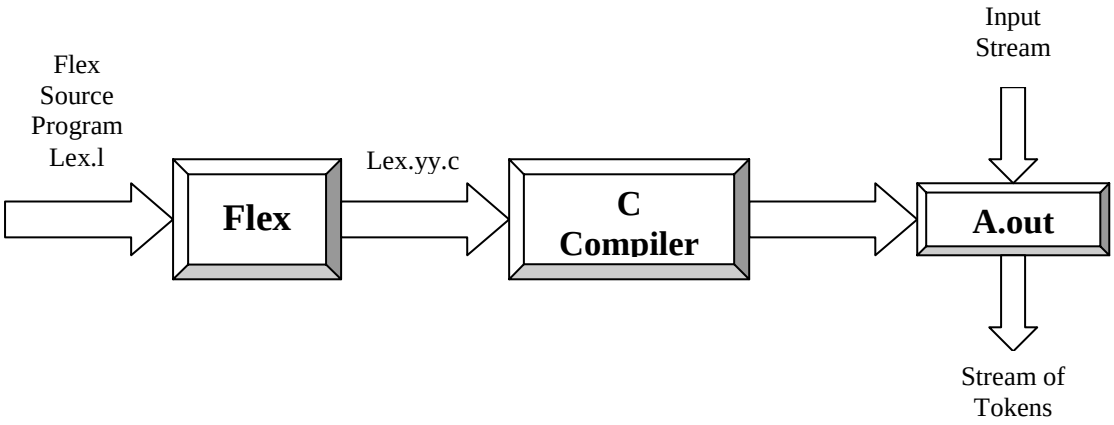

Fig 4.1 – The Lexical Analyzer

As mentioned in Chapter 3, the tool used to develop the lexical analyzer is GNU's Flex. Flex is Lexical Analyzer Generator in C/C++. It is based on UNIX's lex tool. Flex is relatively easy to use and is quite flexible. It can also cooperate with syntactical analyzer generators. The code is divided in three parts that are separated with the % symbol.

Part A

%

Part B

%

Part C

The first part contains comments in /* and */ and C/C++ code in %{ and %}, that will be embedded as is. For example:

```
%{
    #define TK_EOF      0
    #define TK_ID       1

    int lineCount = 0;
%}
```

The first part also contains mnemonic name definitions for character families or canonical expressions. For example:

```
NOT_DQUOTE      [^"]
identifier      {letter}({letter}|{digit})*
```

Finally it includes instructions to Flex and initial state definitions.

In the beginning of the second part one can write C/C++ code in %{ and }% to declare variables for the lexical analyzer function. The most important part is the rules. Rules are in the form of:

Canonical expression Action

For example:

```
clk {printf("clk");return TK_CLK;}
```

Comments are not allowed in the second part and the syntax is strict, so trivial mistakes can lead to different results.

Finally, the third part, as well as the % separator, is optional. It contains C/C++ code that is embedded as is and is usually used to define helpful functions that can be called from the lexical analyzer in the actions of the second part.

The lexical analyzer of PTL recognizes the following keywords:

- vs (The voltage source declaration) TK_VS
- clk (Configuration Clock declaration) TK_CLK
- manufacturer (The DUT vendor declaration) TK_MANUFACTURER
- family (The FPGA family) TK_FAMILY
- device (The device) TK_DEVICE
- int (Integer variable declaration) TK_INT
- signal (signal declaration) TK_SIGNAL
- program (Program Script declaration) TK_PROGRAM
- static (Static declaration) TK_STATIC
- test (Test Script declaration) TK_TEST
- map (Start of Signal and Static Mapping) TK_MAP
- start (Start of Script) TK_START
- end (End of Script) TK_END
- set (Change the value of a signal) TK_SET
- loadb (Load bytes) TK_LOADB
- loadkb (Load KBytes) TK_LOADKB
- get (Get the values of the signals on a port) TK_GET
- readbackb (Readback KBytes) TK_RDBKKB
- readbackkb (Readback Bytes) TK_RDBKB
- wait (Wait for a signal to get the specified value) TK_WAIT
- reverse (Toggle a signal between input and output) TK_REV
- for (Start of for loop) TK_FOR
- endfor (End of for loop) TK_ENDFOR
- low (Value for the Configuration Clock) TK_LOW
- high (Value for the Configuration Clock) TK_HIGH
- lsb (Load least significant bit first) TK_LSB
- msb (Load most significant bit first) TK_MSB
- nop (No Operation) TK_NOP

The lexical analyzer also recognizes the following symbols: (TK_LPA,) TK_RPA, { TK-LBRA, } TK_RBRA, , TK_COM, ; TK_TERM, = TK_EQ, <= TK_VASSGN, => TK_ASSGN, + TK_PLUS, - TK_MINUS, * TK_MULT, / TK_DIV and comments. There can be line comments and multi-line comments. Line comments start with // and multi-line comments start with /* and end with */. Finally, the lexical analyzer recognizes numbers and identifiers and returns their values to the syntactical analyzer.
Another function performed is keeping the line number of each token in a variable. This variable is later utilized by the syntactical analyzer for error messages.

4.2 Syntactical Analyzer

Syntactical analysis imposes a hierarchical structure on the program. This structure is specified by the rules of a context-free grammar. A syntactical phrase is introduced by giving one or more alternatives. An alternative specifies how to construct an instance of the phrase. It lists the members that build up the phrase, where such a member is either a token or the name of a phrase (a non-terminal).
Consider the rule to define statements:

```
statement : id ASSIGN expression
          | set signal state
          ;
```

For example, the first alternative specifies that if *D* is an id and if *E* is an expression then *D* := *E* is a statement.
The syntactical analyzer generator used to develop the language is GNU's Bison. In order for Bison to parse a language, it must be described by a context-free grammar. The most common way to describe rules is the Backus-Normal-Form (BNF). Any language expressed in BNF form is context-free. In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a symbol. Those which are built by grouping smaller constructs according to grammatical rules are called non-terminal symbols; those which can't be subdivided are called terminal symbols or token types. A piece of input corresponding to a single terminal symbol is called a token, and a piece corresponding to a single non-terminal symbol is called a grouping.
The Bison file is in the form:

%{
C/C++ declarations
}%
Bison declarations
%
Grammar rules
%
Extra C/C++ code

The Bison declarations are terminal and non-terminal symbol declarations. For example:

%token TK_ID
%type<number> expression

There can also be precedence and associativity declarations:

%left TK_PLUS
%right TK_EXPON

Symbol type declaration:

```
%union{
    int number;
    char *string;
}
```

The third part, the Grammar Rules has declarations of the general form:

left_part: right_part;

The left part being a non-terminal symbol and the right part, zero or more terminal and non-terminal symbols. There can be alternative right parts and they can have semantic actions. For example:

```
main : TK_START   {
            printf("Start of main program");
    } smts TK_END {
```

```
        printf("End  of program");
    };
```

The last part, as well as the % separator, is optional. It contains C/C++ code that is embedded as is. It can be used to define functions that can be called by the semantic actions.

The grammar of the language developed is presented below in BNF form:

```
<script> ::= <comment>
             <PT>
             <loadlm>
             <CLK>
             <VS>
             <declarations>
             <mapping>
             <main>

<declarations> ::=
             | <declarations> <declaration>

<declaration> ::= <var_def>
            | <static>

<comment> ::=
         | TK_MANUFACTURER TK_CHARCON TK_TERM TK_FAMILY
         TK_CHARCON TK_TERM TK_DEVICE TK_CHARCON TK_TERM

<var_def> ::= <type> <def_some> TK_TERM

<type> ::= TK_INT
      | TK_SIGNAL
<def_some> ::= <def_one>
          | <def_some> TK_COM <def_one>

<def_one> ::= TK_ID
```

```
<CLK> ::=
     |TK_CLK TK_NUM TK_LPA TK_FUN TK_RPA TK_TERM
     |TK_CLK TK_LOW TK_TERM
     |TK_CLK TK_HIGH TK_TERM

<VS> ::=
    |TK_VS TK_NUM TK_LPA TK_VUN TK_RPA TK_TERM

<PT> ::= <test>
    | <program>

<test> ::= TK_TEST TK_TERM

<program> ::= TK_PROGRAM TK_CHARCON TK_TERM

<loadlm> ::=
         |TK_LSB TK_TERM
         |TK_MSB TK_TERM


<static> ::= TK_STATIC TK_ID TK_STATE TK_TERM

<mapping> ::= TK_MAP TK_LBRA <maps> TK_RBRA

<maps> ::=
      | <maps> <map>

<map> ::= TK_ID <assgn> TK_NUM TK_TERM

<assgn> ::= TK_ASSGN
       | TK_VASSGN

<main> ::= TK_START <smts> TK_END
```

```
<compound> ::= TK_LBRA <cmp_smts> TK_RBRA;

<cmp_smts> ::=
          | <cmp_smts> <cmp_smt>

<cmp_smt> : <set>

<smts> ::=
      | <smts> <smt>

<smt> ::= <assignment>
     | <for_smt>
     | <set>
     | <load>
     | <get>
     | <compound>
     | <reverse>
     | <wait>
     | <readback>
     | <nop>

<assignment> ::= TK_ID TK_EQ <expression> TK_TERM

<expression> ::= <expression> TK_PLUS <expression>
            | <expression> TK_MINUS <expression>
            | <expression> TK_MULT <expression>
            | <expression> TK_DIV <expression>
            | TK_NUM
            | TK_ID

<for_smt> ::= TK_FOR <expression> <for_smts> TK_ENDFOR

<for_smts> ::=
          | <for_smts> <f_smt>
```

```
<f_smt> ::= <assignment>
      | <set>
      | <compound>
      | <reverse>
      | <wait>
      | <nop>

<wait> ::= TK_WAIT TK_ID TK_STATE TK_TERM

<set> ::= TK_SET TK_ID TK_STATE TK_TERM

<load> ::= TK_LOADB TK_NUM TK_TERM
      | TK_LOADKB TK_NUM TK_TERM

<get> ::= TK_GET TK_NUM TK_TERM

<reverse> ::= TK_REV TK_ID TK_TERM

<readback> ::= TK_RDBKB TK_NUM TK_TERM
          | TK_RDBKKB TK_NUM TK_TERM

<nop> ::= TK_NOP TK_NUM TK_TERM
```

4.3 Symbol Table

The symbol table constructed for the language is dynamically increasing list with pointers to the first and last node. Each node has a pointer to the next node and is a *struct* of type *node*. The symbol table stores the variables, signals and statics declared by the user. The struct has the following elements:

- A string containing the name of the variable, signal or static declared. The maximum length is currently defined to 20. This number is sufficient for naming variables, signals and static that a user may declare.

- An integer recording its value.
- An integer defining its type (1 is integer, 2 is signal and 3 is static).
- An integer that applies to signals and statics and defines whether the signal has been mapped. This field is 0 when this element has not been mapped and 1 if it has. For integers it is always 0 and can not be changed.
- Another integer that also applies to signals and statics, but not integers. This integer keeps the value of the pin to which the element has been mapped. The previous field is always check first to ensure that this one won't be read accidentally for an integer.
- Another, third integer exists in the symbol table and applies to signals and statics only.
- It defines whether the signal is an input or an output.
- Finally there is a pointer to the next element of the symbol table.

Besides the essential functions needed to insert, delete and find an element, some additional have been implemented.
One function named **error** has been created to produce better error messages than the ones produced from yyerror.
Another, **vartype,** returns the type of the element with the name that is passed as an argument.
The **getsignals** returns the current values of signals and statics to two global variables, so that they can be processed by set or other command.
The **getios** is similar to the **getsignals** with the difference that it returns whether the signals are inputs or outputs.
The **getnames** is used to write the variable names along with the pin to which each is mapped to a file that is later used by the Graphical User Interface.
And finally, the **ismapped** returns whether the signals or static passed as an argument to this function has been mapped.

4.4 Integrity Checks and Compiler Errors

The compiler performs the following integrity checks:

- **Type Checking**

  In this case the correctness of operands is checked. For example, the user can not set the

value of an integer or a static using the command set, or assign a value to a signal or static.

- **Check for unique declaration**
  This checks whether the name of a signal, variable or static is being declared more than one time. If so an error is produced.

Below the error messages and their explanation is presented:

- **Variable *x* already exists!**
  This error is produced when the user tries to declare a variable or signal with the name x but there is one with the same name already declared.
- **Unrecognized programming mode!**
  This means that a programming mode other than the currently supported *serial* or *parallel* is declared.
- **Static *x*, already declared!**
  The user has declared a static with the name x but there is one with the same name already declared
- **Signal or Static *x*, not declared!**
  The user has tried to map a signal or static with the name x that has not been declared to a pin.
- ***x*, has already been mapped!**
  The signal or static x has already been mapped.
- ***x*, is not a Signal or Static!**
  The user tried to map x to a pin, but it is not a signal or static.
- **Cable *x*, has been mapped to another Signal or Static!**
  The user tried to map a signal or static to a cable that is already mapped.
- **Static ,*x*, can not be an input**
  A static can only be mapped as output.
- **A signal cannot be mapped as an output on pins greater than 15, in test mode!**
  The reason this error message was produced is that test mode uses pins 16-23 only as inputs.
- **No such cable!**
  The current HPT supports a maximum of 23 pins and the user tried to map a signal or static to a pin greater than 23.

- **Unable to map beyond the 16th cable in programming mode!**
  Programming mode uses pins 16-23 for downloading configuration data to the FPGA, so these pins are not accessible.
- **Maximum number of signals reached!**
  The user has already mapped 24 signals which is the maximum and tries to map one more.
- **Error! Compound statement in compound statement!**
  A compound statement can not exist inside another compound statement.
- **Use more than one set command inside a compound statement!**
  A compound statement is used to change the values of signals with the set command simultaneously. It is meaningless to use only one set command.
- **Variable *x*, is not declared!**
  Trying to assign a value to a variable that has not been declared produces this error.
- **Error assigning, x!Illegal types**
  The user tried to assign signals or statics, instead of integers.
- ***x*, does not exist!**
  Expressions with variables that have not been declared, produces this message.
- **Addition between x and y is not defined**
  Adding signals and statics with variables or signals and statics is illegal and produces this error message.
- **Subtraction between x and y is not defined**
  The same as the previous but appears when the user subtracts.
- **Multiplication between x and y is not defined**
  The same as the previous but appears when the user multiplies.
- **Division between x and y is not defined**
  The same as the previous but appears when the user divides.
- **You can not iterate more than x times!**
  The maximum number of iterations is limited due to the HPT. Current x is 256 iterations, but can be changed through the macro MAXITER.
- **You can not store more than x bytes!**
  The maximum number of instructions in a for loop is limited due to small HPT storage. Currently x is 256 bytes. This limit can be changed through the MAXDATA macro. Consult the Instruction Opcodes and Sizes for more information.

- ***x* is not a signal!**
  The set instruction can only be used for signals. Using it on integers or statics produces this message.
- **You can not change the value of an input signal!**
  Input signals can not be set to a value. This can only be done if the reverse command is used first and then the set.
- **Signal *x*, not declared!**
  Signals must be declared to be used with the set instruction.
- **You can not use the loadb command in a Test Script!**
  The loadb command can only be used in a program script, this is the error message when the user tries to use it in a Test Script.
- **You can not load more than x bytes at a time!**
  The maximum value for a loadb is currently 256. To loadb more bytes, type another loadb command immediately after this one.
- **You can not use the loadkb command in a Test Script!**
  Same as the loadb error, but for Kbytes.
- **You can not load more than ",tmp," KBytes at a time!**
  Same as the loadb error, but for Kbytes.
- **The argument for get must be 0, 1, 2 or 3!**
  The get command can only take one of four possible arguments. This is the error message produced for any other case.
- **You must map signals to all ports to use 'get 0'!**
  It is important to have signals mapped to all ports of the HPT to use the get command.
- **You have not mapped any signals beyond 8!**
  You can not get the values of pins you have not mapped signals to.
- **You have not mapped any inputs to the Data Port!**
  You can not get the values of pins you have not mapped signals to.
- **Can not reverse a signal mapped to the Data Port!**
  Trying to reverse a signal mapped to the Data Port is not allowed.
- **Can not reverse a static!**
  Statics can only be outputs, so reversing them is not allowed.
- **You can not use the readbackb command in a Program Script!**
  Readbackb command can only be used in a test script.

- **You can not readbackb more than x bytes at a time!**
  The same limit with the load command applies to readbackb.
- **You can not use the readbackkb command in a Program Script!**
  Readbackkb command can only be used in a test script.
- **You can not readbackkb more than x KBytes at a time!**
  The same limit with the load command applies to readbackkb.
- **The msb instruction can not be used in a test script!**
  This command has no use in tests scripts, as there is no bitstream to send.
- **The lsb instruction can not be used in a test script!**
  The same as the previous error message, for the lsb command.
- **The get instruction can not be used in a for loop!**
  A get instruction can not be used inside for loops.
- **Argument for nop must be greater than 0!**
  Passing 0 as an argument to the nop instruction is not possible.

4.5 Summary

This chapter discussed development issues of the PTL, such as the process of constructing the lexical and syntactical analyzers and their structure. Then we moved on to the contents of the Symbol Table. And finally, we described the code integrity checks and the compiler errors.

Chapter 5

Graphical User Interface

Chapter 5

Graphical User Interface

This chapter analyzes the Graphical User Interface developed. The first section describes the Java classes. The second refers to the communication protocol. The third section contains information about its usage.

5.1 Graphical User Interface classes

As mentioned in Chapter 3, the GUI was developed using Sun's Forte for Java Community Edition Integrated Development Environment. As this denotes, the programming language used is Java. The JFC Swing classes were used for the graphical components and the Java Communications API for implementing the connection to the HPT.
A total of eight classes were developed. Each one is described in the following sections.

5.1.1 AppFilter Class

This class is a file filter. It is used when a File Chooser Window appears and only allows files that pass the filter to appear in the window. This class is used when the user chooses to open a script file or load a bitstream. Scripts have an .spt extension, so all other files are not shown. The case that the script file has another extension is covered. An option enables the File Chooser Window to print all the files.

5.1.2 ConfigurationPanel Class

This class creates a Configuration Panel with the port settings. Currently, the serial port is used so the parameters are for a serial port. The following parameters are set by this class:

- **Port Name**. This list contains the names of the ports and is dynamically created by scanning the available serial ports on the system.

- **Baud Rate**. This option refers to the speed at which the serial port can transmit and receive data. The Baud Rates selected were those supported by the HPT:

    1. 2400
    2. 4800
    3. 9600
    4. 14400
    5. 19200
    6. 28800
    7. 57600
    8. 115200

    The default Baud Rate is 9600.

- **Flow Control In/Out**. Flow control means the ability to slow down the flow of bytes in a wire. For serial ports this means the ability to stop and then restart the flow without any loss of bytes. The available options listed below apply to both Flow Control In and Flow Control Out:

        1. None
        2. Xon/Xoff In
        3. RTS/CTS In

    The default value is None.

- **Data Bits**. This is the number of bits used to represent each character. The available options are 5, 6, 7, 8. ReRun works with this option set to 8, which is also the default value.

- **Stop Bits**. The stop bits represent how many bits mark the end of a data block when using asynchronous data transmission. The available options are 1, 1.5, 2 and the default is 1.

- **Parity**. The parity adds an extra bit to each character, which is set or cleared based on the type of parity used (odd or even). The default choice is none, but the user can select odd or even.

5.1.3 Connection Class

The Connection and ReRun classes are the most important for the GUI. The Connection class implements the connection to the serial port and all the associated functions. More specifically the following methods are implemented:

- **openConnection**. This method opens the port represented by the CommPortIdentifier object. Gives the open call a timeout of 30 seconds to allow a different application to give up the port if the user wishes to. Then the parameters for the connection are set. If, for any reason they can't be set, the port is closed and an exception is thrown. After that the input and output streams for the connection are opened. If they can't be opened the port is closed and an exception is thrown. Then the notifyOnDataAvailable is set to true, to allow event driven input and break handling correspondingly and adds an ownership listener to allow ownership event handling. Finally the parameters for the connection are set with a starting Baud Rate of 2400, a reset is sent to the HPT and the GUI waits for a response to verify correct operation. If it gets one, the Baud Rate instruction is sent to the HPT with the desired Baud Rate and the GUI is ready to send the rest of the data. If it gets no response, it retries the operation a total of 3 times and if unsuccessful it produces an error message.

- **setConnectionParameters**. This method is called by the openConnection method to set the connection parameters (Baud Rate, Parity, etc).

- **closeConnection**. Takes the appropriate steps to close the connection.

- **isOpen**. Returns a Boolean that denotes the state of the connection to the port.

- **waitData/waitData1**. These two methods are invoked when a script requires feedback from the HPT. If the appropriate data is received, the script continues. If no data is received or is incorrect, the GUI sends a reset instruction to the HPT and waits for an acknowledge. If no data is received, the waitData method sends two more resets and then produces an error message. The waitData1 produces an error message if no data after the first reset.
- Finally, the **SerialEvent** Listener is implemented in the Connection class. This Listener waits for data from the serial port and depending on what is received, it performs specific actions. The FSM that is responsible for sending the script to the HPT, raises certain flags when some instructions are sent. For example, a get command waits for the values of signals and the GUI must stop sending data until they are received. The following instructions must receive feedback before the GUI resumes sending the script:
  - Reset
  - All get instructions
  - For loops
  - Readback instructions
  - CRC Checks

  The Reset and for loop instruction await one byte, 0x00 and 0x0C respectively. Get 1, get 2, get 3 instructions wait one byte that depends on the values of the signals on the HPT. And finally, get 0 waits for 3 bytes.

5.1.4 ConnectionException Class

This is constructs a Connection Exception with a detailed on no message. It is by the Connection class to throw exceptions regarding the connection.

5.1.5 Parameters Class

This is a class that stores the serial port parameters. The default constructor sets the parameters to no port, 9600 Baud Rate, no flow control, 8 data bits, 1 stop bit and no parity. There is also another constructor that sets all the parameters to the desired values. This class

also implements the methods for settings the parameters and getting the various parameters as strings or integers.

5.1.6 PortRequestedDialog Class

This class notifies the user that another application has requested the port, and then asks if they are willing to give it up. If the user answers "Yes" the port is closed and the dialog box is closed, if the user answers "No" the dialog box closes and no other action is taken.

5.1.7 ReRun Class

This is the basic class of the GUI. It creates the graphical components and implements their Action Listeners. This is where the “main” function is. Running this class first brings up a Splash Screen which remains for a few seconds or until the mouse clicks on it. Then the interface components are created and shown. This class also implements the Action Listeners for the Buttons, Text Areas and Menus. The following menu options are created:

Menus

File Menu

- **Open Script**. This brings up a File Chooser Window that allows the user to select the script file. Only .spt files are displayed .After the file is selected, it is opened in the first Text Area where it can be viewed or edited. When a script file is loaded, the Save Script menu option and the Compile Script button are enabled.
- **Load Bitstream**. This option also brings up a File Chooser Window. This time the user can select a bitstream file. The name of this file is displayed along with a message on the third Text Area. Selecting a bitstream file is necessary for a program script. If the user tries to download a program script without choosing a bitstream, an error message is popped up.
- **Save Script**. The Save Script menu option saves the script.
- **Save As**. The Save As option brings up a File Chooser Window, where the user must specify a name and extension with which the file will be saved. This option can be used to save a new script or an existing one with a different name.

- **Exit**. This option closes the connection to the serial port if it is open and closes the GUI.

Port Settings

- **Port Settings**. This menu option brings up a window with the Settings for the Serial Port.

Help

- **Help**. This option provides the user links to online help.
- **About**. This produces an about box.

Buttons

- **Download Script**. This button starts the process of downloading the script to the HPT. This part is the heart of the GUI.

  Test scripts require only one file, the script, while program scripts also require the bitstream. Each type of script calls a different method. Each method implements a different Finite State Machine, due to the fact that certain instructions do not apply to both scripts. This method handles the data from get commands. It prints it to the screen and saves it to a file (*output.txt*). In case of scripts meant for readback. The data is saved to the *readback.bit* file. The CRC Check is also performed here every about 256 bytes.
- **Compile Script**. This button creates a new Thread and compiles the script. The compilation results are displayed on the third Text Area. Here the *props.gui* file produced by the compiler is processed. The GUI reads the signals' and statics' names and mapping, the type of script and prints them to the middle text area and the output.txt file.
- **Reset**. The Reset Button sends a reset instruction to the HPT and clears variables that have been set to specific values.

Check Box

- **Process Bitstream**. Xilinx bitstreams contain a header with information about the design. This header must be removed before the bitstream is ready to be sent. This check box can be used for such purposes.

A number of other methods are implemented here, openFile, saveFile, shutdown, restart, etc. These methods are used to simplify and modularize the code.

5.1.8 SplashWindow Class

The SplashWindow class pops up a Splash Screen when the GUI is run. This is achieved by creating a thread that runs for time equal to that specified when the constructor is called. If a mouse click on the Splash Screen is detected, it disappears, but the thread keeps running until the specified time has elapsed. This does not affect the performance, as it does nothing.

5.2 Communication protocol

The GUI communicates with the HPT through a protocol. This protocol is substantially a Finite State Machine (FSM). This FSM is different in the case of a test script and a program script. For example test scripts can not contain the loadb or loadkb instructions. Such instructions produce an error and the GUI is reset. In each case a different function is called, the function for test scripts needs the path of the script, while the one for program scripts requires both the script and bitstream files.
In both cases the **props.gui** file is read and depending on the properties the FSM is dynamically modified. For example a set instruction has one argument if no signals are mapped on pins beyond 8, otherwise it has two arguments. This means that another byte must be read from the script and sent to the HPT.
Most instructions are simply sent to the HPT, but some require feedback or acknowledge. The following instructions must wait for data from the serial port:

- **Get**. All get instructions wait for one byte from the serial port, with the exception of get 0 (get data from all ports), that waits for 3 bytes.
- **Readback.** These instructions receive the bitstream of an FPGA and store it in a file. So the GUI waits until all the requested data has been read back to continue.
  Reset. In case of a reset the HPT must send back one byte (0x00) to acknowledge the reset. If this does not happen, the reset is sent a total of 3 times. If nothing happens, the user is notified with an error message.

In case the GUI does not receive data for a specific amount of time, the programmer is reset, as there is probably a synchronization or data transfer error. Currently the time limit is 3 seconds. When the GUI receives data from a get instruction, the following actions are performed. First it writes the data to a file that has been created. This files is named **outputs.txt** and the first line contains the names of the signals used, separated by |. Each get the user writes creates a new line to this file. The values of signals that were not asked for are n/a (not available). This file is also printed on the second text area of the GUI.

When the interface starts communicating with the HPT, it starts at a BAUD RATE of 4800. Then it sends the new BAUD RATE and resets the HPT. When and if the HPT returns the appropriate byte (0x00), the script is downloaded, otherwise an error message notifies the user.

The implementation of a CRC Check was deemed unnecessary. There were two reasons for this decision. First, the user can enable CRC from the vendor's development tools. This CRC is embedded in each frame of the configuration bitstream and if the check fails, the configuration fails and must restart. The same thing would happen even if we had implemented a CRC, because the HPT does not support error recovery. This brings us to the second reason. Taking into consideration that wrong data causes the configuration to fail, a CRC would be meaningful, if the HPT could store it, perform the CRC and then load it. But this is not possible in the current version of the HPT, due to insufficient memory.

5.3 Usage Instructions

The Graphical User Interface developed for ReRun is quite easy to use. Starting, the user has the following options:

- Change the port settings
- Open a script file.
- Write a script in the first Text Area and save it.
- Load the bitstream file. If the script opened is not a program script, this does not affect the transmission of data.

Opening a file enables the Save Script menu option and the Compile Script menu option and button. From here the user can compile the script. After doing so, the compiler messages are

displayed on the third text area. They can either be errors or the message that the compilation was successful.
Having compiled the script and set the port settings, the user can download the script to the HPT. This is done by pressing the Download Script Button.
In case the user is programming a device, the Process Bitstream button is enabled. If the device vendor is Xilinx, this button must be pressed before downloading the script.
The user can view the results of any get commands in the middle text area or in the *output.txt* file. This file is created in the directory ReRun was installed. If the user is writing a readback test script, the data read back is saved in the *readback.bit* file which is also in the directory ReRun is installed.
Finally, the user can press the Reset button at any time. This is useful, when an unexpected error has occurred and the GUI can not communicate with the HPT. Pressing Reset, will reset the HPT.

5.4 Summary

This chapter discussed development issues of the Graphical User Interface. We analyzed the Java classes created. Then we described the communication protocol between ReRun and the Hardware Programmer and Tester. And finally, usage instructions for the GUI were presented.

Chapter 6

Examples of Usage

Chapter 6

Examples of Usage

This chapter contains examples of PTL scripts used to test the system. The tests performed were the programming and testing of 3 FPGAs (two Xilinx and one Altera). The successful configuration was confirmed by viewing the status of the DONE signal (for the Xilinx FPGAs) and CONF_DONE and INIT_DONE signals (for the Altera FPGA). The status of the signals was viewed both on a logic analyzer and on the GUI.
The first section contains the script used to program Xilinx's XC3042-50PC84. The section refers to the programming of Xilinx's XC4010XLPC84. The third contains the PTL code for Altera's Flex 8000 (EPF8282ALC84-3). Finally the fourth section presents a test script.

6.1 Example 1 – Script for programming a Xilinx XC3042-50PC84

The configuration process for the XC3000 family is described in Chapter two. The following script will configure the FPGA using Slave Serial configuration mode. This mode requires three control, three configuration mode select and two status signals. The first control signal is the configuration clock (CCLK) and is produced by the HPT when there is a load instruction. The next is DIN and is used for loading the configuration data. These two signals need not be declared in the script as they have a specific purpose and are controlled when a load command is issued. The configuration select signals are MM0, MM1 and MM2 and must all be set high to select Slave Serial mode. From the remaining three signals, the RESET controls the initialization of the configuration, while the INIT and DONE/PROG are status signals. A high-low-high transition on the RESET signal initiates configuration. The INIT signal is initially high, then low to indicate that the configuration memory is being cleared and then again high. If an error occurs, the INIT will go low. The DONE/PROG signal is low during configuration and it goes high if the configuration is successful. The following script was used to program the XC3000 FPGA. Only three signals were declared, reset (connected to the RESET pin of the FPGA), init (connected to the INIT pin of the FPGA) and done (connected to the DONE/PROG pin of the FPGA). Init and done were declared as inputs to the HPT and reset as an output. After the end of the configuration, the init and done signals were, indicating a

successful configuration. The script is written for Xilinx and was thus downloaded with the Process Bitstream option in the GUI.

```
program "serial";   // Set the programming mode to Serial

msb;                    // The MSB will be loaded first

clk high;               // The configuration clock has a
                        // minimum low time

signal reset,done,init; // Three signals will be used
static mm0 ‘1’;         // The mode pins should be statics
static mm1 ‘1’;         // For Slave Serial they must be
static mm2 ‘1’;         // set to 111
map                     // Signal Mapping
{
     prog <= 0;     // prog is an input and connected to
                    // cable 0 of the HPT
     reset => 1;    // reset is an output and connected to
                    // cable 1 of the HPT
     init <= 2;     // init is an input and is connected to
                    // cable 2 of the HPT
     mm0 => 3;      // The mode pins are statics and
     mm1 => 4;      // can only be mapped as
     mm2 =>5;       // outputs from the HPT
}

start                   //Start of script
     set reset '1';     // Generate a high-low-high
     set reset '0';     // pulse on the reset
     set reset '1';     // signal

     loadkb 3;          //
     loadb 256;         // Load a total of 3,927 bytes
```

```
32      loadb 256;          // This is the size of the
33      loadb 256;          // bit file
34      loadb 87;           //
35
36      get 1;              // Get back the values
37 end
```

The configuration was successful, as not only did the status signals go both high, but also the design downloaded, operated as expected. This was the first FPGA programmed by ReRun. The FPGA was on a GERM Board with various designs. The FPGA was programmed with Baud Rates up to 115200 which is the maximum. For more information on the VHDL code downloaded to the device, refer to the HPT report.

6.2 Example 2 - Script for programming a Xilinx XC4010XLPC84

The script described below configures the FPGA using Slave Serial mode. The configuration of the XC4000 device required three control, three configuration mode select and two status signals as the XC3000. The only difference is that the reset pin is called PROGRAM and the DONE/PROG, is called DONE. Consequently, the programming script is very similar to the one in the previous section.

```
 1 program "serial";       // Set Serial Programming Mode
 2
 3 msb;                    // Load LSB of configuration data
 4                         // first
 5
 6 clk high;               // The clock has a maximum low
 7                         // time
 8
 9 signal prog,init,done;  // Declare the signals that will
10                         // be used
11 static mm0 ‘1’;         // The mode pins should be statics
12 static mm1 ‘1’;         // For Slave Serial they must be
```

```
13 static mm2 ‘1’;          // set to 111
14 map                     // Signal Mapping
15 {
16      init <= 0;         // init is an input to the HPT
17      prog => 1;         // prog is an output from the HPT
18      done <= 2;         // done is an input to the HPT
19      mm0 => 3;          // The statics are mapped and
20      mm1 => 4;          // their values can not change
21      mm2 => 5;          // while the script is executed
22 }
23
24 start                  // Start of Script
25      set prog '1';      // Produce a high-low-high pulse
26      set prog '0';      // on the prog signal to initiate
27      set prog '1';      // configuration
28
29      nop 13;            // Wait until the FPGA clears its
30                         // configuration memory
31      loadkb 34;         //
32      loadb 256;         // Load the configuration data
33      loadb 256;         // (Size of bitstream is
34      loadb 172;         //  35500 bytes)
35
36      get 1;             // Get the values of the signals
37 end
```

The XC4010XL-PC84 FPGA was configured with Line's Round Movement using Bresenham's Line algorithm. The VHDL code for this design was implemented by Giannis Sourdis and the FPGA configured was on an XS-40 Board with a VGA output. The result was visible on a PC Monitor. This FPGA was programmed with Baud Rates up to 115200 which is the maximum. The MultiLinx Download cable, has a maximum Baud Rate setting of 57600.

6.3 Example 3 – Script for programming an Altera Flex 8000

The following script is used to program an Altera Flex 8000, specifically, the EPF8282ALC84-3. The configuration mode that will be used is Passive Serial. The signals used as outputs is are nConfig and the three configuration mode select signals. The ConfDone and nStatus are indicative signals. The nS/P signal must be set low, MSEL0 high and MSEL1 low to enable Passive Serial configuration mode. The Altera devices receive configuration data LSB first, this is declared in line 4 of the script. The clk instruction takes the argument high, which is the default value, if this instruction is omitted. Then we declare the signals. The only output is nConfig, as the mapping defines. The other two, are used for monitoring the configuration process. To initiate configuration the nConfig signal is given a high-low-high pulse, this is done in lines 12-14. Then we use the nop instruction to give the nStatus enough time to be high. Then we load the configuration data to the device. The size of the raw binary file is 5kb. The serial data loading continues until ConfDone goes high, indicating that the device is configured. After the last byte of data is loaded, the DCLK pin is clocked another 10 times until the Flex 8000 releases the ConfDone and initializes the device. Finally, we use the get instruction, to view the state of the signals and confirm the success or failure of the configuration.

```
 1 program "serial";          // Set Serial Programming
 2                            // Mode
 3 clk high;                  // The clock has a maximum
 4                            // low time
 5 lsb;                       // Load LSB of configuration
 6                            // data first
 7 signal nConfig,nStatus,ConfDone; // Declare the signals
 8                                  // that will be used
 9 static nSP ‘0’;            // Set the mode pins to
10 static MSEL0 ‘1’;          // 010 to enable Passive
11 static MSEL1 ‘0’;          // Serial configuration mode
12
13 map              // Signal Mapping
14 {
15 ConfDone <= 0;   // ConfDone is an input to the HPT
```

```
16 nStatus <= 1;    // nStatus is an input to the HPT
17 nConfig => 2;    // nConfig is an output from the HPT
18 nSP => 3;        // The mode pins are statics and
19 MSEL0 => 4;      // can only be mapped as
20 MSEL1 => 5;      // outputs
21 }
22
23 start                // Start of script
24
25   set nConfig '1';    // Produce a high-low-high
26   set nConfig '0';    // on the nConfig to
27   set nConfig '1';    // initiate configuration
28
29   nop 13;             // Sent 13 useless bytes
30   loadkb 5;           // Load 5 Kb of configuration data
31   loadb 5;            // Load 5 more bytes to initialize
32                       // the FPGA.
33
34   get 1;              // Get the values of the signals
35
36 end                   // End of Script
```

This script has been executed at Baud Rates up to 115200 and the configuration was successful. This was confirmed from both the GUI and a logic analyzer. A custom board was created for testing the design. The board had a seven segment display, driven by the FPGA. For more information regarding the VHDL code loaded and extra tests to ensure correct configuration, refer to the HPT report.

6.4 Example 4 – Test Script

In this section we present a test script written to verify the correct configuration of a device with a 4-bit counter. This script presumes that the FPGA has already been configured. First the user must connect the flying wires to the device. The signals used will be clk (connected to the pin defined as the clock by the design), bit0, bit1, bit2 and bit3 of the counter. The following

script will run the design for 5 clock cycles then get the values of the pins, then run it for another 10 and again return the results.

```
 1 test;
 2
 3 signal clk, bit0, bit1, bit2, bit3;
 4
 5 map
 6 {
 7 clk => 0;         // The clock is an output and is
 8                   // connected to cable 0
 9 bit0 <= 1;        // The bit0 is an output and is
10                   // connected to cable 1
11 bit1 <= 2;        // The bit1 is an output and is
12                   // connected to cable 2
13 bit2 <= 3;        // The bit2 is an output and is
14                   // connected to cable 3
15 bit3 <= 4;        // The bit3 is an output and is
16                   // connected to cable 4
17 }
18
19 start                 // Start of script
20
21   for 5               // Iterate 5 times
22        set clk ‘1’;   // Create one
23        set clk ‘0’;   // clock pulse
24   endfor
26
25   get 1;              // Get the values
26
27   for 10              // Iterate 10 times
28        set clk ‘1’;   // Create one
29        set clk ‘0’;   // clock pulse
30   endfor
```

```
31
32 get 1;                    // Get the values
33 end                       // End of script
```

This is a relatively simple script, but it demonstrates the basic techniques of testing designs.

Chapter 7

Conclusions and Future Work

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In the early chapters of this document, we discussed the need for a Universal Programmer and Tester as well as the configuration process and modes. Then we moved on to describing the structure of ReRun, analyzing the language and the Graphical User Interface that were developed. Finally, we saw some tests that were performed.

The results of this thesis were more than satisfactory. Three FPGAs were successfully programmed, two Xilinx FPGAs and one Altera FPGA. By doing so, we achieved our initial goal, to make ReRun a Universal Run-Time Environment for FPGA programming.

Also, the FPGAs were programmed up to a maximum Baud Rate of 115200. This is quite important achievement, considering that the speed matched Altera's MasterBlaster and surpassed the 57600 maximum Baud Rate for the MultiLinx Cable!

Finally, the system developed is modular, which makes it easy to upgrade. Future work is discussed in the next section.

7.2 Future Work

The language can be upgraded in many ways. An array type can be developed along with instructions to use its values as output signals. The instructions can be easily changed so that the parser produces a different output to accommodate for the needs of an upgraded version of the HPT. More instructions, such as ifs, can be added to provide additional functionality. Finally, the parser output can become even more abstract to become completely independent of the current HPT.

The modular structure of the Graphical User Interface allows it to be upgraded easily. The communication with the HPT is currently implemented using the serial port. This module can be changed with one for a parallel or USB port. The Java Communications API provides support for serial and parallel, but in the future USB port will be supported.

Another improvement would be the implementation of a CRC Check. This is relatively easy in the case of a CRC 32, because the java.util. library has a CRC32 class.
Finally, another module that can be upgraded is the one printing the values of signals. This information is currently displayed in text. A graphical representation would be more appropriate.
Finally, more scripts could be written to program additional FPGAs.

Appendix

Appendix

Appendix A - Language Grammar

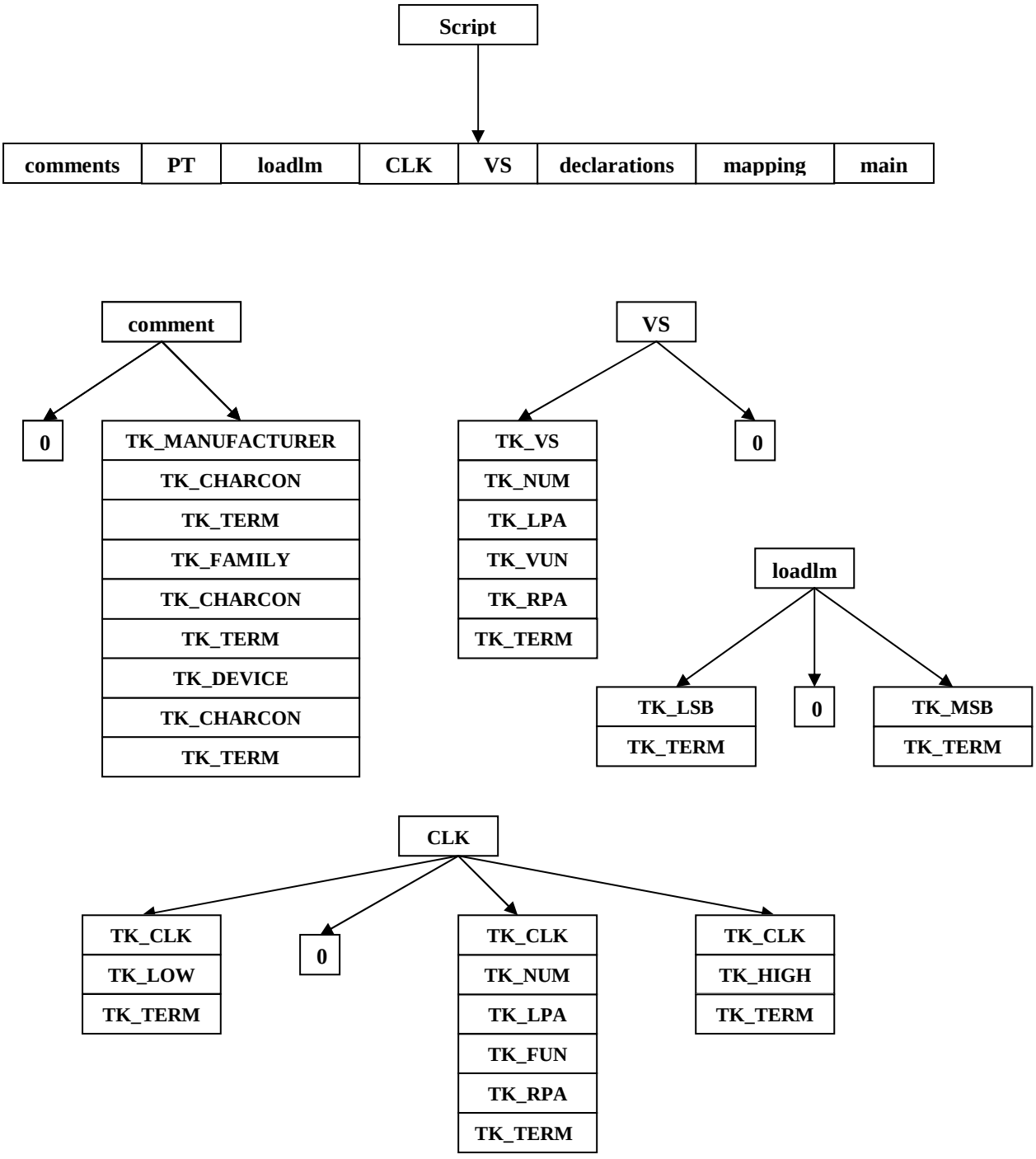

Figure Appendix.1 Language Grammar (continued)

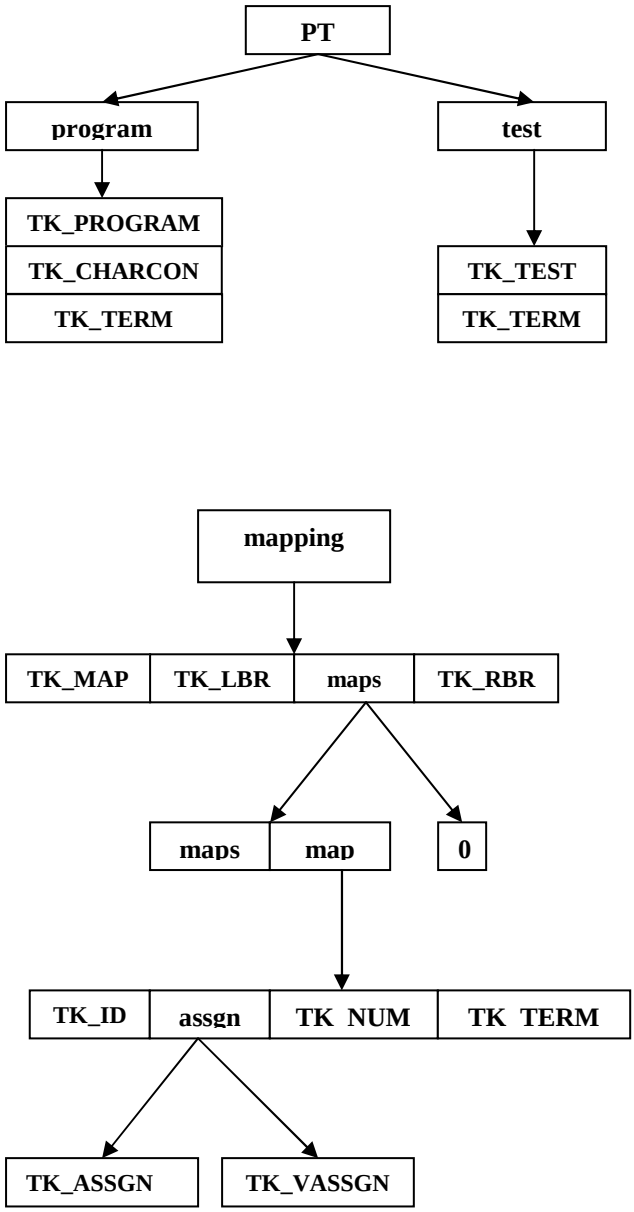

Figure Appendix.1 Language Grammar (continued)

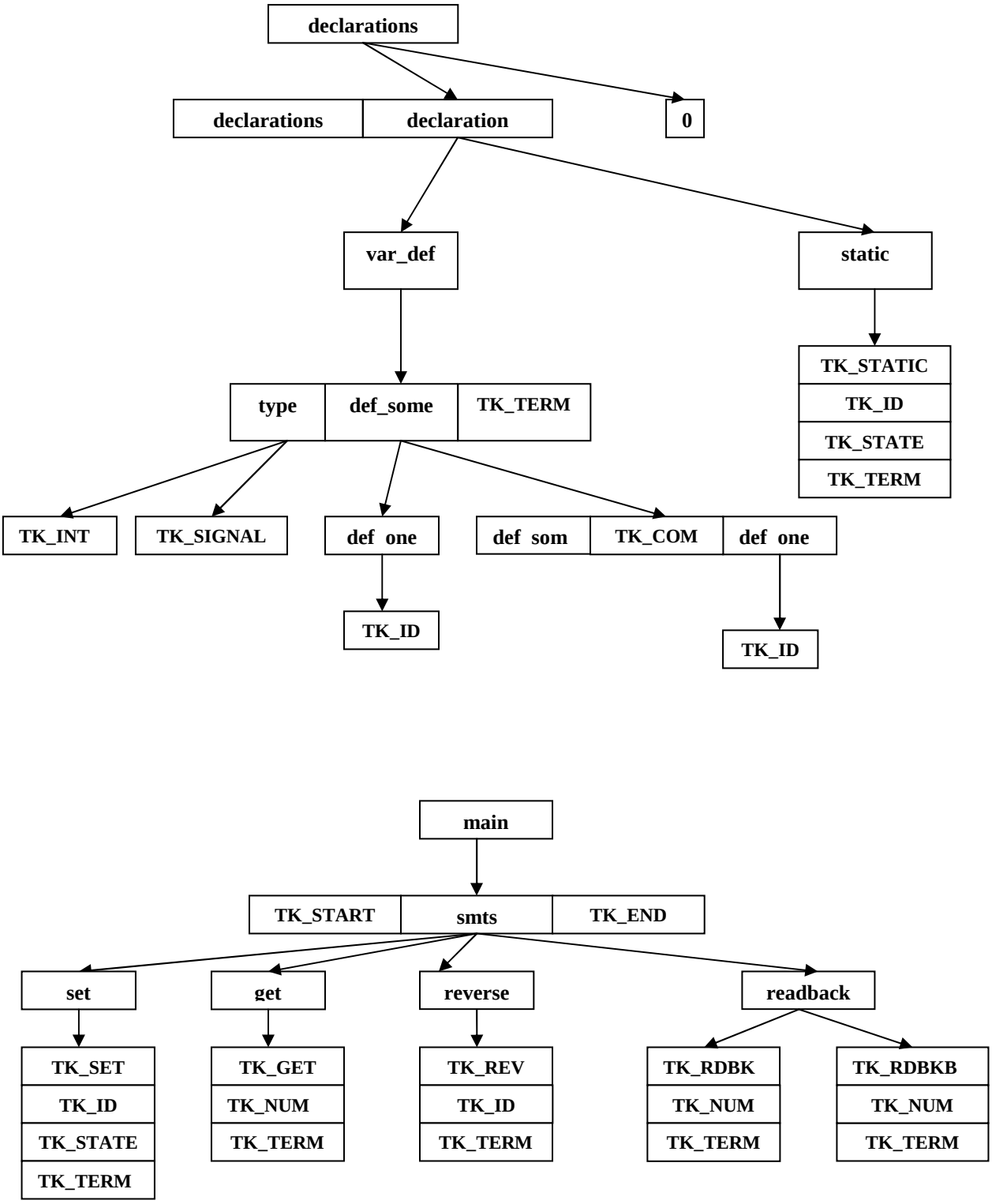

Figure Appendix.1 Language Grammar (continued)

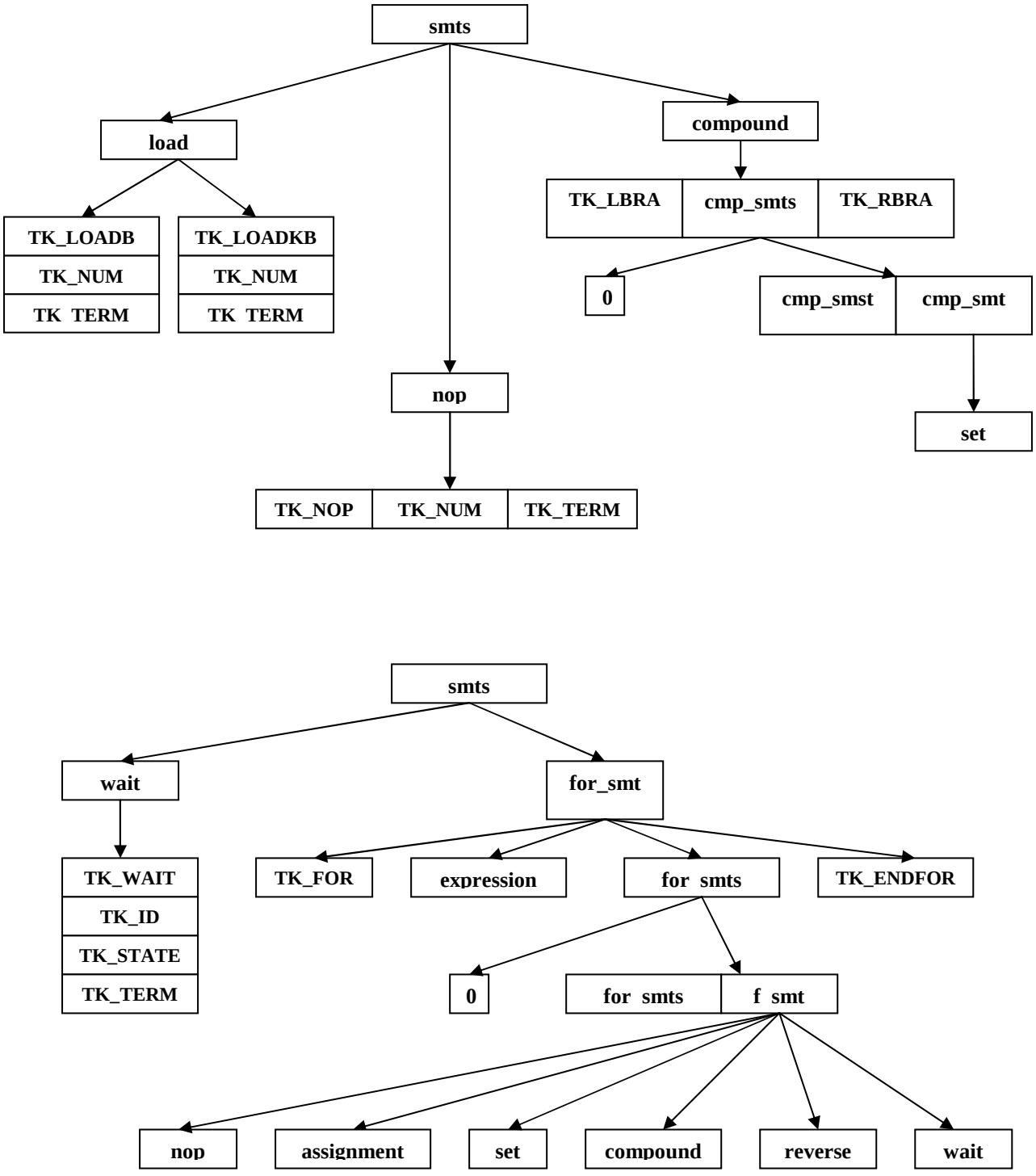

Figure Appendix.1 Language Grammar (continued)

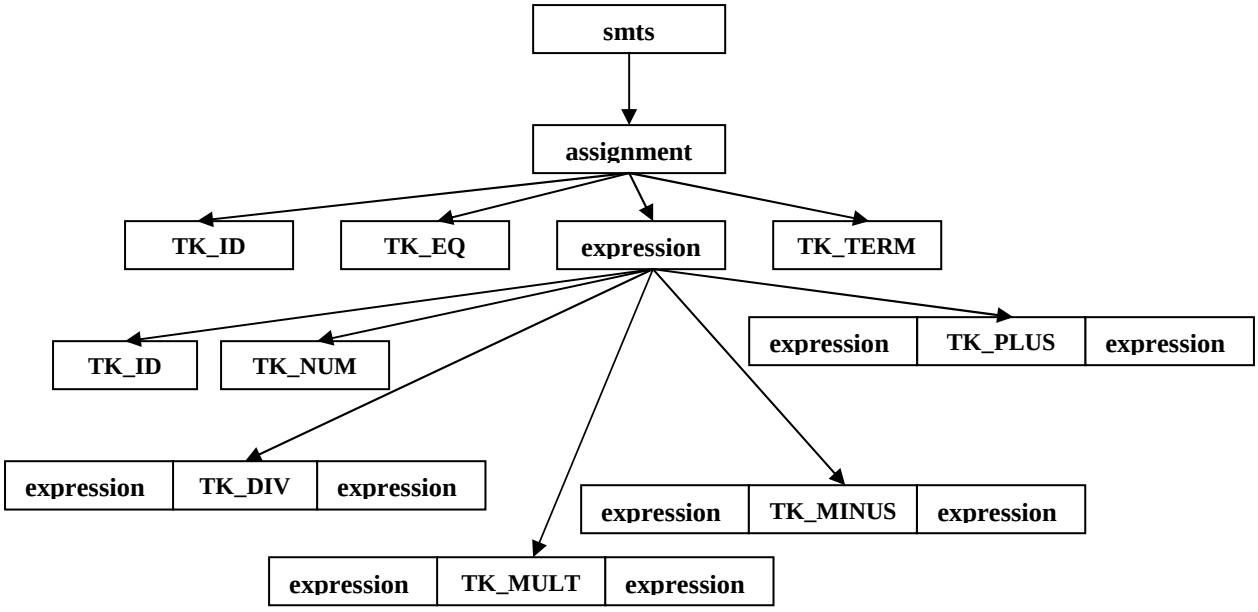

Figure Appendix.1 Language Grammar

Appendix B – Installing ReRun

The examples in this document assume that you have installed in your C: partition. More specifically we assume that you unzipped the javacomm20-win32.zip file in C:\commapi and your JDK installation is in C:\jdk1.3.1_01. If you have installed JDK in another location or unzipped javacomm20-win32.zip in another location modify the example commands appropriately.

1. Install the Java 2 SDK, v 1.3.1_01 in your computer. The file is **j2sdk-1_3_1_01-win.exe**. After the installation is complete Unzip the file **javacomm20-win32.zip**. This will produce a hierarchy with a top level directory commapi.

2. Copy win32com.dll from c:\commapi\ to your <JDK>\bin directory.

C:\>copy c:\commapi\win32com.dll c:\jdk1.3.1_01\bin

If you are using Windows you must also copy the win32com.dll to another directory, the C:\Program Files\Javasoft\JRE\bin.

C:\>copy c:\commapi\win32com.dll C:\Program Files\Javasoft\JRE\bin

Note! If you can not find the win32com.dll file, remember that dll files are usually hidden and that you must enable the option view hidden and system files in Windows.

3. Copy comm.jar file from c:\commapi\ to your <JDK>\lib directory.

C:\>copy c:\commapi\comm.jar c:\jdk1.3.1_01\lib

If you are using Windows must also copy the comm.jar to your C:\Program Files\Javasoft\JRE\lib directory.

C:\>copy c:\commapi\comm.jar C:\Program Files\Javasoft\JRE\lib

4. Copy javax.comm.properties, also from c:\commapi\ to your <JDK>\lib directory.

C:\>copy c:\commapi\javax.comm.properties c:\jdk1.3.1_01\lib

The javax.comm.properties file must be installed. If it is not, no ports will be found by the system.

If you are using Windows, again you must copy the javax.comm.properties to your C:\Program Files\Javasoft\JRE\lib directory.

C:\>copy c:\commapi\javax.comm.properties C:\Program Files\Javasoft\JRE\lib

5. Add comm.jar to your classpath.

If you don't have a classpath defined:

C:\>set CLASSPATH=.;c:\jdk1.3.1_01\lib\comm.jar
If you already have a classpath defined:

C:\>set CLASSPATH=c:\jdk1.3.1_01\lib\comm.jar;%classpath%

In Windows, this must be done from the Environmental tab of the the System Properties window (Control Panel->System).

6. Set the JAVA_HOME to your <JDK>\ directory. If you don't have JAVA_HOME defined:

C:\>set JAVA_HOME=c:\jdk1.3.1_01\
Add <JDK>\bin to your path.

In Windows the same procedure as the previous step must be followed.

C:\>set PATH=c:\jdk1.3.1_01\bin\;%path%

7. Create the directory you wish to install ReRun.

8. Copy the following .java files in that directory:

AppFilter.java
ConfigurationPanel.java
Connection.java
ConnectionException.java
MainWindow.java
Parameters.java
PortRequestedDialog.java
SplashWindow.java

9. Compile the previous files by typing:

javac *.java

10. The next step is copying the compiler to the directory you installed ReRun. The compiler is the file **rerun.exe**.

11. Finally, the Splash Screen file must be placed in the same directory. The name of the file is **mhl.jpg**.

Appendix C – Instruction Opcodes

CNTRL_8_INST ;Use one port of the AVR
(1 byte) opcode: "0x00"

CNTRL_16_INST ;Use two ports of the AVR
(1 byte) opcode: "0x01"

CNTRL_MAP_INST ;Defines Input and Output Signals
(2/3 bytes) opcode: "0x02"

CLEAR_BITS_INST ;Clear Bits
(2/3 bytes) opcode: "0x03"

SET_BITS_INST ;Set Bits
(2/3 bytes) opcode: "0x04"

DATA_SERIAL_INST ;Load Data Serially
(1 byte) opcode: "0x05"

DATA_PARALLEL_INST ;Load Data Parallel
(1 byte) opcode: "0x06"

RESET ;Reset
(1 byte) opcode: "0x07"
Waits for : 0x00 to ACK

PROG_BYTE_INST ;Send Programming Data byte(s)
(2 bytes) opcode: "0x08"

PROG_KBYTE_INST ;Send Programming Data Kbyte(s)
(2 bytes) opcode: "0x09"
CTRL_BITS_INST ;Send Entire Byte(s) to control pins

(2/3 bytes) opcode: "0x0A"

FOR_LOOP_START_INST ;Start of for loop
(3 bytes) opcode: "0x0B"

FOR_LOOP_END_INST ;End of for loop
(1 byte) opcode: "0x0C"

NOP_INST ;No Operation Instruction
(1 byte) opcode: "0x0D"

BAUDRATE_2400 : ;Set Baud Rate to 2400
(1 Byte) opcode:"0x0E"

BAUDRATE_4800 : ;Set Baud Rate to 4800
(1 Byte) opcode:"0x0F"

BAUDRATE_9600 : ;Set Baud Rate to 9600
(1 Byte) opcode:"0x10"

BAUDRATE_14400 : ;Set Baud Rate to 14400
(1 Byte) opcode:"0x11"

BAUDRATE_19200 : ;Set Baud Rate to 19200
(1 Byte) opcode:"0x12"

BAUDRATE_28800 : ;Set Baud Rate to 28809
(1 Byte) opcode:"0x13"

BAUDRATE_57600 : ;Set Baud Rate to 57600
(1 Byte) opcode:"0x14"

BAUDRATE_115200 : ;Set Baud Rate to 115200

(1 Byte) opcode:"0x15"

READBACK_KBYTE ;Reads back Kbytes
(2 Bytes) opcode:"0x16"

Waits for : (Byte 2) KBytes

READBACK_KBYTE ;Reads back Bytes
(2 Bytes) opcode:"0x17"

Waits for : (Byte 2) Bytes

Test Script ;Defines that this is a test script
(1 byte) opcode:"0x18"

GET_ALL_PORTS ;Asks for the values of all ports
(1 byte) opcode:"0x19"

Waits for : 3 Bytes (CTRL0,then CTRL1 and then CTRL_DATA)

GET_CNTRL0 ;Get values for signals mapped to the first 8 pins
(1 byte) opcode:"0x1A"

Waits for : 1 Byte

GET_CNTRL1 ;Get values for signals mapped to pins 8-15
(1 byte) opcode:"0x1B"

Waits for : 1 Byte

GET_DATA ;Get values for signals mapped to pins 16-23.
(1 byte) opcode:"0x1C"

Waits for : 1 Byte

CLK_LOW ;Defines that the FPGA receives data
;on the falling edge of the clock
(1 byte) opcode:"0x1D"

CLK_HIGH ;Defines that the FPGA receives data
;on the rising edge of the clock
(1 byte) opcode:"0x1E"

LSB ;Load the MSB of the bitstream first
(1 byte) opcode:"0x1F"

MSB ;Load the LSB of the bitstream first
(1 byte) opcode:"0x20"

Appendix D – ReRun File Structure and Size

The files of ReRun are:

Java Classes	Size (Bytes)	Number of Lines
AppFilter.java	744	38
ConfigurationPanel.java	7,251	222
Connection.java	37,685	901
ConnectionException.java	385	21
Parameters.java	7,371	319
PortRequestedDialog.java	1955	66
ReRun.java	53,278	1385
SplashWindow.java	2,087	56
Compiler Files		
lex.l	1,926	83
synt.y	26,031	1,333
Total	138,713	4,424

References

References

[1]. Sun Microsystems. *Java Platform.*2002

[2]. Atmel Corporation. *AVR-8 Bit RISC Microcontroller.*

[3]. Xilinx.

[4]. Xilinx. *Spartan-II*. 2001

[5]. Xilinx. *Virtex*. 2001.

[6]. Compaq. *PCI Pamette v1*. 2002

[7]. Xilinx. *Spartan-II Devices*. 2001.

[8]. Xilinx. *MultiLinx Download Cable*. 2000

[9]. Altera Corporation.

[10]. Altera Corporation. MasterBlaster *Serial/USB Communications Cable*. 2001

[11]. Lucent Technologies.

[12]. Xilinx. *XC4000E and XC4000X Series Field Programmable Gate Array.* 1999.

Xilinx. *XC3000 Series Field Programmable Gate Arrays (XC3000A/L, XC3100A/L).* 1998.

Xilinx. *XC5200 Series Field Programmable Gate Arrays*. 1998.

Xilinx. *XAPP176: Spartan-II FPGA Family Configuration and Readback.* 1999.

Xilinx. *XAPP138: VIRTEX Configuration and ReadBack.* 1999.

Xilinx. *XAPP501: Configuration Quick Start Guidelines*. 2001.

Xilinx. *XAPP090: FPGA Configuration Guidelines*. 1997.

[13]. Altera Corporation. *ALTERA Data Book 1998*, 1998.

Altera Corporation. *An 116: Configuring SRAM-based LUT devices. 2001*

Altera Corporation. *An 59: Configuring FLEX 10K Devices*, 1995.

Altera Corporation. *FLEX 10K Embedded Programmable Logic Family*, 1998.

Altera Corporation. *ALTERA Programming Hardware Data Sheet*, 1998.

[14]. Xilinx. *XC3000*. 1998.

[15]. Xilinx. *XC4000*. 1999.

[16]. Xilinx. *XC5200*. 1998.

[17]. Xilinx. *Virtex Configuration Guide*. 2000.

[18]. Compaq. *PCI Pamette v1 Overview*.

[19]. Compaq. *PCI Pamette v1 PamDC.*

[20]. Xilinx. *Xilinx Hardware InterFace.*

[21]. Altera Corporation. *Signal Tap Embedded Logic Analyzer*.

[22]. Altera Corporation. *Quartus II Software.*

[23]. Altera Corporation. *MAX+PLUS II Software.*

[24]. Microprocessor and Hardware Laboratory, Technical University of Crete. Greece

[25]. Technical University of Crete, Chania. Greece.

[26]. Xilinx. *Parallel Cable III.*

[27]. Altera Corporation. *ByteBlaster Serial Download Cable Data Sheet*, 1998.

[28]. GNU. *Flex Lexical Analyzer Generator.*

[29]. GNU. *Bison Syntactical Analyzer Generator.*

[30]. Sun Microsystems. *Java Virtual Machine*

[31]. Sun Microsystems. *Java Communications API.* 1998.

[32]. Brigham Young University's Configurable Computing Laboratory. *JHDL.*1998.

[33]. Sun Microsystems. *Forte For Java Community Edition IDE*. 2001.

[34]. Steven A. Guccione and Delon Levi. *JBits: A Java-Based Interface to FPGA Hardware.*

[35]. Eric Lechner and Steven A. Guccione, "The Java Environment for Reconfigurable Computing", in Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, FPL 1997. Lecture Notes in Computer Science 1304", Wayne Luk and Peter Y. K. Cheung, eds., Springer-Verlag, Berlin, September 1997, pp. 284-293.

[36]. Xilinx, Inc., "XC6200 Development System Datasheet", 1997.

[37]. Steven A. Guccione, "Programming Fine-Grained Reconfigurable Architectures", PhD Thesis, University of Texas at Austin, May 1995.

[38]. Lucent. *ORCA Series 2 Field-Programmable Gate Arrays Datasheet.*1999.

[39]. Lucent. *ORCA OR3LxxxB Series Field-Programmable Gate Arrays.* 1999.

[40]. Lucent. *ORCA Series 4 Field-Programmable Gate Arrays*. 2000.

[41]. John Schewel. *A Hardware / Software Co-Design System using Configurable Computing Technology*.

[42]. Paul Graham, Brent Nelson and Brad Hutchings. *Instrumenting Bitstreams for Debugging FPGA Circuits.* In Preliminary Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines, FCCM 2001.

[43]. Satnam Singh and Phil James-Roxby, *Lava and JBits: From HDL to Bitstream in Seconds.* In Preliminary Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines, FCCM 2001.

[44].

World Wide Web Sites and Resources

[1]. Xilinx: Programmable Logic Devices, FPGA & CPLD. http://www.xilinx.com/

[2]. Altera Corporation: The Programmable Logic Solutions Company. http://www.altera.com/

[3]. Lucent. http://www.lucent.com/

[4]. The Source for Java Technology. http://java.sun.com/

[5]. Forte For Java Developer Resources. http://forte.sun.com/

[6]. Java Communications API. http://java.sun.com/products/javacomm/index.html

[7]. Java 2 Platform Standard Edition. http://java.sun.com/j2se/1.3/

[8]. Microprocessor and Hardware Laboratory (MHL) and ReRun information. http://www.mhl.tuc.gr/