Train Set Bus Controller

by

Nana S. Kender

Diploma de inginer, Universitatea Tehnica Timisoara, Romania, 1994

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

December 1998

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of COMPUTER SCIENCE

The University of British Columbia
Vancouver, Canada

Date Dec. 18, 1998

DE-6 (2/88)

Abstract

This thesis presents the design of a bus interface for a computer controlled train set. This design is useful for experiments in real-time control and embedded systems. The design was based on an experimental methodology based on modeling both hardware and software with programs in a guarded command language. The successful completion of this design provides empirical support for this approach.

Contents

List of Figures

In memoria tatalui meu.
To the memory of my Dad.

Chapter 1

Introduction

Embedded systems can be found everywhere from airplanes to doorknobs. As embedded systems have become more widespread and more complicated, interest in systematic approaches to their design has grown as well. An embedded system differs from a general purpose computer by its specialization for a particular group of tasks; often the system has to deal with interfacing and communication between a wide variety of sensors (i.e. input devices) and actuators (i.e. output devices). Design specification plays an important role in the design process.

Embedded systems are inherently concurrent. There are the sensors to be monitored, the actuators for process control, and several software tasks may coordinate all these activities. If we take as an example the automatic control of trains on the same track system, several trains move on the track at the same time; position sensors need to be monitored, each train has its own set of geographic objectives and time schedule. This raises a coordination problem: avoiding collisions and fulfilling the schedule requirements. Our model train set – a research project in the Integrated System Design lab, see section 1.1 – includes several pieces of hardware, each with its own notion of time: the host computer, the ISA bus, the train bus, the switch controllers, the sensor interfaces, the signaling protocol on the track by which commands are sent to the speed controllers in each train. These all operate concurrently.

Concurrency may introduce non-determinism. Most hardware description languages (HDLs) generally have deterministic semantics to appeal to designer intuition and to simplify synthesis. Multiple notions of time, multiple clock-rates, and non-determinism are hard or impossible to represent in such commonly used frameworks. Although many HDLs prevent the designer from expressing non-determinism in their models, this does not ensure that reality will cooperate: events may occur in the implementation in orders that were not considered by the HDL model. Such

incomplete modeling can lead to errors in the final design.

This thesis uses a concurrent programming language, Synchronized Transitions (ST), for design specification. An introductory description of the language is given in chapter 4. It consists of a set of state variables and transitions; transitions can be combined either synchronously or asynchronously. Considering the train set example, we have components that operate synchronously but at different speeds so there may not be a synchronous way to describe their interaction; also the software that controls the whole application can issue commands at its own pace, which should not have to be predictable. Here, the concurrency and non-determinism inherent in the ST language provide a natural way to model the design.

ST was originally proposed by Ravn and Staunstrup [14] and has been used extensively for design and verification of both synchronous and asynchronous circuits. Much of this work has been done by students at the Danish Technical University in Lyngby. The ST compiler used allowed either synchronous or asynchronous descriptions, but not both; mixing the two kinds of operators was not allowed. The ST version used in this thesis removes this limitation.

The mixing of combinators in UBC ST provides a natural way to express concurrency and non-determinism encountered in co-design. The utility of this approach is demonstrated by using it to design a bus interface for a computer controlled train set. A more detailed description of the necessity and usefulness of mixing combinators is given in chapter 4; chapter 5 shows in what way this feature affects the translation from ST to VHDL.

1.1 Train set project

Chapter 3 presents the train set. The train set is a test bed for embedded system design in the Integrated System Design lab in the Computer Science department at the University of British Columbia. It consists of a model train set (with tracks, engines and cars) connected to a computer via a parallel bus. The test bed is used for studying and illustrating safe implementations (e.g. running the trains without accidents) in an integrated software/hardware system.

The project was started in 1992 by two former graduate students, and although it has been operational, several factors have determined the decision for important design changes. These are explained in detail in chapter 3. The main concern are electrical problems (transmission line effects) and design modularity; also the possibility to use the project for design verification studies.

1.2 Summary of thesis contributions

The research undertaken for this thesis included the design, implementation, and test of a useful interface for real-time control and embedded system research. The design is more modular than the previous design and it allows a standard computing platform to be used for the controller. The thesis offers a documented design for future verification projects.

Furthermore, this thesis proposes a methodology for hardware/software co-design. The train bus controller represents a component of interest from a co-design point of view: an interface between two buses with two different clocking methodologies. This design process was an experiment to validate the design methodology. The ST language provided the framework for co-simulation. The ST design model was used – with manual translations – for board layout and component programming. Currently, we do not have automatic synthesis and layout tools based on ST. Instead, the detailed ST description was translated to VHDL. Although this is not an efficient design approach, it allowed a comparison of ST with VHDL.

Chapter 6 gives an evaluation of the chosen methodology and test results. It summarizes the accomplishment of this thesis and suggests further work.

Chapter 2

Co-Design

2.1 What Is Co-Design ?

Co-design refers to a common framework for designing the hardware and software architectures for an embedded system. "Embedded system" is a definition for a very wide range of digital systems used mostly in dedicated applications, containing analog circuitry and often mechanical parts as well. Co-design has become a strategic technology for systems as simple as household items like thermostats or answering machines, as well as for complicated medical instrumentation, automobile control systems, or "fly by wire" aircrafts.

While the design of general purpose computers aims at a solution which minimizes cost while maximizing speed, storage capacity etc. for a broad range of applications, embedded systems are very application specialized. The application domain also dictates the co-design methodology for a particular embedded system. Apart from cost constraints, these systems also often have power, weight, and physical size limits (Wolf [21]); embedded systems usually have to meet hard real-time deadlines.

For general purpose computers, requirements are relatively well defined: the instruction set determines the correct behaviour of the processor, and performance targets for benchmarks give high-level timing properties. For embedded systems, specifications must describe the interaction of the system with its environment, and producing a correct and complete specification can be much more challenging. For example, consider the design of an ignition and fuel-injection controller for an engine. The designer's intention may be in terms of performance and efficiency of the engine with its controller. Thus, a specification for the controller must include a detailed model of the engine. Such a model may not be available at early stages in the design.

Adding to the difficulty of the co-design problem is also the fact that in embedded systems, as the name says, the computing part is embedded, therefore hard to access for debugging. Traditional break-point debugging is often infeasible for embedded designs. Again, consider the example of the engine controller: the embedded system operates in a real-time environment. The only way to test the controller is when it is connected to an engine. If the software stops at a breakpoint, the engine will stop running.

Given the specification of a system which may include hardware, software, analog and mechanical components, co-design represents the process of going from the specification level to the implementation level. A typical co-design flow is hard to give since co-design tools differ in how they delimit the design steps, but a generic co-design methodology would include some key phases such as (see Gajski et al. [5]):

- Specification: stating system requirements and building a model of the design from these. If we take the engine control example, design specification includes [9]: (a) identifying the tasks of the system – in this case providing the correct amount of fuel and firing spark plugs at the right time, (b) identifying the inputs to the controller, such as signals from the crank position sensor or the manifold pressure sensor, (c) usually making abstraction of details, deciding how the system is going to fulfill its tasks. This includes dealing with deadlines, with obstacles to correct functioning – such as noise –, with power constraints, safety requirements, etc. In other words, design specification gives a description of the design as the outside world expects it to behave.

- Allocation & Partitioning: In general, a specification should describe *what* the system is supposed to to, and an implementation describes *how* it does it. However, most work on specification for co-design has been done by the CAD community with a goal of automatic design synthesis from the specification. To make this goal practical, specifications for co-design tend to include top-level design decisions. For example, a specification may be in terms of a collection of communicating processes [15] or as a "control data flow graph" [11]. Each process can be implemented in either hardware or software, and the specification describes the data and control interactions between these processes.

  Allocation is the process of choosing the type and number of components for implementation. partitioning defines the mapping of functions and/or processes from the specification onto the allocated components. Most often allocation and partitioning constitute one step only since if the partitioning is not satisfactory, a different allocation may be tried. Much work has been done towards CAD tools for this step of co-design [7, 12, 11]. There is usually

no single optimal solution to this allocation & partitioning problem; there are trade-offs that need to be balanced. For example, one might want to put all the functions into a general system processor to minimize cost. However, such a processor may not be fast enough to handle everything. For the engine controller, there may be one processor that computes the injector setting and spark timing. A second processor may handle the real-time interface with the engine. A dedicated DSP may process some of the data that is sampled to assess the engine's operation. An ASIC may provide communication between these processors.

- Scheduling: The operation of the embedded system is partitioned to tasks, and tasks have been assigned to various kinds of CPUs, ASICs, etc. The same hardware component may handle several tasks. This is why scheduling is needed. Taking the engine controller example, there are operations that need to be done for every rotation of the crankshaft: actuating fuel injectors and firing spark plugs. These have real-time constraints. There are operations that can be done on a longer time scale, but still must be fast compared to human response time: computing the amount of fuel to be injected, setting the throttle valve in the intake manifold. There are computations that can be done on an even longer time scale: changing mode of operation based on engine temperature, air temperature, oxygen content in exhaust, etc. These processes need to be scheduled so that all deadlines are met.

  There are many approaches to solve this NP-hard problem. One that takes into account non-determinism (the impossibility to predict task execution times) is the work of Gerber et al. [6]: an offline component checks if there is a possible scheduling to meet all constraints and then produces a calendar that has lower and upper bound functions for the start times of the tasks rather than absolute numeric conditions. The online component then fills in the numeric values of the parameters inside the functions as they become known (the parameters in the bound functions of one task are start and execution times of other tasks). Other approaches, as reviewed by R.Camposano et al. in [4], are

  - ASAP/ALAP (as soon as possible/as late as possible) scheduling;
  - List scheduling – the difference between the ASAP and ALAP times are computed for all the tasks; this denotes the *mobility* of a task. The task with the least mobility has the highest priority. Other criteria for priority can also be used.
  - Force-directed – also starts from computing the ASAP/ALAP difference, then the algorithm builds a model of "operation density" using the prob-

abilities of operations to fall into certain execution steps. From here there may be different variations of summing up these probabilities and deciding on the actual scheduling.

- Path-based – each of the possible execution paths is scheduled, then the path schedules are merged to form a single state transition graph.
- heuristic approaches: "percolation" scheduling, and scheduling by simulated annealing or simulated evolution. These are based on an initial (trivial or random) schedule from which better solutions are iteratively generated using heuristics.

- Communication Synthesis: components need to communicate with each other. This communication can be implemented using shared memory, buses, special serial links, etc. In the case of our train set, communication happens through a special network - a bus with an established protocol. For the engine controller it is signal lines, that is serial ports, that connect the controller to sensors and actuators.

Analysis and validation is necessary after every step. In our case, the bus controller fits into the context of a bigger co-design research project, the train set described in the next chapter. The enumerated design steps apply mostly only to the train set as a whole. For the controller, the choices for allocation and partitioning were quite straightforward, as described in chapter 5.

2.2 Current Issues in Co-Design

A good reference for the reader regarding current issues in co-design is an IEEE roundtable [16] where seven designers were invited to express their views on the definitions, characteristics and shortcomings of co-design. This roundtable actually shows how vaguely defined the term co-design still is. For instance, Paulin sees co-design as all-hardware, programmable or not, while Harr says designers start from the premise that everything should be done in software and only the necessary hardware should be included.

Nagasamy points out that the "co" in co-design should stand for "concurrent and cooperative design", and that the real challenge lies in designing the software *along* with the hardware. Agnew embraces this viewpoint too. Having different teams working independently on hardware and on software is not really co-design. Ernst prefers calling it computer-aided co-design because it is the CAD support that unifies hardware and software development, but he further states that no decent tools exist that go all the way down from specification to Register Transfer Level

(RTL) code, which is due the variety of target architectures. Wolf [20] also points out pressing needs for CAD tools in several design phases such as co-simulation, restructuring and partitioning of processes, system- and program-level evaluation.

In Paulin's opinion [16], the main issue in co-design is compilers for the diversity of processor architectures and the real-time constraints typical to embedded systems. But Nagasamy sees a single pressing issue, which is the capture of the design engineer's specification. Paulin and Harr agree this time that the design process in most cases starts from an incomplete specification; such specifications are assumed complete and accurate before design starts, in other words, this high-level description is usually not "debugged". An impressive percentage of the roundtable article is taken by arguments for the need for good specifications. Yasuura comes right out and says academics should focus on a new specification model.

2.3 Models and Specification Languages for Co-design

As Staunstrup states in [19], the aim of high-level design techniques is to reduce design time and effort by moving decisions upwards in the abstraction level of the design models. As designs become larger, the details are too many to be all grasped by one designer. Models are necessary for design overview. As electronic designs become more complex they push the abstraction levels of models upwards, making it common today to describe hardware circuitry in a similar way to the abstraction found in software.

In other words, designing hardware and software is not that much different anymore. The delimitations of hardware and software components within an application become less evident, and the designer needs to be able to model his design in a manner general enough to cover both hardware and software, without any commitment from the start as to what should be implemented in hardware and what in software. But most existing modeling tools and languages today start from either pure hardware description (for example Gupta and DeMicheli's approach [7]), pure software description (for example COSYMA [12]), or mixtures of both but clearly delimited by using a different language for each (like CoWare [15] and Ptolemy [8]).

Vulcan [7] is a hardware-software cosynthesis system which performs automated partitioning on an internal graph representation; initially, the design is specified in HardwareC, a subset of C. Vulcan starts from assuming an all-hardware design and then performs repeated iterations of trying to move parts of the design to software; the criteria are: (a) whether time constraints are still satisfied and (b) minimizing communication overheads. Vulcan can handle parallel processes: hardware and software components may run in parallel [11]. The method targets systems

consisting of ASICs and a CPU to reduce ASIC size.

COSYMA [12] is quite similar to Vulcan, except it starts from an all-software assumption of a design specified in C*. C* is a superset of C with added features to allow for timing constraints, task concepts and task communication. COSYMA targets systems consisting of one CPU and one ASIC for processor speed-up. As mentioned, it is a "software-oriented" approach; hardware is added only where necessary because timing constraints are violated by the all-software solution. It is an automated software-oriented partitioning tool with hardware extraction when needed.

LYCOS [11] is another automated partitioning tool. It also starts from translating specifications to an internal graph representation, but it aims at not to limit the designers in using the specification language they prefer; it currently supports C and VHDL but the research group is working towards including other specification languages among the accepted ones – including Synchronized Transitions. The way it works is that based on the internal representation graph, LYCOS generates "Basic Building Blocks" which later can be moved between hardware and software. The partitioning algorithm – PACE – is based on elaborate methods for estimating software execution time, hardware execution time and hardware area size.

CoWare [15] is a hardware-software co-design environment which aims at integrating hardware and software components that were specified, implemented, simulated in different languages and with different tools. A similar approach is found in Ptolemy [8]; partitioning and mapping is done at the very beginning of the design process. CoWare uses notions such as processes, ports, channels, protocols, communication mechanisms – Ptolemy has blocks with portholes – the idea is that after hardware and software components are produced by different tools, they need to be interfaced correctly to result in the final system. These approaches do not solve the problem of the separate design of hardware and software for an embedded system, from the root of the problem, but aim at helping the designer "glue" the components together in a correct and reliable way at the end.

The computational model of a design is a "delicate balance between abstract and concrete" (Staunstrup [19]): if it is too concrete, the designer is constrained by low-level decisions in early phases; if it is too abstract, it may become difficult at later stages to make an efficient realization. However, it should be abstract enough to describe computations in a range of technologies.

2.4 System Modeling and Co-simulation

As mentioned above in section 2.1, the process of going through various levels of abstraction of the design model requires simulation after the model for each level has been established, to make sure the model still respects the initial system requirements. If design is actually co-design of hardware and software, then simulation has to become co-simulation of hardware and software, since we want to simulate the system as a whole and not just isolated components.

From the examples given in the previous section, we can conclude that system-level specifications can be viewed as homogeneous – when a single language is used, like in [7], [12] - or heterogeneous, where different languages are used for hardware parts and software parts ([11], [8], [15]). This thesis uses and emphasizes the benefits for the former, using ST as the single language for modeling the design. A most common example for heterogeneous specifications on the other hand is the mixed C-VHDL model. Heterogeneous specification approaches make co-simulation more complex and difficult since they have to deal with interfacing, translating protocols, etc.

2.5 Summary

Co-design is a relatively new field of computer science, and one in which designer's efforts to cope with the difficulties of the moment have had to be "quick fixes" for the simple reason that industry and market did not have time to wait for "elegant" solutions. Although much work has been done by academia to address the most pressing needs in industry, some issues, as mentioned in section 2.2, are still not at all addressed, or in our opinion not satisfactorily solved. Some of these issues, which the present thesis addresses, are:

- high-level specification with possibilities for automatic verification;
- modeling concurrency and non-determinism inherent to embedded systems;
- providing a single specification language for both hardware and software - modeling the design as a whole rather than splitting it into hardware and software from the start.

Chapter 3

Bus Interface Specification

3.1 The train set

As mentioned in chapter 1, the train set is a model train set built and used in our lab for real-time application experiments.

3.1.1 Acknowledgments and brief history of the project

Since the train bus project was started several years ago and many students (and 2 faculty members) contributed to its design, I will give a brief history of the work:

- 1992: First train-set built. The idea to do a train set was Dr.Carl Seger's based on a similar set-up at the University of Waterloo. The hardware was designed and built by Andy Martin and Eric Borm. Mike Donat and Nancy Day wrote demonstration software (to move trains randomly without collisions).
- 1993: Train-set considered in class project in CpSc 513. A bus-based design was chosen. The existing buses were evaluated and we chose our own design for reasons of scalability and potential for verification (see section 3.4.4). The original train-bus protocol was worked out in a project involving Catherine Leung and Dwight Makaroff.
- 1994: David Weih wrote an ST model for the ISA to train-bus interface.
- 1995: Mohammad Darwish designed hardware for the bus interface based on David's code.
- 1996: Dr.Mark Greenstreet recognized the ground shift problem. His solution to this (see section 3.4.3) involved changing the protocol from two-phase to four-phase; also adding series resistors to drivers for signals other than the

clocks; using complementary signaling for clocks to improve robustness. It appears that complementary signaling is not needed for other signals; doing so would result in an unwieldy number of wires in the bus.

- 1996: I took over the project.

3.1.2 Description of train set components

Figure 3.1 is a diagram of the train track system; it shows tracks, switches (marked with an "S") and position sensors (small circles). The train set has been working with three trains in the past; they are shown in the starting position.

There are about 18 meters of track, 13 track switches and 60 position sensors. The sensors are photo-darlington transistors that detect the shadow of a train as it passes over the sensor. The train engines include decoder chips like the ones used in infrared (IR) controllers for TVs, VCRs etc. Commands are sent to the trains by pulsing the power supply that drives the track. Each such command has 9 bits: 5 bits specify a train, and 4 specify the speed. Each train has 8 forward and 8 reverse speeds. There are 32 train addresses, but not enough room on the track for that many trains. Typically, the train set has been operated with three trains.

The figure-8 type of track layout requires a polarity reverser. Trains may start in the same direction and end up facing each other. This means that if initially both had all the right wheels touching the positive rail and all the left wheels touching the negative rail, eventually one of them will have all the right wheels negative; that train went across a polarity reverser. Polarity reversal means there must be electrical breaks in the rails. Furthermore, the engines draw current from all of their wheels, which means that all of the wheels on one side of the engine are connected in parallel. The polarity reverser is a segment of track that is electrically isolated from the segments before and after it. As a train approaches this segment, the segment is connected to the power supply in the same polarity as the segment that the train is currently on. This allows the train to safely cross onto the reverser. When the train is fully on the reverser, the polarity of the reverser is flipped. This allows the train to safely continue on to the next segment. The trains have bridge rectifiers in them so they can accept power of either polarity without changing direction. The direction control is independent of the polarity of the tracks. The power is DC (a reverser would still needed if AC power were used).

The difference between the existing and the proposed train set is in the implementation of the controller. The existing train-set was implemented by two former graduate students in 1992, as described in section 3.2. The necessity for a more modular design which could better serve design and verification experiments

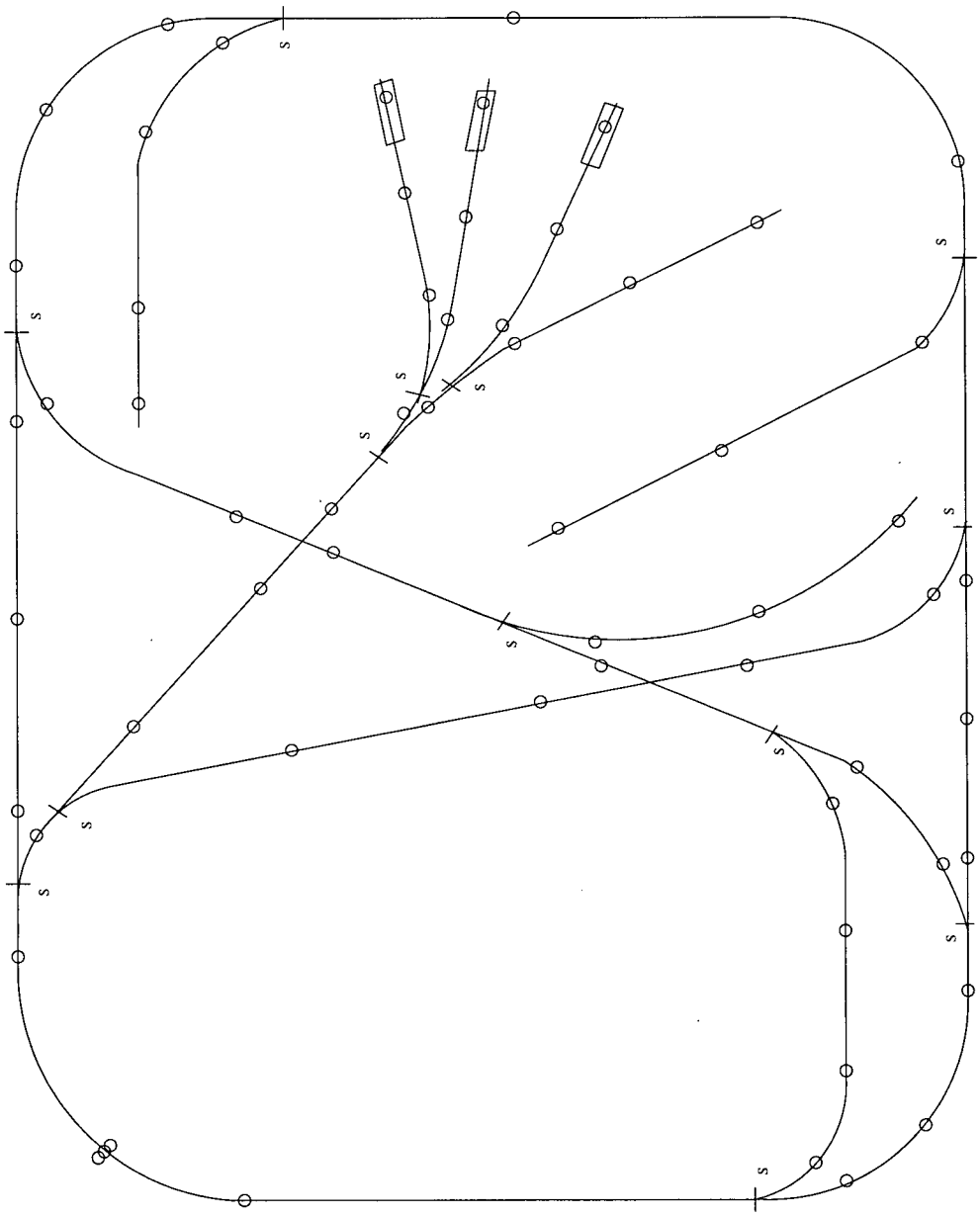

Figure 3.1: The track topology

became evident later. A key piece of the new design is a bus interface, which is the focus of this thesis.

3.2 The existing design

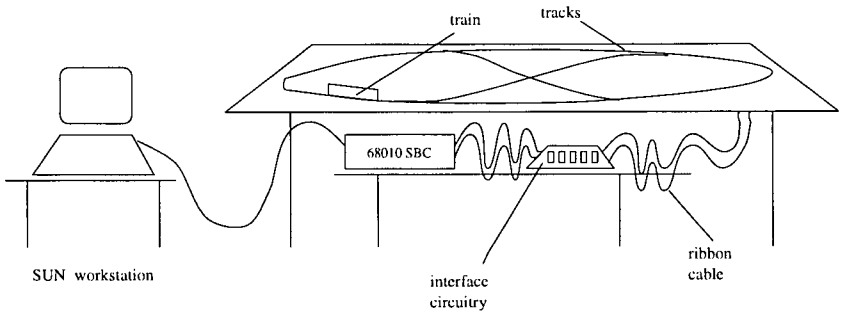

Figure 3.2: The old architecture

Figure 3.2 shows the existing train controller architecture. Programs are written and cross-compiled on the SUN workstation. Executables are then downloaded over a serial link to the single board computer (SBC). A MOTOROLA 68010 based controller was used because UNIX doesn't provide real-time guarantees; the simpler computer can respond to real time events. The 68010 SBC communicates with the interface hardware using its parallel port. This involves using undocumented features of the parallel port to allow data to be read from the port.

The interface hardware consists of 31 TTL chips in addition to some discrete components for the speed controller and to read the photo transistors. There is a rats nest of wires under the track layout table, to connect to the switches and sensors in the track. The design is monolithic and incompletely documented, which makes any hardware modification difficult and unreliable.

3.3 The proposed design

To support more experimentation, we wanted a more modular design. An adequately documented design is also a prerequisite for verification. The key change is to make the control and sensor hardware distributed. This will allow individual pieces to be replaced for design experiments.

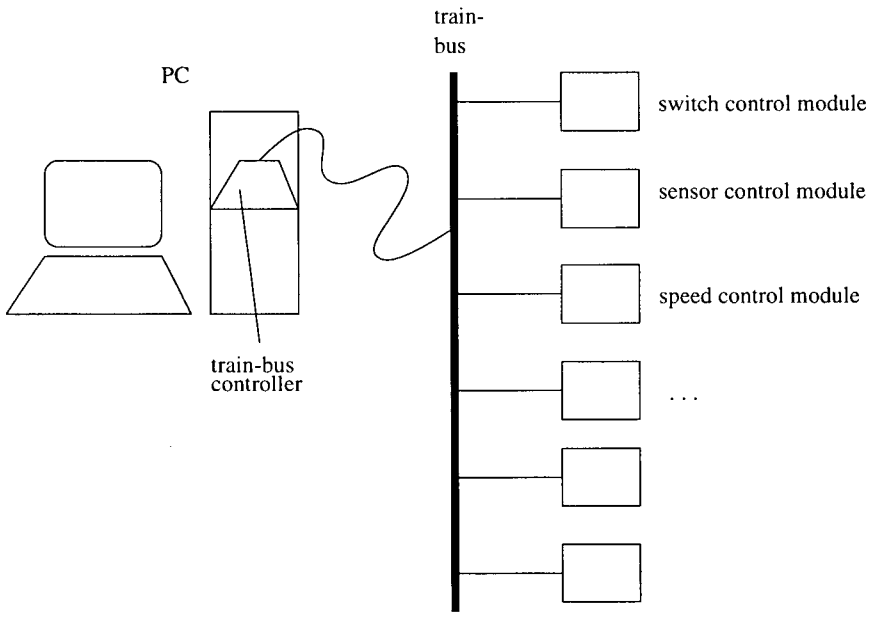

Figure 3.3: The proposed architecture

Figure 3.3 presents the proposed architecture for the train set control. The software will run on a PC as the host computer; the train-bus controller will be a PC-board. We chose a PC-based controller because PCs are readily available, cheap, and provide a popular software development environment. This also means we can further expand the design in the future – we can add other interfaces, such as network cards to communicate with other computers during experiments. The use of DOS makes it possible for an application to override all operating system functionality, thus making real-time applications possible.

The separate modules for each control and sensor function allow the design to be modified one piece at a time; they provide an alternative to the monolithic rats nest of wires of the current design.

3.4 The train-bus

The decentralization of the design requires an organized means of communication between the modules. This is the train-bus, which provides a simple interface to sensor and actuator modules. The simple design should help teaching and verifi-

cation research.

3.4.1 High-level design choices

A parallel bus was chosen for teaching and verification purposes. Also, a protocol for the bus had to be established: it should be synchronous, master-slave. This should make it easier to teach to CS grad students with little hardware experience. Bus operations are simple. There is a clear mapping of wires to functions (unlike a serial bus where the same wire carries several different logical signals at different times). We hope that this will make formal verification easier as well.

From a mechanical point of view, the decision was for ribbon cable. Devices can be connected via crimp connectors. The bus goes under most of the track so that sensor and switch modules can be close to the devices that they sense or control. We expect to use between 5 and 15 meters of ribbon cable. This makes it impractical to ensure tight control of electrical properties.

3.4.2 Logical details

The wires of the train-bus are split into logical groups as follows:

- 8 wires for data
- 8 wires for address
- 4 wires for command

The justification for choosing these numbers is as follows: for address, 4 bits presented a risk for running out of device addresses if the design was successful. 8 bits offer 256 distinct addresses, which seems large enough to be a safe choice (since slave devices may be controllers that can handle several sensors/actuators). For data, an 8 bit bus is small enough to keep the bus from having too many wires and seems adequate for typical control applications. The 4 bits for command offer 16 possible commands, which means again flexibility for future extensions.

3.4.3 Electrical details

This section presents some issues that need to be considered when implementing a long bus with many devices. The devices draw no DC power. The bus operates at a relative low clock frequency, giving the signals time to settle to valid digital values. However the low frequency does not mean we can completely ignore high-frequency effects. Ringing, spurious triggering, etc. are all possibilities because the logic devices can drive their outputs with small rise and fall-times, the ribbon cable

can transmit relatively high-frequency signals, and the logic devices can respond on short time-scales. We have to make sure that this sensitivity to short-time scale behaviour doesn't cause the system to malfunction.

Our concern is about reflections and about power voltage shifts that may occur for imbalanced changes of the signals. According to Kirchhoff's current law, the current through the ground circuit must equal the current through the signal wires. The signal wires each have their own impedance so it will take a certain time for the data/address lines to become low, during which the ground voltage level is shifted upwards. As a worst case example, let's see what happens if 8 address and 4 command wires happen to change in the same direction at the same time (later in this section, figure 3.5 and the related explanation show why we are considering this example). We get the voltage divider effect shown in figure 3.4, where $V1/V2 = 2/3$ and $V1 + V2 = 5$, which means that the signal lines will go to 2V (instead of 5V) and the 8 ground lines will go to -3V.

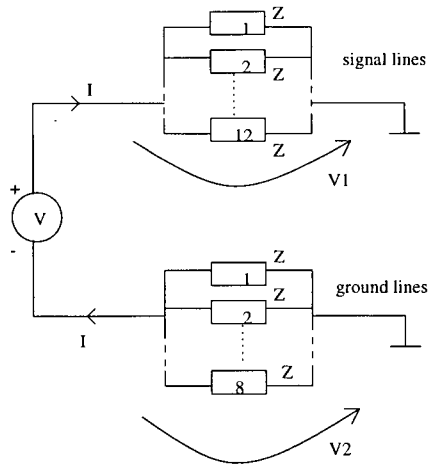

Figure 3.4: Voltage shifts at imbalanced changes of signal lines

The signal lines of the train bus are very long and have to be analyzed as transmission lines. Reflection along the cable lines causes the power lines to bounce back and forth after such shifts, and since ground is the reference voltage, the circuit behaves as if the signal lines would bounce. Even if power shifts would not be a problem, reflection means that every time we have a transitions at the driving end of a line, this voltage change will travel along the cable and get reflected at the other end, which results in ringing on the signal lines.

We apply a strategy to deal with this problem:

- We use series resistors to limit drive current for address, data, and command. The impedance of the ribbon cable was measured (about 70 ohms). Using 680 ohm series resistors gives a total drive impedance of $680/12 = 56$ ohms (for the worst case considered, of 12 wires switching simultaneously). Ground impedance is $70/8 = 9$ ohms. Thus, we expect a worst-case ground shift of about 1 volt.

- We use differential clocking, so clock signals should have minimal contribution to ground shift. Also, with the 4-phase protocol, address and command lines are changing at different times from data lines (less wires that may change level at the same time). A timing diagram of the new protocol is shown in figure 3.5. The actions corresponding to the 4 marked clock edges are as

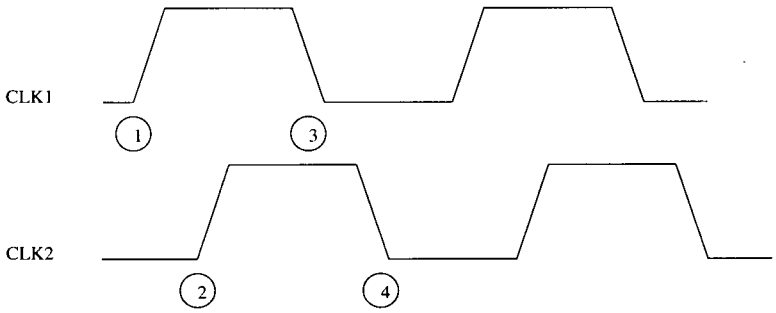

Figure 3.5: A 4-phased Train-Bus Protocol

follows:

  - 1. controller sends address and command;
  - 2. device reads address and command;
  - 3. controller or device sends data (depending if it was a *read* or a *write*);
  - 4. data received.

- We use Schmitt-triggers and C-element clock debouncers, as in figure 3.6. The Schmitt triggers annihilate slight variations on the signal lines, while the Mueller-C element eliminates ringing. That is, one signal can oscillate until the other one changes, without causing problems.

As a summary, we make the clock distribution robust and we only look at the address, data, and command signals at times when they are guaranteed to have settled.

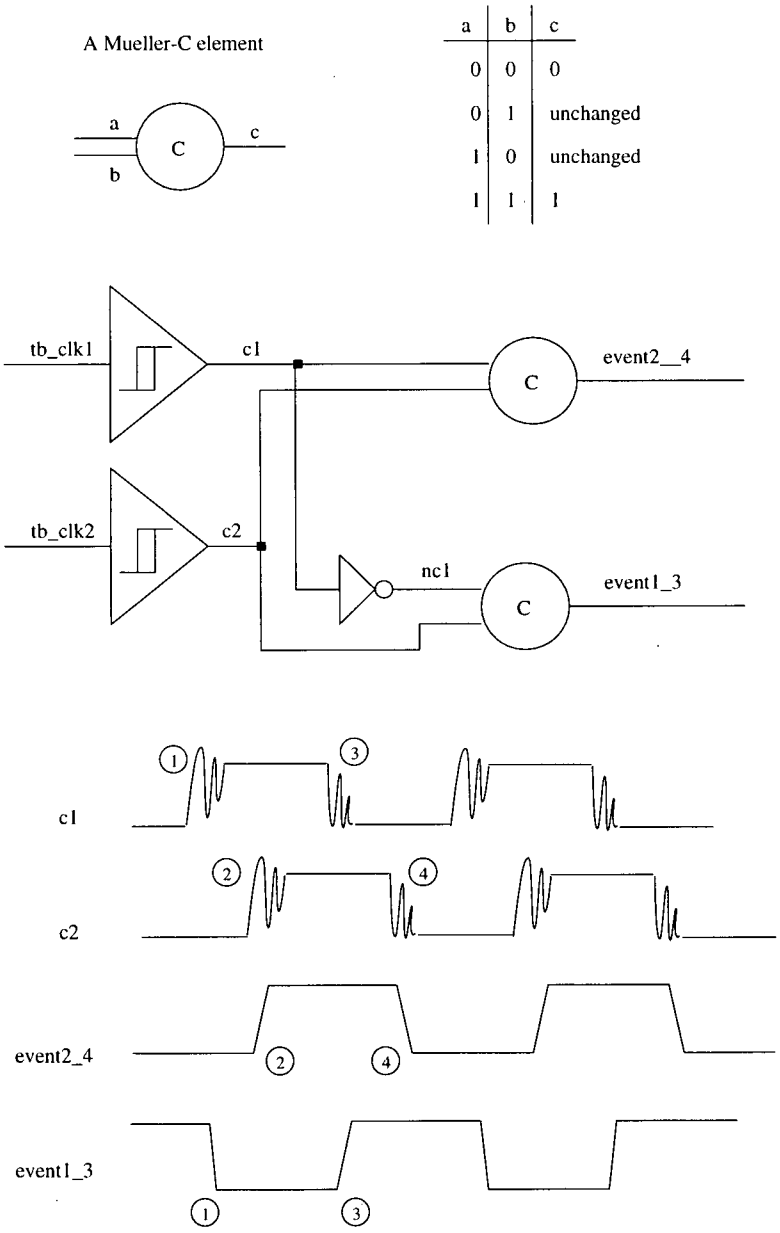

Figure 3.6: The train-bus clocks

3.4.4 Comparison with other bus-schemes

- HP-IB: (Hewlett Packard Instrumentation Bus interface, or GP-IB or IEEE 488 standard) This bus standard has similar bandwidth (also 8 bit parallel data). It allows a limited number of devices to connect (15). Devices present a DC load, which limits scalability.

- IIC: This is a serial, twisted pair interface standard for embedded controllers. It would require less wiring and we could use micro-controller chips that have the interface hardware built-in. In that case however, all the design details would be buried in those microcontrollers. We believe that it would be difficult to write a satisfactory formal model for such a controller. This would be a barrier to our verification efforts. Also, hiding all of the hardware would make the design be of less pedagogical value

3.5 The ISA to train-bus interface

As mentioned in section 3.3, the proposed design uses a controller between the host PC and the train bus; the controller is the interface between the ISA-bus and the train-bus.

The design of this interface is the subject of this thesis: the interface is needed before other modules can be built and tested. Also, interfacing between two different bus protocols with two different sets of timing requirements provides an interesting design example.

The ISA bus

We used the PC's Industry Standard Architecture (ISA) bus because it's simple and we do not need the higher performance of other PC buses. Following is a very brief description of the ISA-bus, presenting only details relevant for our application. More information can be found in [17].

A standard ISA 8-bit I/O cycle is given in figure 3.7.

The meanings of the signal names are:

- BCLK - bus clock: the ISA clock;

- AEN - address enable. This line is driven by the platform circuitry as an indication to ISA resources not to respond to the address and I/O command lines when the DMA controller is the bus owner.

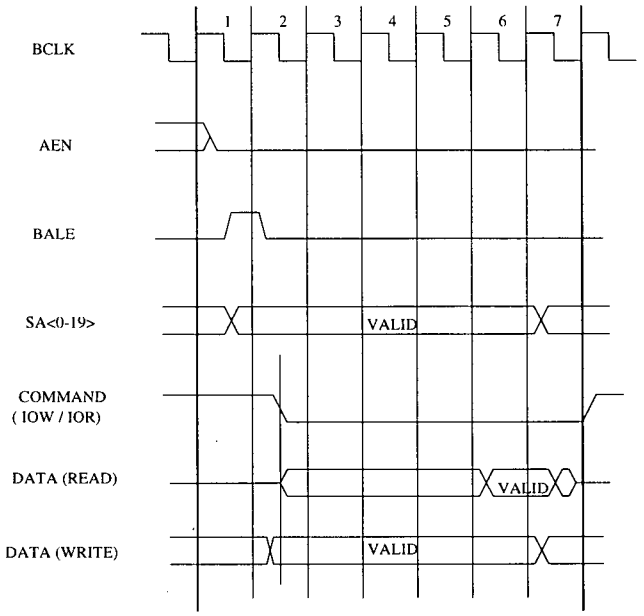

Figure 3.7: Standard ISA cycle

- BALE - bus address latch enable. This signal is driven by the platform CPU to indicate when the address lines are valid. This signal is used to latch the address lines.
- SA(0-9) - the address lines.
- IOW - I/O write: indicates a write cycle to an input/output port;
- IOR - I/O read: similar as above;

The cycle starts with a rising edge of BALE. On the falling edge of BALE, the address is guaranteed to be valid (on lines SA0-9). On the rising edge of $BCLK_3$ we are guaranteed to have the correct value for the command lines. There are two separate lines for read and for write: IOR and IOW, both active low. If neither is driven low, this means the cycle is not an IO-access. If it is an IO-write cycle, the data to be written is already valid at the falling edge of IOW. If it is an IO-read cycle, the data must be valid on the fifth rising edge of BCLK after IOW goes low ($BCLK_7$ in the figure).

The ISA-bus supports many other operations. For example, there are "early read" and "early write" operations that can be performed if the device asserts the

appropriate signals. Alternatively, the device can signal that it is not ready and the bus stalls until the device indicates that it is ready. We use the default timings for simplicity. There are also 16-bit transfer cycles for both IO and memory operations, but we do not need to use these.

Read and write transactions on the train-bus

A timing diagram for the 4-phase clocking was shown in figure 3.5. The train-bus protocol operates on a master-slave basis; the controller is always the master of the train bus, while the devices connected to it are the slaves. Regardless of the direction of the transfer, it is always the master who initiates it. The least significant bit of the command specifies whether it is a command of type write or read, so there are 8 write commands and 8 read commands possible. The signal diagram for a write command on the train-bus is shown in figure 3.8 (a write to a device connected to the train-bus). On the rising edge of $tbclk_1$ (clock-event #1), the master (the

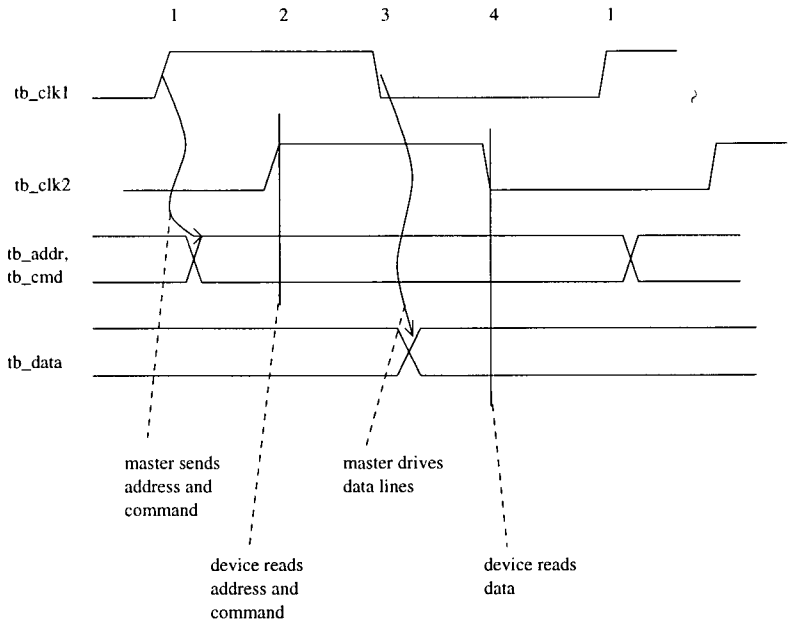

Figure 3.8: A write to a device on the train-bus

bus controller) sends the address of the device the command is intended for, and the actual command. On the following clock event (#2), the rising edge of $tbclk_2$, the devices decode the address and command lines. On the falling edge of $tbclk_1$ (clock-event #3) the master drives the data lines with the information intended for

the addressed device, and on the falling edge of $tbclk_2$ (clock-event #4) the device that decoded its own address on event #2 will read the data.

A read is quite similar except the direction of the data transfer is reversed so on clock-event #3 the slave device (instead of the master) is the one who drives the data lines with the requested data, and it is the master who reads the data lines on clock-event #4.

The top-level controller design

We can model the design as a Finite State Machine (FSM) for the ISA bus on one end, another FSM for the train-bus at the other end, and FIFOs in-between for communication.

The application running on the PC, i.e., performing write and read transactions on the ISA-bus, will see the controller as a collection of registers. We need one for Data, Address and Command each, and then we also need one for Status because we need to know when the controller is ready to accept a new command; and since the train-bus operates at a much lower frequency than the ISA-bus, this means we must have a way to inform the application when there is requested data available to read. So our Status register has two bits of information, Cf0 (Command FIFO stage 0 is full) and Rf1 (Result FIFO stage 1 is full). The application must also be able to reset the controller; this is done by performing a write to the Status register. Every program running on the host computer for the train-set control will need to start by doing a write to this register. Note that the Status register may be both read and written, as well as the Data register; but the Address and Command registers may only be written.

Before sending a command, the program has to check whether the controller is ready for a new command, so it needs to read the Status register until the Cf0 bit indicates the FIFO has room in its bottom stage for a new command. The controller guarantees that only the application can set this bit, so once it was read as empty, it will stay so until the application issues a new command, that is, performs a write to the Command register. The application may write to the Address and Data registers without changing the status of the controller; only a write to the Command register changes that. A write to the Command register will cause the FIFOs to advance even if there were not new values supplied to the Data and Address registers. This means that if a new write command was issued for instance, the last values written into those registers will be used for the new command.

As mentioned in the section above, the controller does not need to know about the exact command, only about the transfer direction (read/write). A write command does not need to be retired, but a read command does, because when

issuing the command the application only asks for the data, and then it has to wait until the device provides the requested data. To retire a command, the program needs to read the Status register until the Rf1 bit indicates that the result FIFO has new data available; then it may perform a read from the Data register to get the actual data value. As with Cf0, the controller guarantees that once Rf1 became high, it will stay so until the program performs a read from the Data register, which is the only way to reset this bit. The controller also guarantees that the data read is always in the order the read commands were dispatched. Due to the FIFOs, we may have several train-bus transactions outstanding at any time; however, the condition for correct operation of the controller is that operations are completed in the same order that they were issued by the program.

3.6 Summary

We presented the train-set – the history of the project, the description of its components, and the train-bus. For the train-bus we justified the choice for the type of the bus (and the number of wires) as well as for the protocol. We explained the electrical considerations that were the reason for changing the protocol from two-phase to four-phase.

We further justified the choice for a PC-platform for the bus-controller. The circuit is an interface between the ISA bus on the PC and the train-bus on the experimental train set up; we described the protocols used in each of these buses, and the interaction between the two protocols.

Chapter 4

Specification Language

4.1 Why ST ?

An important issue in co-design is correct design specification capture (Nagasamy, [16]; Wolf, [20]). Because of the complex application nature of embedded systems, a clean and complete specification is both very important and hard to achieve. The penalty of specification errors grows with the time until detection.

Embedded systems can contain hardware, software, analog circuits, and mechanical parts. Interfacing them correctly is as important as it is challenging. The components are working at the same time, often at different rates. This is why writing specifications in sequential programming languages may not be accurate.

Staunstrup [19] makes a strong case for the use of concurrent programming languages in the design of embedded systems: for such systems, the order of external events and computation systems cannot be known in advance or prescribed, so sequential languages are not adequate since the operation sequence is unknown. Concurrent models are more appropriate since they can model non-determinacy, simultaneity, and multiprocessing, which arise in embedded systems.

Synchronized Transitions (ST) [18] is a concurrent programming language that will be described in section 4.2. It is easy to learn and use, yet very powerful for modeling. Its advantages address many of the current co-design problems (some cited above) and bottlenecks.

A major co-design bottleneck is in the design flow from specification to implementation (Nagasamy, [16]). In that respect, ST code is easy to translate to either another programming language like C or a hardware description language like VHDL. Key areas of co-design like partitioning and co-simulation are also helped by using ST. The same notation is used to model both hardware and software. Although we cannot claim we have practical synthesis techniques for going from an

arbitrary ST program to a hardware implementation or to an efficient software implementation, we do have techniques that work if the program is written in a certain style. However, the styles that are suitable for hardware implementation are not the same as the styles that are suitable for software implementation.

We also take a refinement based approach. The design starts with a high-level program that models the key behaviours of the intended system. This program is successively refined, adding more detail at each step, until we get something we can implement. Although there may be little or no distinction between hardware and software at the most abstract level, we tend to head towards hardware-specific or software specific styles for different parts of the program in this refinement process. Thus, partitioning is done by the designer as part of the refinement process.

Co-simulation is done naturally since everything, hardware and software, is modeled in the same ST program.

Last but not least, verification, or rather co-verification, is well supported by ST. It can even be done automatically [10, 13]

4.2 Synchronized Transitions

Synchronized Transitions (ST) is a concurrent programming language. In ST a design is modeled as a set of independent **transitions**. Transitions consist of a **guard** and a **multi-assignment**. For example

$\ll a \; AND \; b \rightarrow c := d \gg$

is read "a and b enables c gets d". "a AND b" is the guard of this transition; in other words, the assignment $c := d$ can only be executed if the guard evaluates to true. A transition is said to be **enabled** if the guard is satisfied.

Transitions are executed **atomically**, i.e. the evaluation of the guard and performing the multi-assignment is a single indivisible operation.

4.2.1 Combinators

ST offers three transition combinators, $\|$, * and $+$. If two or more transitions are combined with the **asynchronous** combinator, $\|$, then at each step in program execution, one is selected **non-deterministically** from those that are enabled. For example, if we have

$$\begin{array}{ll} & \ll \quad a \rightarrow c := d \quad \gg \\ \| & \ll \quad b \rightarrow e := f \quad \gg \end{array}$$

then if only one of a or b holds, the corresponding assignment will be executed; however, if both hold, one of the transitions will be chosen, but the choice is not specified by the program. This allows abstract models to describe a wide class of behaviours. It also provides an opportunity for optimization when deriving an implementation. For example, the implementation may choose which transition to execute to maximize performance or minimize the amount of hardware required, or to optimize some trade-offs. The final implementation is often completely deterministic.

The operands of the **product** combinator, *, are performed as a single, atomic state transition.

$$\begin{aligned} &\ll \quad a \rightarrow c := d \quad \gg \\ *&\ll \quad b \rightarrow e := f \quad \gg \end{aligned}$$

is equivalent to

$\ll a \text{ AND } b \rightarrow c, e := d, f \gg$

in other words the combined transitions are enabled only if both of their guards are enabled.

For transitions combined with the **synchronous** combinator, +, at each step during program execution all enabled transitions are executed as a single atomic operation. For example, if we have:

$$\begin{aligned} &\ll \quad a \rightarrow c := d \quad \gg \\ +&\ll \quad b \rightarrow e := f \quad \gg \end{aligned}$$

this means that if only one of a, b is true, then only the corresponding assignment is executed, and if both a and b are true, then the two assignments are executed as an atomic operation.

Any transition or combination of transitions that causes a write conflict is illegal. A write conflict occurs when there is more than one assignment to the same variable in the same atomic operation. A write conflict can occur if transitions are erroneously combined with the + or the * operator, or even within a single transitions if the same variable is used more than once on the left side of a multiassignment: for example, if two array elements, $a(i)$ and $a(j)$ are on the left side, and $i = j$ holds when the transition is enabled.

4.2.2 Modular Designs

A solution to dealing with increasingly complex designs is making them modular. A modular design can be flat, i.e. all modules communicating on the same level, or hierarchical – a module can contain other modules in its internal structure.

Synchronized Transitions supports modular and hierarchical designs through the use of **cells** - collections of state variables and transitions. In a hardware analogy, cells are like subcircuits. One cell may have several different instantiations just like a circuit may have many instances of the same subcircuit with different external connections. Likewise, a cell may be composed of other cells just as a circuit design may consist of a hierarchy of subcells. An ST program has a **top cell** that is the root of this cell hierarchy.

The interface of a cell to the outside world are its formal parameters. These can be of type *static* or *state*. Formal parameters that have been declared as STATE variables are bound to storage locations (like latches on a chip or memory locations of a process) when the cell is instantiated. Both the cell and its parent can read or write these parameters, which provides a mechanism of communication between cells. The values of STATIC parameters are bound when the cell is instantiated and are typically used to control the size of data structures or control recursive instantiations. The default for formal parameters is STATE.

ST also has arrays, records and functions, which have the usual interpretation. ST programs can be split into modules; each module consists of a definition part (file .std) and an implementation part (file .sti). This eases the writing and maintaining of the program; however, a module can contain descriptions of several cells, so *program* modules do not necessarily have a clear mapping to the *design* modules (the cells); this depends on the programmer's way of organizing his program files.

ST also allows a module to be written with the definition part in ST and the implementation part in C. This is a way to refine an abstract ST description to detailed code.

4.2.3 ST for Co-design

Modeling synchronous circuitry with +

Let's take a three-bit synchronous counter. In ST, it would be described as follows:

$$\ll incr \rightarrow qo := NOTq0 \gg \qquad (* \, t1 \, *)$$
$$+ \quad \ll incr \text{ AND } q0 \rightarrow q1 := NOTq1 \gg \qquad (* \, t2 \, *)$$
$$+ \quad \ll incr \text{ AND } q0 \text{ AND } q1 \rightarrow q2 := NOTq2 \gg \qquad (* \, t3 \, *)$$

A diagram of the digital circuitry to implement this is shown in figure 4.1. "expr$_i$" in this case mean just "NOT q$_i$", but they could be more complicated in other examples (here we could just have taken them from the negated output of the

D-latch, but we wanted to illustrate an approach); "enable$_i$" is high if transition i is enabled and low otherwise. For example, if $enable_1$ is high, on the next clock cycle the q0 latch will load $expr_1$, otherwise it will keep its old value.

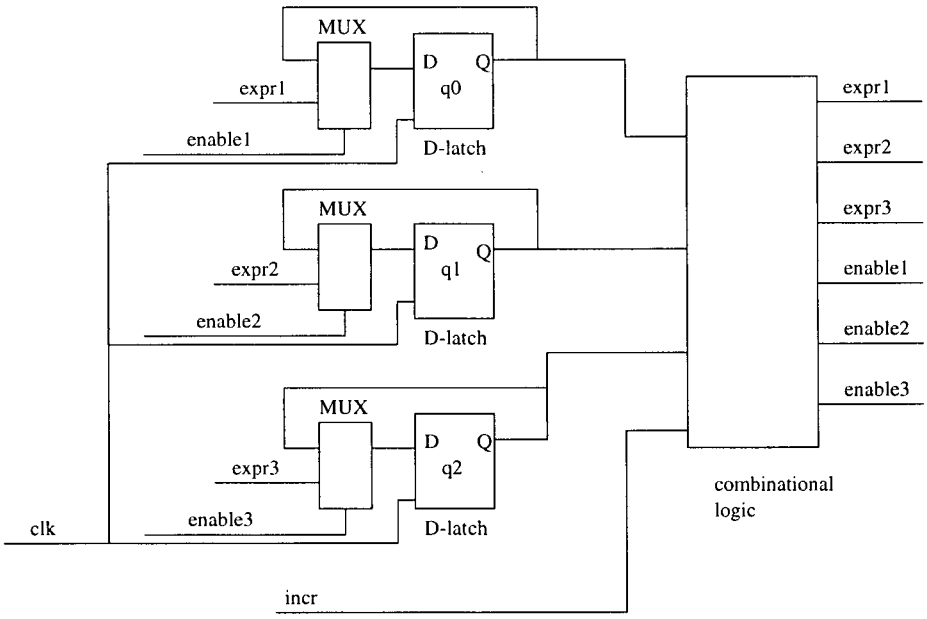

Figure 4.1: A direct mapping of the ST 3-bit counter to hardware

This hardware can be described in VHDL as follows:

```
entity counter is
  port( clk, inc: in std_logic;
        cnt: out std_logic_vector(2 downto 0) );
end counter;

architecture struct of counter is
  component mux
    port( in0, in1, select: in std_logic;
          out: out std_logic );
  end component;

  component d_latch
    port( d, clk: in std_logic;
          q: out std_logic );
```

```
  end component;

  component comb_logic
    port( in1, in2, in3, in4: in std_logic;
          out1, out2, out3, out4, out5, out6: out std_logic );
  end component;

  signal expr1, expr2, expr3: std_logic;
  signal enable1, enable2, enable3: std_logic;
  signal d0, d1, d2: std_logic;
  signal q: std_logic_vector(2 downto 0);

begin
  m1: mux( q(0), expr1, enable1, d0);
  m2: mux( q(1), expr2, enable2, d1);
  m3: mux( q(2), expr3, enable3, d2);
  q0: d_latch( d0, clk, q(0) );
  q1: d_latch( d1, clk, q(1) );
  q2: d_latch( d2, clk, q(2) );
  c: comb_logic( q(0), q(1), q(2), inc,
        expr1, expr2, expr3, enable1, enable2, enable3 );
end struct;
```

where the the multiplexer and the D-latch are regular multiplexers and D-latches and the combinational logic performs the logic operations described in the ST code.

Mixing combinators: non-determinism to model multiple clock domains

As mentioned, the train-bus controller is an interface between two synchronous buses operating at different clock rates. A description of this in ST would look like:

$$
\begin{aligned}
& (isa1 + isa2 + isa3 + ...) \\
\| \; & isaTOtrainbusFIFO() \\
\| \; & trainbusTOisaFIFO() \\
\| \; & (tb1 + tb2 + tb3 + ...)
\end{aligned}
$$

where $isa1, isa2, ...$ are transitions synchronous to the ISA-bus and $tb1, tb2, ...$ are transitions synchronous to the train-bus. Note that this ST model does not state the relative frequency of the ISA-bus and the train-bus and is robust to changes in the train-bus frequency. In other words, we do not have to worry about frequency details at this high-level description stage.

To refine this model for a particular clock frequency, even a specific clocking protocol, we can write:

$$
\begin{aligned}
& (isa1 + isa2 + isa3 + ...) \\
+ \;& isaTOtrainbusFIFO() \\
+ \;& trainbusTOisaFIFO() \\
+ \;& trainbusClockGenerator() \\
+ \;& \ll tbEdge1 \gg *(tb1a + tb1b + tb1c + ...) \\
+ \;& \ll tbEdge2 \gg *(tb2a + tb2b + tb2c + ...) \\
+ \;& \ll tbEdge3 \gg *(tb3a + tb3b + tb3c + ...) \\
+ \;& \ll tbEdge4 \gg *(tb4a + tb4b + tb4c + ...)
\end{aligned}
$$

The asynchronous combinator $\|$ has disappeared and the synchronous combinator $+$ is used instead, since we went from a high-level description of components operating at different speeds, to a refined (lower-level) description where we already decided about the clocking methodology.

Note there are transitions without any actions in the example above (e.g. $\ll tbEdge1 \gg$); when combined with the product operator *, such transitions act as guards for the transitions they are combined with. The execution of this code looks as follows: at every execution step, the following group of transitions will be executed as an atomic operation:

- every enabled transition from the *isa*-group;
- every enabled transition from the $trainbusTOisaFIFO$-group;
- every enabled transition from the $isaTOtrainbusFIFO$-group;
- the counter model is incremented; the counter divides the ISA-clock signal to generate the clocking required for the train-bus. In our case, this is a 4-phase clocking which means the events of interest are the rising and falling edges of two differential clocks ($tbEdge1$ to $tbEdge4$).
- whenever $tbEdge_i$ is true, every enabled transition from the group tb_i is executed.

The train-bus clock generator is modeled in a similar fashion. It consists of two cells, a counter modulo 21 and a counter modulo 4 that generates the two delayed clocks phi_1, phi_2 and the pulses corresponding to the 4 edges (rising and falling for the two clocks).

The "counter21" cell (in figure 4.2) is a value of a CELL type, its value is given by the initializer expression, which is roughly a lambda expression. The type definition for the CELL type defines the types and storage classes of the parameters

```
STATIC
 counter21: COUNTER21 =
  STATIC
       max: INTEGER = 21;
  STATE
            count: INTEGER;
  BEGIN
         << reset -> count := 1  >>
       + << NOT reset >> *
                ( << count := (count+1) MOD max>>
                + << inc :=  count=0 >>
                )
  END;
```

Figure 4.2: The counter modulo-21

for the cell. The code is straightforward: it generates a positive pulse for *inc* every 21 execution steps.

The counter4-cell (see figure 4.3) uses the output of the modulo-21 counter to toggle phi_1; and phi_2 follows phi_1 by a delay equal to the period of the signal *inc*. This means that if we use the four events corresponding to the 4 edges of the signals phi_1 and phi_2, they will be separated by periods equal to the *inc* period, and if we look at the set of 4 events, they will repeat after a period of $4 * period(inc)$.

The two cells are connected in the main cell of the controller hierarchy as shown in figure 4.4. The full code of this cell, with comments, can be found in appendix A; the purpose of the above fragment was just to illustrate that we need 3 of the 4 events to control the operations of some of the FIFO buffers, so we presented these actions in a simplified way.

Mixing combinators: non-determinism to model the environment

The non-determinism of ST can be used to model the environment for a design. For example, the train-bus controller should work with any legal software running on the PC, and we do not want to limit our model to describing one specific program. The ST model for the ISA interface can perform any sequence of ISA reads and writes that correspond to legal transactions with the bus controller. Figure 4.5 gives a state diagram of a "dispatcher" cell – a cell modeling the legal actions a train-set control software may take to send commands to the train-bus. Note that from state START, several actions may be taken without any condition specified on the arrows

```
STATIC
      counter4: COUNTER4 =
  STATE
      phi1, phi2: BOOLEAN;
  BEGIN
    << reset -> phi1, phi2 := FALSE, FALSE >>
  + << (NOT reset) AND inc >> * (
                 << phi1 := NOT phi2 >>
               + << phi2 := phi1 >>
               )
  + << ev1 :=  inc AND (NOT reset) AND phi1 AND (NOT phi2) >>
  + << ev2 :=  inc AND (NOT reset) AND phi1 AND phi2 >>
  + << ev3 :=  inc AND (NOT reset) AND (NOT phi1) AND phi2 >>
  + << ev4 :=  inc AND (NOT reset) AND (NOT phi1) AND (NOT phi2) >>
  END;
```

Figure 4.3: The differential clock generator

```
STATIC
bcCell: BCcell =        (* the model for the bus controller *)
        (* ... declaration of internal signals *)
  BEGIN
          counter21( reset, inc )
        + counter4( reset, inc, phi1, phi2, tbEdge1, tbEdge2,
                                                tbEdge3, tbEdge4 )
        + << tbEdge1 >> * (
              AdvanceAddressFIFO()
            + AdvanceCommandFIFO()
            )
        + << tbEdge3 >> * WriteTrainData()
        + << tbEdge4 >> * ReadTrainData()
     (* + ... *)
```

Figure 4.4: Instantiations of the counter cells

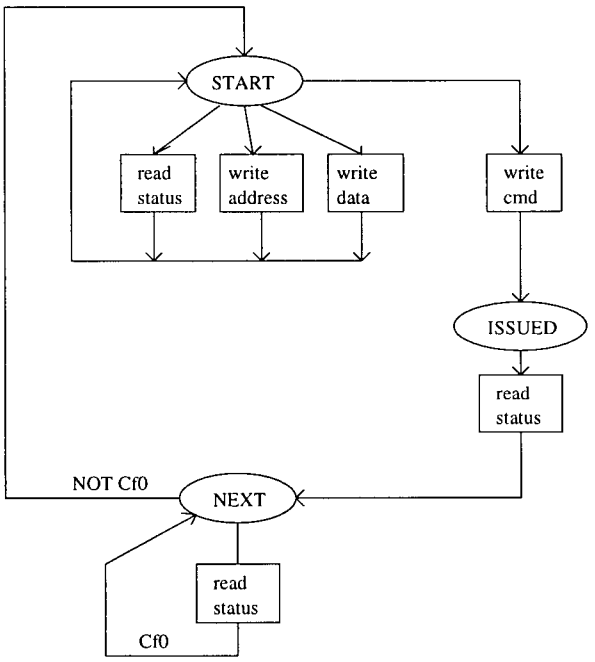

Figure 4.5: State Diagram for the Command Dispatcher

```
  << state = START >> * (
      ReadStatus()
   || WriteAddress()
   || WriteData()
   || WriteCmd() * << state := ISSUED >>
   )
+ << state = ISSUED -> state := NEXT >> *
      ReadStatus()
+ << state = NEXT >> * (
     << Cf0 >> * ReadStatus()
   + << NOT Cf0 >> * << state := START >>
   )
```

Figure 4.6: ST code for the dispatcher cell

towards these actions; this means that from state START, any of "read status", "write address", and "write data" can be done without advancing to the following state; only writing a command advances the state machine to ISSUED. After we issued a command, we cannot issue the next before we know the controller is ready to accept a new command, which is signaled by the Cf0 status bit being false. Cf0 is an acronym for "Command FIFO, stage 0"; Cf0 being true indicates that the stage that accepts a new command is full. So the state machine stays in the intermediary state "NEXT" until the status bit Cf0 indicates a new command can be dispatched, and the state machine moves back to START. Also note that since we have a choice of actions in the START state, it means the order of sending the data and address components is not specified; what's more, we can dispatch a new command without providing new data and/or address values; in that case, the last written data and address values will be the ones sent onto the train-bus.

The ST code for such a cell is presented in figure 4.6. The non-determinism of the choice between the actions that can be taken from the START state is modeled by using the asynchronous combinator ||.

We also need a model of a cell that can retire data requested by a previous read command. The state diagram for such a "retire" cell is given in figure 4.7. It is similar to the one for the dispatcher, but simpler because there is not a choice of actions that can be done from a certain state. A new response can only be read if another status bit, Rf1 (which stands for "response FIFO 1 full), is true. The corresponding code is in figure 4.8.

The application program will look like:

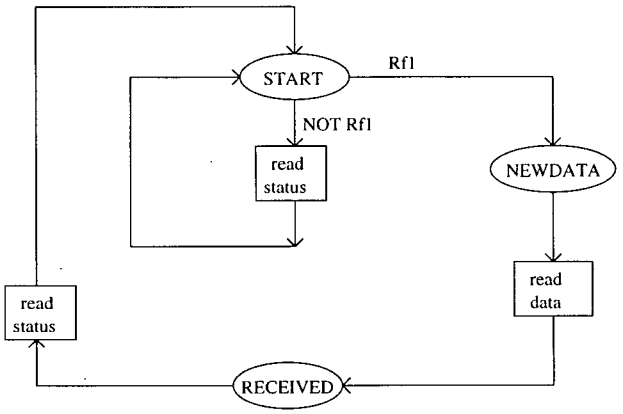

Figure 4.7: State Diagram for the Retiring of a Command

```
  << state = START >> * (
     << NOT Rf1 >> * ReadStatus()
   + << Rf1 >> * << state := NEWDATA >>
   )
+ << state = NEWDATA -> state := RECEIVED >> * ReadData()
+ << state = RECEIVED -> state := START >> * ReadStatus()
```

Figure 4.8: ST code for the retire cell

dispatchCell()
|| *retireCell*()

This allows the client to perform split-transaction operations – i.e., the client may dispatch a new command while one or more previous commands are still outstanding; the client is guaranteed to have the commands retired in the order they were dispatched.

4.3 Summary

After reviewing some of the problems that the co-design community is faced with, we are proposing the use of Synchronized Transitions as a specification language for co-design. The language has been extensively used for hardware design by other research groups, and the book published about ST [18] mentions the possibility of using it for co-design. We enumerate the features of ST that make it suitable for embedded system design, and give examples of the use of ST's synchronous and asynchronous combinators. The use of both combinators in the same design model has not been applied in previous research with ST. We explain the advantages of removing this limitation (i.e., not being able to mix the two types of combinators) by giving examples of how we apply this new method to our train-bus controller design.

Chapter 5

The Design

The bus controller is an interface between two buses running at different speeds and having different clocking methodologies and protocols. It consists essentially of buffers and logic to control the buffering so that the two protocols are respected.

5.1 ST model design

5.1.1 The FIFO Buffers

Sending commands to the train-bus

As described in chapters 3 and 4, the train-bus can be logically split into train-data, train-address and train-command. Sending a command means sending the three components; data and address are optional and the order of sending them does not matter (as shown in figure 4.5). Because the two buses run at different speeds – the ISA bus goes 84 times faster than the train bus – FIFOs are used to temporarily store the information until it can be sent out to the trainbus. A new command can only be sent if the FIFO is not full. The commands that are implemented so far are of type *read, write* and *idle.*

A register view of the bus controller

The PC sees the bus controller as a collection of registers it can write to/ read from. The four registers are *Data, Address, Command* and *Status*, and their addresses are (in hexadecimal) 0x300, 0x304, 0x308, 0x30C.

A register view of the bus controller is given in figure 5.1. The number of stages for each FIFO is shown there as well: 3 for Address and Command, 4 for Data-Write, 2 for Data-Read and Status-Read. A write to the Status register

causes a reset of the whole controller; the picture shows the two stages of the counter generating the clocks (see section 4.2.3, figures 4.2 and 4.3), which have all outputs set to "0" on reset. The data-write FIFO has an additional stage compared to the address and command ones, because of the four-phase clock - this FIFO is being read at a different clock edge.

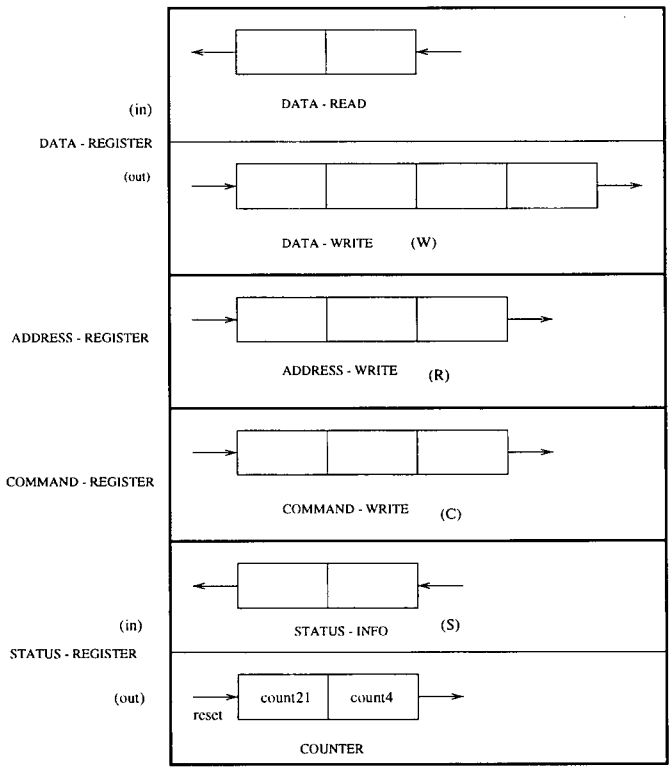

Figure 5.1: Bus Controller - Register View

The top stages of the Address (A) and Command (C) FIFOs are being read at tbEdge1, while the top stage of the Data-Write (W) FIFO is being read at tbEdge3. The A and C FIFOs advance when the shiftCf (shift Command FIFO) signal is high, and that signal goes high whenever the last C-FIFO stage is empty. But that stage could be empty when the last W stage is not. On the other hand, new commands may be sent by the application to the controller if the C-FIFO is not full. That means we could send a new data component to the W-FIFO when the W-FIFO is still full, if it doesn't advance on the same signal. This is why an additional W stage was added, to buffer the delay from tbEdge1 to tbEdge3. In

other words, this is in order to avoid the situation illustrated in figure 5.2, where we risk overwriting the write-data. For a better understanding see also figure 5.3; the signals names are explained in the following section and in appendix A. Since it is an ST cell instantiation diagram, it does not explicitly show the clock signal. The VHDL code has a clock signal (the controller operates on the ISA clock) input for every buffer showed in the figure (each square represents one FIFO buffer, the names and stages of the FIFOs are printed beneath the buffer-cells). The buses carrying the information to be stored in the latches (buffers) are shown as horizontal lines entering the left side of each buffer, while the FIFO control signals are shown as connected to the top side of each buffer (these are usually load or chip-select or output-enable signals; the command buffer, stage 2, is the special buffer described in section 5.1.2).

The design simulation worked correctly with only 2 stages for the A and C FIFOs and 3 stages for the W-FIFO, but several *idle* commands appeared on the train bus in-between the useful ones, which meant that too much time was lost waiting for the the first stage of the C-FIFO to become empty in order to send a new command. (The controller issues *idle* train-bus cycles, identified by the idle-command code, every time it does not have a command ready to be issued, or when the result FIFO is full and it therefore cannot record the result for a new command.) And since the ISA-bus runs at a higher frequency than the train-bus, it could provide the useful commands at a much higher rate than the train-bus could process them; this means the only problem were the insufficient FIFO stages, so I increased the FIFO depth; currently, sending 10 or 20 commands works without idle train-bus cycle insertion.

The buffer-cells used to model the FIFO are defined in the buffers.sti module (see appendix A), and their instantiations are shown in appendix B. Different buffers had to be used since some require load-signals, others are tristate buffers and require output-enable signals; some are 4-bit wide (C-FIFO), some have 2-bit inputs (the 2 status bits Rf1 and Cf0 in case of the Status-FIFO), others are 8-bit wide. Because of the ST syntax, declaring a general buffer and instantiating it in different ways would have been complicated. Also for debugging purposes, I wanted to be able to print different messages from inside different buffer cells, which would not have been possible for different instantiations of the same ST cell.

Section 4.2.3 gave a detailed description of how the interfacing of the two bus protocols is achieved (see especially figure 4.5). Figure 5.4 reminds the reader of the steps for sending/retiring commands from the train-bus; this is a diagram that was used as a sketch to write the ST-code for the module "driver.sti" (appendix A).

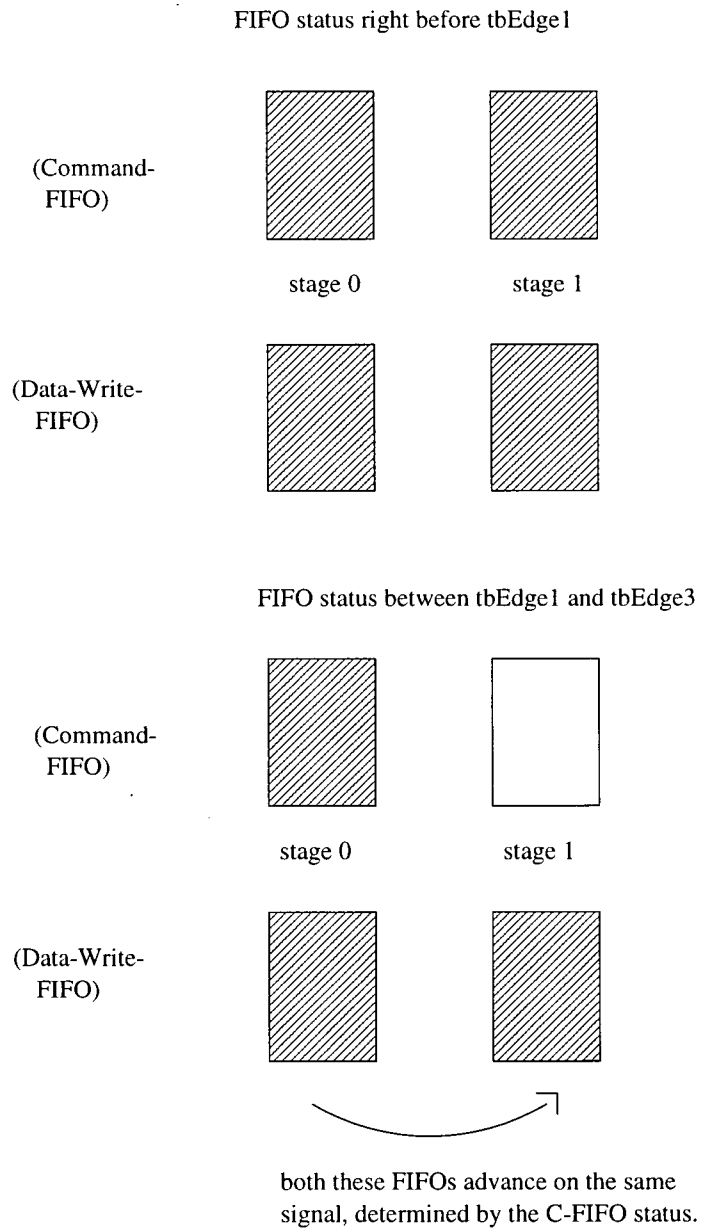

Figure 5.2: The need for an additional data-write buffer

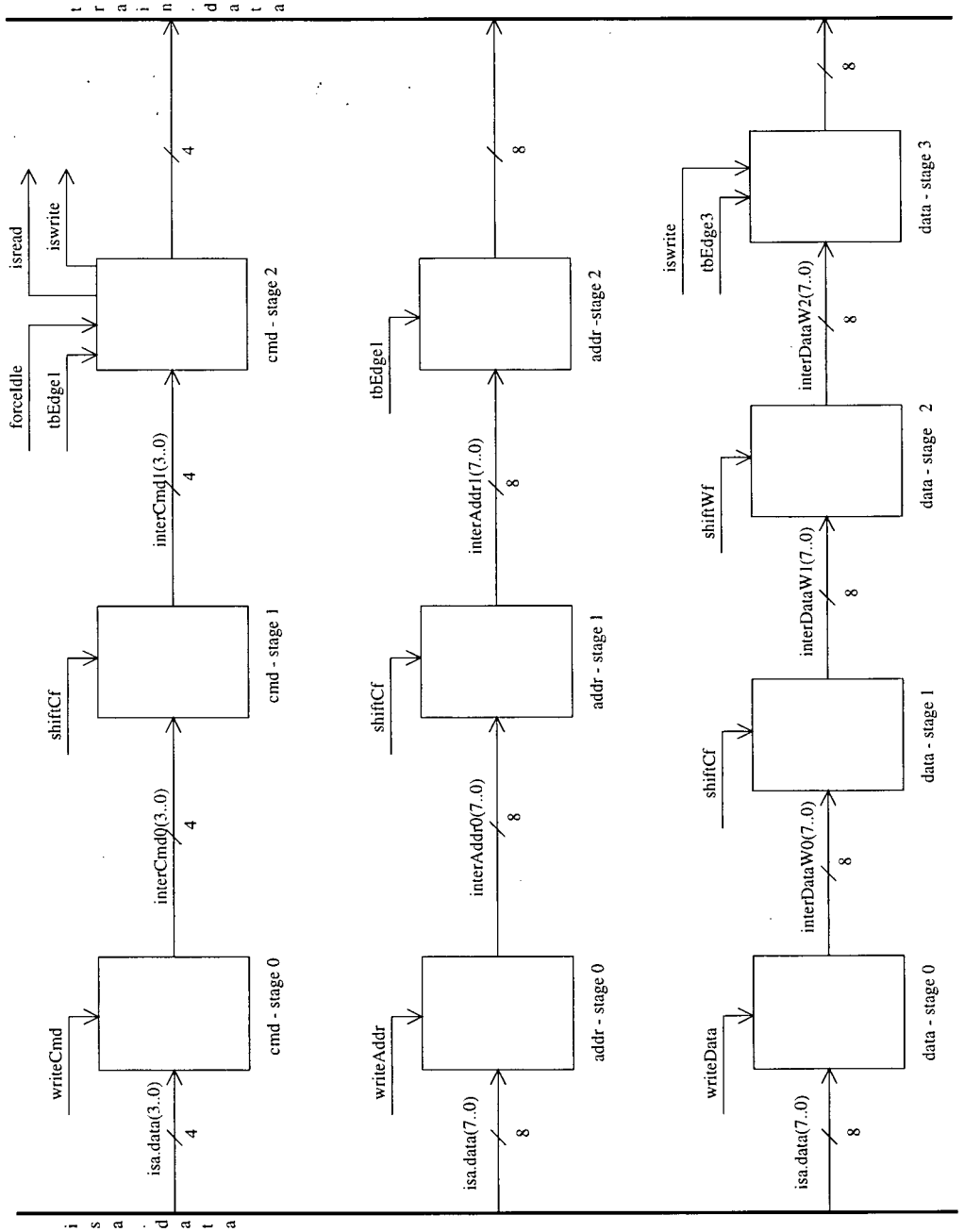

Figure 5.3: The "out-FIFOs": Cmd, Addr and Data-Write

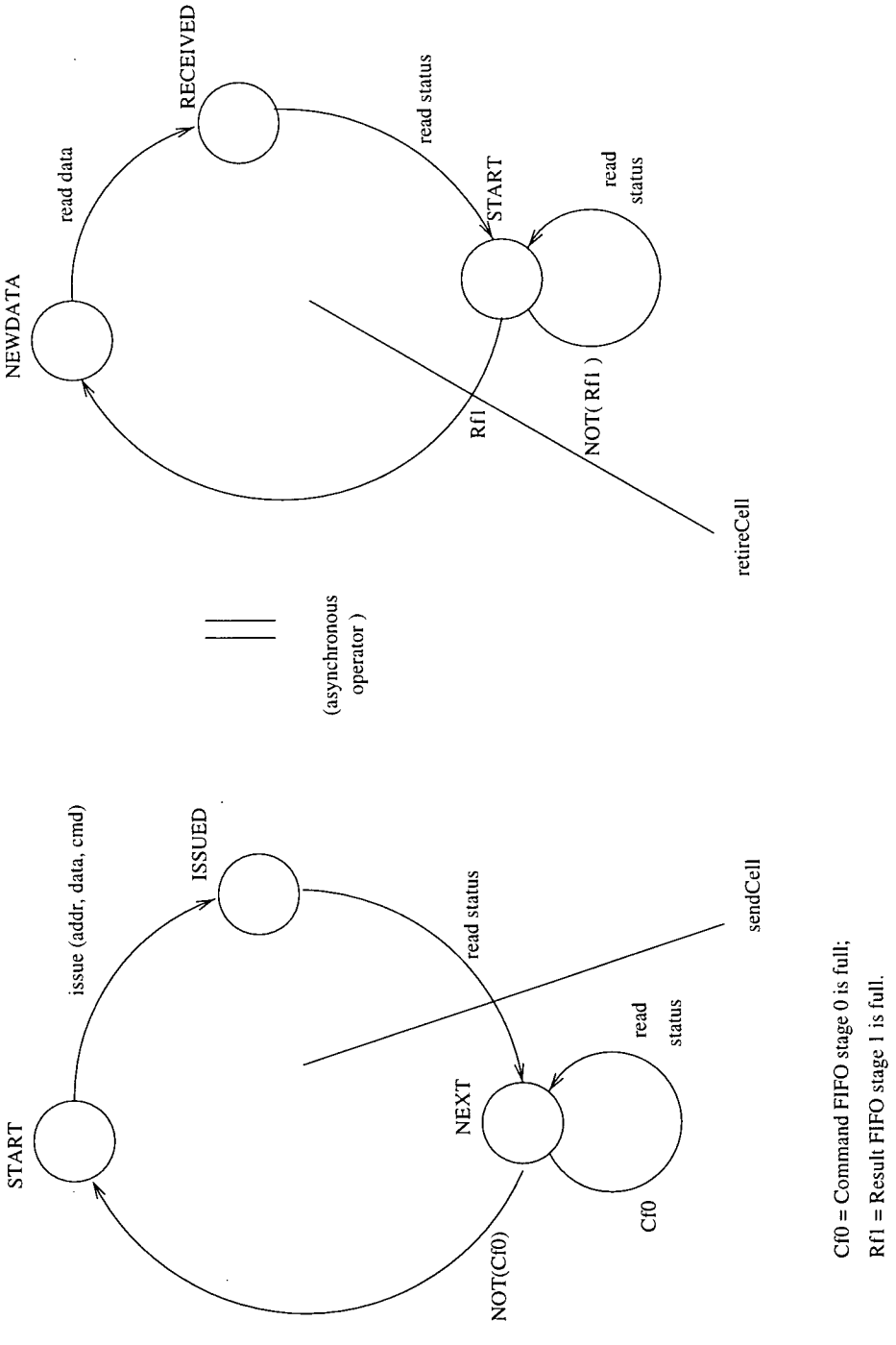

Figure 5.4: State Diagram for Module *driver.sti*

Reading the Status

At every clock cycle (every ST program step), the outputs Cf0 (command FIFO full) and Rf1 (result FIFO has data available) from the queue controller are latched into the first Status FIFO stage buffer. At any point the application decides to do a read from the Status Register, this information is available within the same ISA-IOread cycle .

Reading Response Data

Whenever the controller has result data available (Rf1=high), the application can do a read from the Data Register, which will cause the controller to anable the tristate outputs of the final stage of the Result FIFO to drive the ISA-data lines.

New response data from the train-bus is loaded on every tbEdge4 pulse into the first FIFO stage when the current command is a read. Current command means the last command that was sent out, i.e. the command sent out at the previous tbEdge1 pulse. The signals telling whether this was a read or a write are the "isread" and "iswrite" outputs from the special buffer that makes the last C FIFO stage.

The control of the FIFOs to ensure the above described operation is presented in the following section.

5.1.2 The Control Logic

Control logic actually means everything that's not a FIFO buffer.

Address Decoding

The bus controller is IO-mapped. It needs to know when it is being accessed by the program running on the PC, which is being done by first decoding the address - the *addrdec* cell - and then looking whether it is a write to or a read from that address - the *readwrite* cell.

The address decoder samples the ISA-address lines at the correct moment (defined by the state of the ISA lines AEN, BALE) and decodes it for TrainData, TrainAddress, TrainCommand or TrainStatus:

```
<< NOT aen AND ale >>
* (
     << addrIsTD := BAToInt( addr ) = trainData >>
   + << addrIsTA := BAToInt( addr ) = trainAddr >>
   + << addrIsTC := BAToInt( addr ) = trainCmd >>
```

```
    + << addrIsTS := BAToInt( addr ) = trainStatus >>
  )
```

BAToInt is an ST library function that takes an argument of type Boolean Array and converts it into an integer.

The read/write cell uses these signals and ISA-iow, ISA-ior to generate writeCmd, writeAddr, writeData, writeStatus, readData, and readStatus. Of the four registers that constitute the interface of the controller to the outside world, two may only be written (Cmd and Addr), two may be both written and read by the master (Data and Status).

```
<< writeAddr :=  addrIsTA AND (NOT iow) >>
      + << writeData := addrIsTD AND (NOT iow) >>
      + << writeCmd := addrIsTC AND (NOT iow) >>
      + << writeSts := addrIsTS AND (NOT iow) >>
      + << readData := addrIsTD AND (NOT ior) >>
      + << readSts := addrIsTS AND (NOT ior) >>
```

The Counters or Clock Dividers

Clock dividers are needed to generate the train-bus clocks from the ISA clock. This is done by two ST counter cells, "count21" and "count4" (see also section 4.2.3). This means the train-bus will run 84 times slower than the ISA-bus.

The counter modulo-4 is needed to generate 4 pulses, 21 ISA-clock cycles apart; it outputs these on lines tbEdge1, tbEdge2, tbEdge3, tbEdge4. In other words, these will be clock signals with a period equal to 84 ISA-clock cycles and with a pulse width of one ISA-clock cycle. The train-bus only needs the two differential clock signals tbclk1, tbclk2 so that the slave devices can detect rising and falling edges on these lines. But the controller also needs to take actions on certain edges so we use the same cell to detect them. However, tbEdge2 is not used inside the controller since only the slave devices act on this edge (see figure 3.5, section 3.4.3).

The Queue Controller

The queue controller, or FIFO controller, generates the FIFO command signals and also status information signals. FIFO command signals are

- *shiftCf* - shift Command FIFO
- *shiftRf* - shift Read FIFO
- *shiftWf* - shift Write FIFO

```
STATIC
 qcontroller: QCONTROLLER=
  STATE
   (* Cf1 = Command FIFO stage 1full;
    * Rf0 = Result-data FIFO stage 0 full;
    * Wf2 = Write-data FIFO stage 2 full *)
   Cf1, Rf0, Wf2: BOOLEAN;
```

Figure 5.5: The state variable declaration

- *forceIdle* - force an idle command out on the bus if no new command has been issued. This doesn't actually affect the FIFO advancing, but the output of the FIFO.

The shift-FIFO signals are generated based on space in the next FIFO stage; for this, we need to keep track of the status of the FIFO stages (empty/full). Some of these status variables are also necessary outside the queue controller cell, as actual status information about the bus controller; they must be available to the application sending the commands/reading the results, so that it knows when to send/retire a command. Retiring a command means reading the result requested by the command; the point at which the result is available is several ISA clock cycles away, which means the status has to be checked to see if it arrived. The status variables that tell the application program when the bus controller can accept new commands or has new data available are:

- *Cf0* - Command FIFO stage 0 is full;
- *Rf1* - Response FIFO stage 1 is full.

Since this cell is the most difficult to understand, I'm going to list its definition code here, even though it is a bit longer, so that the reader may recognize the above explanations in the code. The code is organized as follows:

- State-variable declarations: state variables indicating the state of some of the FIFOs; see figure 5.5
- function definitions (figure 5.6): I defined some functions just to make the cell body code easier to read. These functions are of type BoolFn0 (Boolean Function) and their meaning is:
  - "issuecmd" (issue command): we may only issue a new command – i.e., enable the outputs of the buffers connected to the train-bus – if there is

```
STATIC
  (* some functions to make the main cell body easier to read:*)
  (* issue command: *)
  issuecmd: BoolFn0 = BEGIN Cf1 AND (NOT Rf0) END;
  (* advance Command-FIFO: *)
  advancefifoc: BoolFn0 = BEGIN Cf0 AND (NOT Cf1) END;
  (* advance Result-FIFO (i.e. data-read-FIFO): *)
  advancefifor: BoolFn0 = BEGIN Rf0 AND (NOT Rf1) END;
  (* data available *)
  availdata: BoolFn0 = BEGIN isread AND tbEdge4 END;
```

Figure 5.6: Function definitions for the *qcontroller* cell

a new command sitting in the top C-FIFO stage and there is room in the bottom R-FIFO stage to record the new result if necessary;

- "advancefifoc" (advance Command FIFO): if stage 0 is full and stage 1 is empty;
- "advancefifor" (advance ResultFIFO): if stage 0 is full and stage 1 is empty;
- "availdata" (data available): response data from the slave device is loaded from the train-bus into the R-FIFO on tbEdge4 if the current command on the train-bus is a read-command, i.e., if a read-command was issued onto the train-bus on the last tbEdge1

- transitions for the Command-FIFO control (figure 5.7): If the C-FIFO is advancing and we are not writing to it, the bottom stage (0) becomes empty (transition t1). It becomes full when there is a write to the Command register (indicated by the "writecmd" signal) (t2). If stage 0 is full, stage 1 is empty ("advanceficoc"-function evaluates to true) and we are not writing to the Command register, we can advance the FIFO (t4). Finally, if we issue the content of the top C-FIFO stage to the train-bus, and we are not currently advancing the FIFO, the top stage becomes empty (t5). (notations: Cf0 = Command FIFO stage 0, Cf1 = stage 1, shiftCf = "shift Command FIFO")

- transitions for the Data-Write-FIFO control (figure 5.8): On tbEdge3, write-data is let out to the train-bus, and since the W-FIFO (write data-FIFO) advances only on tbEdge1, the top stage (2) of this FIFO becomes empty (transition t1). The following condition results from the fact that the W-FIFO advances stage 0 to 1 synchronously with the C-FIFO, but stage 1 to

```
BEGIN
      << reset OR ((NOT writeCmd) AND advancefifoc())
        -> Cf0 := FALSE >>                              (* t1 *)
    + << (NOT reset) AND writeCmd -> Cf0 := TRUE >>     (* t2 *)
    + << shiftCf := Cf0 AND NOT Cf1 >>
        (* advance the FIFO: *)
    + << NOT reset AND advancefifoc() AND (NOT writeCmd)
                  -> Cf1 := Cf0 >>                      (* t4 *)
    + << reset OR (NOT advancefifoc() AND issuecmd() AND
        tbEdge1) -> Cf1 := FALSE >>                     (* t5 *)
```

Figure 5.7: Command FIFO control signals

```
    + << reset OR tbEdge3 -> Wf2 := FALSE >>            (* t1 *)
    + << NOT reset AND shiftWf AND NOT shiftCf
                           -> Wf2 := TRUE >>            (* t2 *)
    + << shiftWf := Cf1 AND NOT Wf2 >>                  (* t3 *)
```

Figure 5.8: Data-Write FIFO control signals

2 separately, depending on the state of W-stage 2; so the W-FIFO advances: stage 0 to 1 when *shiftCf* is true, stage 1 to 2 when *shiftWf* is true. It is okay to advance stage 1 to 2 if we are not currently writing to stage 1 (transition t2). For the *shiftWf* assignment, Cf1 indicates the same as a supposed variable Wf1; that is, the state of stage 1 of the C-FIFO is the same as the state of stage 1 of the W-FIFO since for stage 0 to 1 the W-FIFO is synchronous with the C-FIFO (transition t3).

- transitions for the Result-FIFO control (figure 5.9): If the R-FIFO (result, or data-read FIFO) advances and there is not new data to be loaded, stage 0 becomes empty (transition t1). It becomes full when new data is written to the bottom stage (0), on tbEdge4 if the current command, i.e., the one that was issued to the train-bus on the previous tbEdge1, is a read (t2). If the current ISA-IO-read cycle is one from the controller DATA register, and the R-FIFO is not currently advancing, the top stage, 1, becomes empty (t4). If stage 0 is full, stage 1 is empty, and we are not writing to stage 0, it is okay to advance the FIFO (t5).

- transitions for forcing an idle train-bus cycle (figure 5.10): If the conditions for

```
+ << reset OR (NOT (tbEdge4 AND isread) AND advancefifor())
   -> Rf0 := FALSE >>                              (* t1 *)
+ << NOT reset AND isread AND tbEdge4
   -> Rf0 := TRUE >>                               (* t2 *)
+ << shiftRf := Rf0 AND NOT Rf1 >>
+ << reset OR (readData AND NOT advancefifor())
                          -> Rf1 := FALSE >>       (* t4 *)
+ << NOT reset AND advancefifor() AND NOT(tbEdge4
    AND isread) -> Rf1 := Rf0 >>                   (* t5 *)
```

Figure 5.9: Result FIFO control signals

```
+ << reset OR (NOT issuecmd() )
    -> forceIdle := TRUE >>
+ << (NOT reset) AND issuecmd() -> forceIdle := FALSE >>
```

Figure 5.10: *Qcontroller* module: forcing an idle train-bus cycle

issuing a new command are not met (see "issuecmd"-function), the controller issues an idle command out on the train-bus.

Forcing an idle command on the train-bus is achieved by controlling the special buffer that makes the last C-FIFO stage, with this "forceIdle" signal from the queue controller – figure 5.11 presents the ST-code for the buffer. This buffer is also the one that generates the isread and iswrite signals used as inputs in the *qcontroller* cell. It generates these signals by just looking at (a) the least significant bit of the command code it contains, which indicates a read or a write, (b) whether it had to force an idle or not, in which case none of the signals is true. "IntToBA()" is a function that takes an Integer and translates it into a Boolean Array with a specified number of bits (in our case 4, the bit-width of the train-command lines)

The *contrlogic* cell

This is not the most proper name for this cell, since it is not the only control logic component, but it is not specialized on certain things either like the others. It is just combinational logic generating the reset signal and output enable signals. The output enable signals are for the buffers with outputs connected to the ISA-data lines. When the board is not addressed by the ISA bus, it should not be driving the ISA data lines at all. The two buffers with outputs connected to the ISA data-bus

```
buf2c: Buffer4spec =
  STATE
      cmdIdle: INTEGER = 3;
  STATIC
     (* the least significant bit of the command, in(0),
      * indicates a read or a write command *)
      iswritefn: BoolFn0 =
          BEGIN NOT force AND NOT in(0) ) = cmdWrite) END;
      isreadfn:  BoolFn0 =
          BEGIN NOT force AND in(0) ) = cmdRead) END;

 BEGIN        (* forces idle when force is TRUE *)
      << cs >> *
              ( << NOT force -> out := in >>
              + << force -> out := IntToBA( cmdIdle, 4) >>
              + << iswrite := iswritefn() >>
              + << isread := isreadfn() >>
              )
 END;
```

Figure 5.11: The special buffer for Command-FIFO stage 2

are the second (or final) stages of the Status and the Reply-Data FIFOs.

5.1.3 ST simulations

In order to test the ST program, a *driver* module was necessary to simulate the master application; that is, to send commands to devices and retire them (read response data if any). This module was built according to the state diagram from figure 5.4. We also needed an ISA-bus simulator (modules *isa* and *IO*, the latter containing the *inByte* and *outByte* cells that model the communication between the application and the ISA-bus).

Another necessary component was a module simulating a slave device connected to the train-bus. This is the *device* module; all it needs to do is detect whether it is being addressed (by comparing the information on the train-address lines to its own identification information) and respond to the different commands (provide a response in case it was a read).

5.2 Implementation Decisions

As justified in chapter 3, we are using a PC as the host computer; the bus controller is implemented as an ISA-board. The general idea was to use programmable logic on such a board. This implied translating the ST-code into VHDL-code that could automatically be synthesized by available CAD tools.

For previous versions of the bus controller, Progammable Array Logic (PAL) integrated circuits were used (AMD PAL22V10 – see databook [1]). For a more elegant design we initially decided to use the MACH-4 family Complex Progammable Logic Devices (CPLDs), MACH445 [2]. That is, we wanted to use more powerful chips and the choice was for the mentioned ones because at that point the department already had licenses for the necessary CAD tools to program those.

Since the download program was using around 80% of one device's resources, we decided to use two and forced the CAD-software to partition the design onto 2 chips. We took this decision in order to leave room for further changes in the design; since the controller is a hardware piece for an experimentation project, it will almost certainly be further modified in the future. Debugging also needs to be kept in mind. However, the MACH-4 family has a reputation for being hard to reprogram with the same pinout. Although I tried out the procedure succesfully with a minor change, reprogramming it later with a much different code did not succeed. Also, in the meantime partitioning is not supported anymore, so every time we would need to re-program the chip, we would have to do manual partitioning. This is why recently

we decided to change the implementation from using the MACH445's to using only one Xilinx Spartan chip [22].

5.3 From ST to VHDL

After successfully simulating the ST program, the next step is to translate the ST code into VHDL. Several things have to be taken into consideration here:

5.3.1 Clock signal

First of all, for a program, the clock is implicit: it is the rate at which instructions are executed. In ST, at every execution step, one or more transitions are picked for execution. When going from ST to VHDL, the clock has to be explicitly specified as a signal connecting all components (the equivalent of the cells in ST).

5.3.2 Sensitivity lists

Another problem is raised by the sensitivity lists of VHDL processes. Sensitivity list means a list of signals to which a process is sensitive ([3]). When any of these signals change value, the process resumes execution of the sequential statements; after executing the last statement, the process suspends again until a new change occurs on one of the sensitivity list signals.

Now when going from ST to VHDL, one would be tempted to include all formal parameters of a cell into the sensitivity list for the VHDL component representing that cell. But this is not how the ST program executes: in ST, at every timestep, one or more transitions have their guards evaluated and are executed. This rather corresponds to having only the global clock signal on the sensitivity list of any VHDL process. The decision about what should be included on the sensitivity list depends on how cells are executed in ST; this issue is addressed in the following section.

5.3.3 On mixing ST combinators

In all previous work where ST has been used, mixing synchronous and asynchronous combinators has been considered illegal by the compiler ([18]). The ST version used in this thesis allows mixing synchronous and asynchronous combinators.

The need and advantages of mixing the synchronous and asynchronous combinators has been shown in chapter 4. Suppose we have an ST program where we model subcircuits working at the same clock rate; in this case, we use the synchronous combinator +. Within each cell however we may be using the asynchronous

"bar" combinator, ||. From the definition of the + combinator, it results that at every time step, all transitions combined with + whose guards are enabled, will be executed in one atomic operation. The question arises here whether to regard the cell as a transparent hull for the transitions it groups together, or as an indivisible body of transitions. In the former case, it can be represented as

$$\begin{aligned} &(\ll \ \mathit{action} \ \gg \\ &\|\ll \ \mathit{action} \ \gg) \\ &+\,(\ll \ \mathit{action} \ \gg \\ &\|\ll \ \mathit{action} \ \gg) \end{aligned}$$

and the way to interpret it is to non-deterministically pick one transition from every cell and execute all of them that have their guards satisfied, in one atomic step.

In the latter case, if we view the cells as indivisible action bodies, then once a cell is "fired", it is like a spawned process, meaning its enabled transitions – the ones combined with || – execute repeatedly until no more are enabled; we could say we let the whole cell "settle down", before we fire it again. (this is from a discussion with Joergen Staunstrup, the author of [18], this last summer (1998) at the Danish Technical University in Lyngby, Denmark).

The convention used in the UBC ST compiler is to view cells like transparent hulls of the transitions grouped in its body. In other words, we do not let the cells "settle down", but, if there are transitions combined with ||, we pick some transition each time the subcell is "called" in the top cell.

This convention is the reason why, when I went from ST to VHDL, I only included the clock signal on the sensitivity lists of most of the processes (except combinatorial logic and some components that use other signals for latching values). This is because in our version of ST, we do not spend more than one clock cycle in a cell at a time, so the next time we pick a transition of that cell, we will have all the updated information about all the other signals that are input parameters to the cell.

5.4 Summary

I described the process of refining the high-level specification of the controller, some issues I encountered and how solving them affected decisions about implementation details – e.g. FIFO depths. I further described the process of translating the simulated ST version to VHDL; the decisions I took, based on the conventions we made in order to be able to apply the mixing of ST combinators.

Chapter 6

Evaluations and future work

This project had two main purposes, as presented in the introduction: to produce a hardware piece necessary for further development of a model train set for real-time experiments, and to experiment with a new hardware-software codesign methodology.

6.1 Testing the train controller board

The PC-board, as tested so far, now corresponds to the requirements. Although it was not possible yet to test it with the train-bus connected to the board and devices connected to the train-bus, I ran a few tests by sending write and read commands and looking at the train-bus outputs with an oscilloscope. The testing applications were written in DJGPP, a version of the Unix C++ compiler for the PC. One of the test programs is presented in figures 6.1 and 6.2

These tests included:

- Doing sequences of writes to the train-bus, alternating the values for address and data (I used values 0x00 and 0xff). This works correctly; I checked by using the oscilloscope to see the transitions on the train-bus lines.
- Doing long sequences of write commands works without idle train-bus cycles inserted by the controller; this shows that the FIFO depth was chosen big enough (see section 5.1.1).
- Inserting (forcing) an idle command in a sequence of useful read/write was checked by looking at how the train-command lines change
- Executing a program that takes the following actions:

```
#include <stdio.h>
#include <pc.h>
#define TDATA 0x0300
#define TADDR 0x0304
#define TCMD 0x0308
#define TSTATUS 0x030C
#define DATA0 0x00
#define DATA1 0xFF
#define ADDR0 0x00
#define ADDR1 0xFF

void reset_controller( void )
{
 outportb( TSTATUS, 0 ); /* doesn't matter what we write,
                            just a write to the Status Register
                            resets the controller */
}

void getstatus( unsigned char *cf_full, unsigned char *rf_empty)
{
 unsigned char status;
 status = inportb(TSTATUS);
 *cf_full = (unsigned char)(status && 0x01);
 *rf_empty = (unsigned char)(!(status && 0x02));
}

void issuecmd( unsigned char addr, unsigned char cmd,
               unsigned char data )
/* addr, data are optional here */
{
 outportb( TADDR, addr );
 outportb( TDATA, data );
 outportb( TCMD, cmd );
}
```

Figure 6.1: Sample test application (in C++) for the bus controller: constant and function definitions

```
void main(){
 unsigned char data;
 unsigned char cf_full=0, rf_empty=1;
 int i;
 reset_controller();
 printf("\n controller reset");
 for (i=0; i<20; i++) {
  do {
        getstatus(&cf_full, &rf_empty);
    } while (cf_full);
 issuecmd( ADDR1, 0x04, DATA1 );
  do {
        getstatus(&cf_full, &rf_empty);
    } while (cf_full);
 issuecmd( ADDR0, 0x04, DATA0 );
 }
}
```

Figure 6.2: Sample test application (in C++) for the bus controller: main function body

- write 0xff;
- read data;
- write 0x00;
- read data;

also works, where "data" always returns the previously written value, which is because the controller sees the train-bus as a big capacitor. This feature was seen in the following experiments:

- when writing 0xff and no action afterwards, the train-data lines go to logic "1" for a short pulse then gradually fall towards "0".
- also for the read-after-write experiment, if I had the probe on train-data line 0, I read 0xfe instead of 0xff, because the data-write buffer connected to it was tristated and touching the line with the probe discharged it.

I have tried several different sequences of alternating read and write commands and I believe the controller works according to the specification; however, testing read

commands really requires a device providing responses from the other end of the train-bus. This is addressed in section 6.4.

For testing the correct generation of the train-bus clock signal I did not need any application program; just inserting the controller card into the ISA connector and programming the FPGA is what it takes to get the clock signals at the train-bus connector pins. All lines tbclk1, tbclk2, plus the signals that are internal to the controller but essential to the correct functioning, *ev1*, *ev3*, and *ev4*, look correct.

I made a header (2x15 pins) geometry that I used to connect all the unused pins of the FPGA, so that if I needed to test or debug the board I could add internal signals to the outputs of the FPGA and thus observe their behaviour on the oscilloscope. (Because of the very small dimensions of the FPGA package, it is extremely difficult and probably not very reliable to attach an oscilloscope probe to its pins.) To test *ev1, ev3, ev4* I connected these to some of the 2x15-header pins.

These test pins were very helpful for some of the problems I encountered:

- I first programmed the device only with the code for generating the clock signals for the train-bus and the event detection of the 4 edges of those two clocks. The two clock lines did not look correct at all, and from the oscilloscope display I saw it was because of erroneous pulses that were about one ISA-clock cycle long. This made me follow the advice of a colleague and use the VHDL condition "clk'event"; even though I thought it was redundant if *clk* was included on the sensitivity list of the process. It was the only thing I could think of trying because this part of the code had looked perfectly correct when I simulated it. It works correctly now, with "clk'event".

- The PC ceased to accept keyboard input when the FPGA on my board was fully programmed. The only way for my board to interfere with the PC was by driving the ISA-data lines, so I connected this observation with the fact that the synthesizing CAD-tools gave a warning: "Unknown port type for isadata", although I used the correct syntax. I also remembered that when I previously used a Synopsys product for compiling VHDL code (when trying to program the MACH445s), I was not able to use two different tristate buffers to alternatively drive the same line; at that time I had to rewrite the code so that in one of my components I was multiplexing the two internal buses. So now I did the same: I inserted a multiplexer so that only one tristate buffer was connected to each output ISA-data port. This worked correctly.

- When trying to issue write commands, I did not see the correct behaviour on the train-bus lines at first. Then I added all necessary debugging information – signals indicating the states of the Command FIFO, etc – as output port

connections so that I could see how their values changed. This is how I realized that in the translation from ST to VHDL, at some point I had omitted a "NOT" in the code for the queue controller. Once I corrected that too, all the tests I did (as mentioned above) were satisfactory.

6.1.1 Problems I encountered while going through the whole design process

One conclusion from my experience is that the time it took me to write and debug the ST code for the controller, or the VHDL code, was noticeably smaller in comparison to the time it took me to design a PC-board and successfully program the device on the board. This is partly because it was the first time I ever undertook such a project. After deciding to change the board, in other words when I was forced to go through the same process once again, it went much faster. I was then stuck with a Xilinx download cable/software bug (Xilinx web solution record #3701) which took me one week to find and solve (with technical support help). I found the reason for this experience (similar to the one programming the MACHs) to be the fact that CAD-tools have to be marketed very fast because of the fierce competition, and thus bugs are detected and fixed after the software is already in use.

The solution to the Xilinx downloading tools bug – the software patch indicated on the web did not solve the problem – was to disconnect the PROG wire of the download cable from the board; and to solder a pull-up resistor (2.2 KOhm is what I used) for that pin. I also needed a pull-up resistor for the DONE pin; in the current state, I just soldered a wire to the pull-up I had for the "response" line of the train-bus, because we are not currently using it anyway. This needs to be changed if we will actually connect the train-bus.

6.2 ST for co-design

I started work on this design virtually without any previous experience either in ST or in VHDL. That is, I had learned about hardware description languages (including VHDL) during my undergrad courses and I had used Verilog in a course project, but I never actually went all the way to implement a design in any HDL. At least I had had some exposure to VHDL, while ST was a totally new programming environment. However I did not find it difficult to adjust to it. A few simple homework exercises I did in the Formal Verification course where I learned about ST were enough to help me understand this new notion of non-determinism in a program. Once I grasped the concept, I came to think it is essential in correctly modeling real-time applications.

Since ST is not supported by any automated synthesis tool, I had to model my design in ST, simulate it and debug it; after I was happy with the behaviour of the model, I had to translate this into VHDL. This means the final version was totally deterministic. But the modules I used to simulate the controller (the ISA-bus model, application program model, train-bus device model) did make use of the possibility to model non-determinism in ST. This I believe to be a strong argument to use ST for co-design: the ability to make good and reliable simulations by correctly modeling the real-time environment that the design has to interact with.

I firmly sustain this viewpoint since I also had to simulate and debug the VHDL version of my design (since the translation is not done automatically, this process is prone to errors). The worst problem I had with VHDL was the way in which different ways of describing things that should essentially synthesize to the same hardware, resulted in such different behaviours after synthesizing (see also the problems I had to correct, section 6.1).

6.3 Conclusions

I used Synchronized Transitions to design a bus controller – a PC-ISA board with a XILINX Spartan FPGA – without having had any previous hardware experience (although several course projects in the past included designing some piece of hardware, I have never actually implemented and tested anything). I found ST very helpful in this process and I was finally able to see that the implemented design, the actual hardware, works correctly.

I think ST is a good choice for a co-design specification language since the specification that one can simulate does not depend on whether the model will be implemented in hardware or in software. It can therefore also provide a means of communication between hardware and software teams that would work on the same project. For my particular design case, I found it very helpful in dealing with interfacing between two components running at different speeds, which I believe is due to the possibility of expressing non-determinism in ST.

6.4 Future work

6.4.1 Completing the train-bus design

As mentioned in section 6.1, a complete test of the device requires another device to be present at the other end of the train-bus, to respond to commands sent by the application on the host PC. A first step would be to program a few simple Programmable Array Logic devices (like the 22V10 [1]) and connect them to the

train-bus connector on the controller board. This way it would be possible to test the Result-FIFO.

The next step, after making sure the controller works correctly, would be to actually connect the train-bus itself (several meters of ribbon cable). This is in order to test if the 4-phase protocol we proposed is a reliable communication protocol. We further need to design the devices that will be connected to the train-bus – speed controller, position controller, track switch controller – in the same modular manner and responding to the new protocol.

The final task will be to write the software for controlling the trains. Here ST would be a way to model the application before implementing it.

6.4.2 Better CAD support for ST based design

From my current experience, I believe that from the information contained in the ST model of the design, much of the final design can be generated automatically. More precisely, after deciding how to implement the design (i.e. partitioning decisions), most of the schematic could be generated automatically – in our case, the connections between the FPGA and the ISA-bus on one side, the FPGA and the train-bus on the other side, are all described in the ST code. (Of course the analog parts are not included there - like amplifiers or current limiters for some of the train-bus lines, or decoupling capacitors for the FPGA.) Also, all the information for generating the download file for the programmable device (the FPGA) is included in the ST.

The most useful tool would therefore be an automatic ST-to-VHDL translator, or better yet, ST-to-EDIF. This may constitute a future research objective.

Another very useful feature would be to be able to see simulation results of hardware modeled in ST, as waveforms. Currently, the way to simulate ST code is by inserting "WriteString" (a function that writes a string to the standard output) in places of interest in the code, which seems rather awkward. For instance, if I wanted to see how a certain digital signal behaved, I had to make the program write out the value of the signal every time it changed.

I think that the availability of such support tools would make ST a very convenient co-design tool that would offer a reliable specification language, easy to learn and use, with high modeling power for real-time applications; and the possibility of going through a safer stepwise refinement process from specification to implementation.

Bibliography

[1] Advanced Micro Devices, Inc. *PAL Device Data Book*, 1992.

[2] Advanced Micro Devices, Inc. *Mach 3 and 4 Family Data Book*, 1994.

[3] Peter J. Ashenden. *The Designer's Guide to VHDL.* Morgan Kaufmann Publishers, San Francisco, 1992.

[4] R. Camposano, D. Knapp, and D. MacMillan. *Hardware/Software Co-Design.* Kluwer Academic Publishers - NATO ASI, 1997.

[5] Daniel D. Gajski, Jianwen Zhu, and Rainer Doemer. *Hardware/Software Co-Design: Principles and Practice*, chapter Essential Issues in Co-Design. Kluwer Academic Publishers, 1997.

[6] R. Gerber, W. Pugh, and M. Saksena. Parametric dispatching of hard real-time tasks. *IEEE Transactions on Computers*, 44(3), 1995.

[7] R. Gupta and G. DeMicheli. Hardware-software co-synthesis for digital systems. *IEEE Design & Test*, 10, September 1993.

[8] Asawaree Kalavade and Edward A. Lee. A hardware-software codesign methodology for DSP applications. *IEEE Design & Test*, 10, September 1993.

[9] Ed Lansinger. Developing an engine control system. *Circuit Cellar INK*, September 1995.

[10] Trevor W. S. Lee, Mark Greenstreet, and Carl-Johan Seger. Automatic verification of asynchronous circuits. *IEEE Design & Test*, 12, Spring 1995.

[11] J. Madsen, J. Grode, P. V. Knudsen, M. E. Petersen, and A. Haxthausen. Lycos: the lyngby co-synthesis system. *Design Automation for Embedded Systems*, 2(2), 1997.

[12] Achim Oesterling, Thomas Benner, Rolf Ernst, Dirk Hermann, Thomas Scholz, and Wei Ye. *Hardware/Software Co-Design: Principles and Practice*, chapter The Cosyma System. Kluwer Academic Publishers, 1997.

[13] Tarik Ono-Tesfaye, Christoph Kern, and Mark R. Greenstreet. Verifying a self-timed divider. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1998.

[14] Anders P. Ravn and Jørgen Staunstrup. Synchronized Transitions. Technical Report DAIMI PM-219, Computer Science Department, Århus University, Århus, Denmark, January 1987.

[15] K. Rompaey, D. Verkest, I. Bolsens, and H. D. Man. Coware - a design enviroment for heterogenous hardware/software systems. In *Proceedings of the European Design Automation Conference*, Geneve, September 1996.

[16] D&T Roundtable. Hardware-software codesign. *IEEE Design & Test of Computers*, january-march 1997.

[17] Edward Solari. *ISA & EISA - Theory and Operation.* Annabooks, 1992.

[18] Joergen STaunstrup. *A formal approach to hardware design.* Kluwer Academic Publishers, Boston, 1994.

[19] Joergen Staunstrup. *Hardware/Software Co-Design: Principles and Practice*, chapter Design Specification and Verification. Kluwer Academic Publishers, 1997.

[20] Wayne Wolf. Lessons from the design of a pc-based private branch exchange. *Design Automation for Embedded Systems*, 1, 1996.

[21] Wayne Wolf. *Hardware/Software Co-Design: Principles and Practice*, chapter Hardware/Software Cosynthesis Algorithms. Kluwer Academic Publishers, 1997.

[22] Xilinx, Inc. *Spartan and SpartanXL Families Field Programmable Gate Arrays*, 1998.

Appendix A

ST code for the controller

The ST code presented is a refined specification version. Its refinement degree corresponds to the point where this project took over a class project started by another student. Changes were made to adapt the design to a more reliable protocol as described in section 3.5.

```
(*************************************************)
(*
* the definition module for the counter modulo-21
*
*)
DEFINITION MODULE counter21;
 FROM useful IMPORT BoolArray;
 EXPORT counter21;

 TYPE
   COUNTER21 =  CELL( reset: BOOLEAN; q: BoolArray;
                                          countmax: BOOLEAN );

 STATIC
  counter21: COUNTER21;

END.
```

```
(************************************************)
(*
* the implementation module for the counter modulo-21
*
*)
IMPLEMENTATION MODULE counter21;
 TYPE
     COUNTER21 = CELL( reset, countmax: BOOLEAN );

 STATIC
  counter21: COUNTER21 =
   STATIC
         max: INTEGER = 21;
   STATE
         count: INTEGER;
   BEGIN
        << reset -> count := 1  >>
      + << NOT reset >> *
              ( << count := (count+1) MOD max>>
              + << countmax :=  count=0 >>
              )
   END;
END.

(************************************************)
(*
* the definition module for the counter modulo-4
*
*)
DEFINITION MODULE counter4;
 EXPORT counter4;

 TYPE
     COUNTER4 = CELL( reset, inc, ev1, ev2, ev3, ev4: BOOLEAN );

 STATIC
  counter4: COUNTER4;

END.
(************************************************)
(*
* the implementation module for the counter modulo-4
*
*)
IMPLEMENTATION MODULE counter4;

 STATIC
  counter4: COUNTER4 =
 STATE
   edge1, edge2: BOOLEAN;
 BEGIN
   << reset -> edge1, edge2 := FALSE, FALSE >>
 + << (NOT reset) AND inc >> * (
         << edge1 := NOT edge2 >>
       + << edge2 := edge1 >>
       )
 + << ev1 := inc AND (NOT reset) AND edge1 AND (NOT edge2) >>
 + << ev2 := inc AND (NOT reset) AND edge1 AND edge2 >>
 + << ev3 := inc AND (NOT reset) AND (NOT edge1) AND edge2 >>
 + << ev4 := inc AND (NOT reset) AND (NOT edge1) AND (NOT edge2) >>
   END;

END.

(************************************************)
(*
* the definition module for the address decoder
*
*)
DEFINITION MODULE addrdec;
 FROM useful IMPORT BoolArray;
 EXPORT addrdec;

 TYPE
  ADDRDEC = CELL( ale, aen: BOOLEAN; addr: BoolArray;
          addrIsTD, addrIsTA, addrIsTC, addrIsTS: BOOLEAN );

 STATIC
  addrdec: ADDRDEC;

END.
(************************************************)
(*
* the implementation module for the address decoder
*
*)
IMPLEMENTATION MODULE addrdec;

  FROM useful IMPORT BAToInt;
  FROM stringIO IMPORT WriteString;
  FROM strings IMPORT BoolToString, IntToString;
  IMPORT con;

STATIC
 addrdec: ADDRDEC =    (* address decoder *)
   STATIC
    trainData, trainAddr, trainCmd, trainStatus : INTEGER =
    300H,       304H,       308H,      30CH;

  BEGIN
   << NOT aen AND ale >>
   * (
    << addrIsTD := BAToInt( addr ) = trainData >>
  + << addrIsTA := BAToInt( addr ) = trainAddr >>
  + << addrIsTC := BAToInt( addr ) = trainCmd >>
  + << addrIsTS := BAToInt( addr ) = trainStatus >>
    )
  END;

END.

(************************************************)
(*
* the definition module for the read/write decoder;
* this module generates the signals informing the controller if
* there is a read or write to one of its registers
*)
DEFINITION MODULE readwrite;
 EXPORT readwrite;

 TYPE
   READWRITE = CELL( addrIsTD, addrIsTA, addrIsTC, addrIsTS,
             iow, ior,
             writeAddr, writeData, writeCmd, writeSts,
             readData, readSts: BOOLEAN );
 STATIC
  readwrite: READWRITE;

END.
(************************************************)
(*
* the implementation module for the read/write decoder
*
*)
IMPLEMENTATION MODULE readwrite;

 FROM strings IMPORT BoolToString;
 FROM stringIO IMPORT WriteString;
 IMPORT con;

 STATIC
  readwrite: READWRITE =    (* i/o controller *)
   BEGIN
        << writeAddr :=  addrIsTA AND (NOT iow) >>
      + << writeData := addrIsTD AND (NOT iow) >>
      + << writeCmd := addrIsTC AND (NOT iow) >>
      + << writeSts := addrIsTS AND (NOT iow) >>
      + << readData := addrIsTD AND (NOT ior) >>
      + << readSts := addrIsTS AND (NOT ior) >>
   END;
```

```
END.

(*************************************************)
(*
* the definition module for some of the control logic;
* this module generates the internal reset signal, and the output
* enable signals for driving the output buffers connected to the
* traindata and the isadata lines
*)
DEFINITION MODULE contrlogic;
 FROM useful IMPORT BoolArray;
 EXPORT contrlogic;

 TYPE
  CONTRLOGIC = CELL( isaReset, readData, readSts, writeSts, dataOC,
                                  statusOC, reset : BOOLEAN );

 STATIC
  contrlogic: CONTRLOGIC;

END.
(*************************************************)
(*
* the implementation module for some of the control logic
*
*)
(* the VHDL code will also have the isa-clock buffering *)
IMPLEMENTATION MODULE contrlogic;
  FROM stringIO IMPORT WriteString;
  FROM strings IMPORT BoolToString;
  IMPORT con;

 STATIC
  contrlogic: CONTRLOGIC =
   BEGIN
         << reset := isaReset OR writeSts >>
       + << dataOC := readData AND (NOT isaReset) >>
       + << statusOC := readSts AND (NOT isaReset) >>
   END;

END.

(*************************************************)
(*
* the definition module for the queue controller
*
*)
DEFINITION MODULE qcontroller;
 FROM useful IMPORT BoolArray;
 EXPORT qcontroller;

 TYPE
   QCONTROLLER = CELL( tbEdge1, tbEdge3, tbEdge4, reset, writeCmd,
            readData, isread, iswrite, Cf0, Rf1, shiftCf, shiftRf,
            shiftWf, forceIdle: BOOLEAN );

 STATIC
  qcontroller: QCONTROLLER;

END.
(*************************************************)
(*
* the implementation module for the queue controller
*
*)
IMPLEMENTATION MODULE qcontroller;      (* queue controller *)
  FROM stringIO IMPORT WriteString;
  FROM strings IMPORT BoolToString, IntToString;
  FROM useful IMPORT BoolToIntFn, BoolFn0, BAToInt;
  IMPORT con;

 STATIC
  qcontroller: QCONTROLLER=
   STATE
```



```
 (* Cf1 = Command FIFO stage 1full;
  * Rf0 = Result-data FIFO stage 0 full;
  * Wf2 = Write-data FIFO stage 2 full *)
Cf1, Rf0, Wf2: BOOLEAN;

STATIC
 (* these are the binary codes of the commands: *)
cmdwrite, cmdread, cmdmreset: INTEGER = 4, 5, 2;

 (* some functions to make the main cell body easier to read:*)

 (* we may only issue a new command - i.e., enable the outputs of
  * the buffers connected to the train-bus - if there is a new
  * command sitting in the top C-FIFO stage and there is room in
  * the bottom R-FIFO stage to record the new result
  * if necessary *)
issuecmd: BoolFn0 = BEGIN Cf1 AND (NOT Rf0) END;
(* advance Command-FIFO: *)
advancefifoc: BoolFn0 = BEGIN Cf0 AND (NOT Cf1) END;
(* advance Result-FIFO (i.e. data-read-FIFO): *)
advancefifor: BoolFn0 = BEGIN Rf0 AND (NOT Rf1) END;
(* response data from the slave device is loaded from the
 * train-bus into the R-FIFO on tbEdge4 if the current command
 * on the train-bus is a read-command, i.e., if a read-command
 * was issued onto the train-bus on the last tbEdge1: *)
availdata: BoolFn0 = BEGIN isread AND tbEdge4 END;

BEGIN
      (* if the C-FIFO is advancing and we are not writing to
       * it, the bottom stage (0) becomes empty *)
   << reset OR ((NOT writeCmd) AND advancefifoc())
     -> Cf0 := FALSE >>

 + << (NOT reset) AND writeCmd -> Cf0 := TRUE >>
 + << shiftCf := Cf0 AND NOT Cf1 >>
     (* if stage 0 is full, stage 1 is empty (this is what the
      * advancefifoc function says) and we are not
      * writing to stage 0, we can advance the FIFO *)
 + << NOT reset AND advancefifoc() AND (NOT writeCmd)
     -> Cf1 := Cf0 >>  (* TRUE *)
     (* if we issue the content of the top C-FIFO stage
      * to the train-bus, and we are not currently advancing
      * the FIFO, the top stage becomes empty: *)
 + << reset OR (NOT advancefifoc() AND issuecmd() AND tbEdge1)
     -> Cf1 := FALSE >>

     (* on tbEdge3, write-data is let out to the train-bus,
      * and since the W-FIFO (write data-FIFO) advances only
      * on tbEdge1, the top stage of this FIFO becomes
      * empty:*)
 + << reset OR tbEdge3 -> Wf2 := FALSE >>
     (* The following condition results from the fact that
      * the W-FIFO advances stage 0 to 1 synchronously with
      * the C-FIFO, but stage 1 to 2 independently, depending
      * on the state of W-stage 2 *)
 + << NOT reset AND shiftWf AND NOT shiftCf -> Wf2 := TRUE >>
     (* Cf1 denotes the same as a supposed variable Wf1;
      * that is, the state of stage 1 of the C-FIFO is the
      * same as the state of stage 1 of the W-FIFO *)
 + << shiftWf := Cf1 AND NOT Wf2 >>

     (* if the R-FIFO (result, or data-read FIFO) advances and
      * and there is not new data to be loaded, stage 0
      * becomes empty *)
 + << reset OR (NOT (tbEdge4 AND isread) AND advancefifor())
    -> Rf0 := FALSE >>
     (* new data is written to the bottom stage, 0, on tbEdge4
      * if the current command, i.e., the one that was issued
      * to the train-bus on the previous tbEdge1, is a read *)
 + << NOT reset AND isread AND tbEdge4 -> Rf0 := TRUE >>
 + << shiftRf := Rf0 AND NOT Rf1 >>
     (* if the current ISA-IO-read cycle is one from the
      * controller DATA register, and the R-FIFO is not
      * currently advancing, the top stage, 1, becomes
      * empty *)
```

```
    + << reset OR (readData AND NOT advancefifor())
       -> Rf1 := FALSE >>
    + << NOT reset AND advancefifor() AND NOT(tbEdge4 AND isread)
       -> Rf1 := Rf0 >>          (* Rf1:= TRUE *)

        (* if the conditions for issuing a new command are not
         * met (see issuecmd-function comment), issue an idle
         * command out on the train-bus *)
    + << reset OR (NOT issuecmd()) -> forceIdle := TRUE >>
    + << (NOT reset) AND issuecmd() -> forceIdle := FALSE >>
   END;
END.

(*************************************************)
(*
* the definition module for all the FIFO buffers;
* there are many different types of buffer cells because
* it was easier this way to deal with the many different bit-widths
* and load/output-enable commands required; also for debugging
* purposes, I needed to be able to print out different things from
* inside different buffer-cells, which would not have been possible
* if I would have defined a single general-purpose buffer. There are
* a few general-purpose buffers, which I use in different
* instantiations, so those have a generic name
*)
DEFINITION MODULE buffers;
 FROM useful IMPORT BoolArray;
 EXPORT bufld, bufcsld, buf0r, buf1r, buf0s, buf1s, buf0c, buf3d,
        buf2c ;
 TYPE
 (* 8-bit, ld (load) input signal *)
  Buffer8ld = CELL( ld: BOOLEAN; in, out: BoolArray);
 (* 8-bit, cs (chip-select) and ld signal *)
  Buffer8csld= CELL( cs, ld: BOOLEAN; in, out: BoolArray);
 (* 8-bit, oe (output-enable) and ld signal *)
  Buffer8oeld = CELL( ld, oe: BOOLEAN; in, out: BoolArray);
 (* 8-bit, cs and ld and oe signal *)
  Buffer8csldoe = CELL( cs, ld, oe: BOOLEAN; in, out: BoolArray);
 (* 8-bit, oe signal *)
  Buffer8oe = CELL( oe: BOOLEAN; in, out: BoolArray);
 (* 2-bit, loads on every clock edge *)
  Buffer2 = CELL( in0, in1: BOOLEAN; out: BoolArray );
 (* 4-bit, ld input signal *)
  Buffer4ld = CELL( ld, in0, in1, in2, in3: BOOLEAN; out: BoolArray);
 (* 4-bit, special buffer with input cs and outputs force, isread and
 * iswrite to be described further below *)
 Buffer4spec = CELL( cs, force, iswrite, isread: BOOLEAN;
                                    in, out: BoolArray);

 STATIC
  bufld: Buffer8ld;          (* generic *)
  bufcsld: Buffer8csld;      (* generic *)
  buf3d : Buffer8oeld;       (* Data-write FIFO, stage 3 *)
  buf0r: Buffer8csld;        (* data-Result FIFO, stage 0 *)
  buf1r: Buffer8csldoe;      (* data-Result FIFO, stage 1 *)
  buf0s: Buffer2;            (* Status(-read) FIFO, stage 0 *)
  buf1s: Buffer8oe;          (* Status(-read) FIFO, stage 1 *)
  buf0c: Buffer4ld;          (* Command(-write) FIFO, stage 0 *)
  buf2c: Buffer4spec;        (* Command(-write) FIFO, stage 2 *)
   (* stages missing from these declarations are instantiations of
    * generic buffers *)

END.
(*************************************************)
(*
* the implementation module for the FIFO buffers
*
*)
IMPLEMENTATION MODULE buffers;
 FROM useful IMPORT BoolFn0, IntToBA, BAToInt;
 FROM strings IMPORT BoolToString, IntToString;
 FROM stringIO IMPORT WriteString;
 IMPORT con;

 (* as mentioned, the combination digit-letter gives the stage
 * and the type of the FIFO for which it is intended *)
STATIC

 buf0r: Buffer8csld =
  BEGIN
       << cs AND ld -> out := in >>
      * WriteString( BEGIN con.cat2(
              "controller loads tb-data: ",
              IntToString(BAToInt(in), 16)) END )
  END;

 buf0s: Buffer2 =
  BEGIN
      << out(0), out(1) := in0, in1  >>
                              (* don't care about out(7..2) *)
  END;

 buf2c: Buffer4spec =
   STATE
       cmdIdle: INTEGER = 3;
    STATIC
      (* the least significant bit of the command, in(0),
       * indicates a read or a write command *)
       iswritefn: BoolFn0 =
           BEGIN NOT force AND NOT in(0) ) = cmdWrite) END;
       isreadfn:  BoolFn0 =
           BEGIN NOT force AND in(0) ) = cmdRead) END;

  BEGIN        (* forces idle when force is TRUE *)
       << cs >> *
              ( << NOT force -> out := in >>
              + << force -> out := IntToBA( cmdIdle, 4) >>
              + << iswrite := iswritefn() >>
              + << isread := isreadfn() >>
              )
  END;

 buf1r: Buffer8csldoe =
   STATE
       temp: BoolArray(8);
  BEGIN
       (* << cs >> * *)
       (
         << ld AND NOT oe -> temp := in >>
       + << oe AND NOT ld -> out := temp >>
         * WriteString( BEGIN con.cat2(
              "controller driving isa-data lines with data= ",
              IntToString(BAToInt(temp), 16)) END )
       + << oe AND ld -> out := in >>
         * WriteString( BEGIN con.cat2(
              "controller driving isa-data lines with data= ",
              IntToString(BAToInt(in), 16)) END )
       )
  END;

 buf1s: Buffer8oe =
  BEGIN
       << oe -> out := in >>
       * WriteString( BEGIN con.cat2(
          "controller driving isa-data lines with status= ",
           IntToString(BAToInt(in), 16)) END )
  END;

 buf0c: Buffer4ld =
       STATIC cmdidle : INTEGER = 3;
  BEGIN
       << ld -> out(0), out(1), out(2), out(3) :=
                in0,    in1,    in2,    in3    >>

  END;

 buf3d: Buffer8oeld =
  BEGIN
       << ld AND oe -> out := in >>
  END;
```

```
  bufld: Buffer8ld =                (* for 0a and 0d *)
   BEGIN
         << ld -> out := in >>
   END;

  bufcsld: Buffer8csld =
   BEGIN
         << cs AND ld -> out := in >>
   END;
END.


(*************************************************)
(*
* the top definition module of the bus controller (bc)
* this is where the tope cell, ''whole'', is defined;
* here I also have the types for ISA_hard, ISA-soft
* and the trainbus (record-types)
*)
DEFINITION MODULE bc;
  FROM useful IMPORT BoolArray;
  EXPORT bcCell, whole, ISAhard, ISAsoft, TB, ISAaddr, dataT,
         cmdT, TBaddr;


  TYPE
    (* Software description of the ISA bus. *)
   ISAsoft = RECORD
                op: BOOLEAN;
                a: INTEGER;
                d: INTEGER;
                busy: BOOLEAN
              END;

    (* Hardware description of the ISA bus. *)
   ISAhard = RECORD
          a: BoolArray(10);
          d: BoolArray(8);
          aen, ale, ior, iow, rst: BOOLEAN
        END;
    (* description of the trainbus *)
   TB = RECORD
         a: BoolArray(8);
         d: BoolArray(8);
         c: BoolArray(4)
       END;

  T = CELL(STATIC iofile: STRING; STATIC count: INTEGER);
  BCcell = CELL( isa: ISAhard; tb: TB;
                 tbEdge1, tbEdge2, tbEdge3, tbEdge4: BOOLEAN );
  ISAaddr = INTEGER;
  dataT = INTEGER;
  TBaddr = INTEGER;
  cmdT = INTEGER;


(*************************************************)
(*
* the top implementation module of the bus controller (bc)
* this is where the tope cell, ''whole'', is implemented
*)
IMPLEMENTATION MODULE bc;
  FROM addrdec IMPORT addrdec;
  FROM readwrite IMPORT readwrite;
  FROM counter21 IMPORT counter21;
  FROM counter4 IMPORT counter4;
  FROM qcontroller IMPORT qcontroller;
  FROM contrlogic IMPORT contrlogic;
  FROM buffers IMPORT bufld, bufcsld, buf0r, buf1r, buf0s, buf1s,
                                        buf0c, buf3d, buf2c ;
  FROM isa IMPORT isaCell;
  FROM device IMPORT devCell;
  FROM driver IMPORT driver;
  FROM strings IMPORT IntToString, BoolToString;
  FROM stringIO IMPORT WriteString;
  FROM useful IMPORT IntToBA;
  FROM useful IMPORT BAToInt;
  IMPORT con;

STATIC
bcCell: BCcell =        (* the model for the bus controller *)

  STATE  (* internal signals *)
        shiftRf, shiftCf, shiftWf: BOOLEAN= FALSE, FALSE, FALSE;
        forceIdle, iswrite, isread: BOOLEAN;
        Cf0, Rf1: BOOLEAN = FALSE, FALSE;
        reset: BOOLEAN;
        writeData, writeAddr, writeCmd, writeSts: BOOLEAN;
        readData, readSts: BOOLEAN = FALSE, FALSE;
        addrIsTD, addrIsTA, addrIsTC, addrIsTS: BOOLEAN =
          FALSE,    FALSE,    FALSE,    FALSE;
        interCmd0, interCmd1: BoolArray(4);
        interDataR : BoolArray(8);
        interDataW0, interDataW1, interDataW2: BoolArray(8);
        interAddr0, interAddr1 : BoolArray(8);
        interStatus: BoolArray(8);
        dataOC, statusOC: BOOLEAN; (* active high !! *)
        q: BoolArray(5);          (* don't actually need it *)
        countmax: BOOLEAN;
        true, false: BOOLEAN= TRUE, FALSE;


  BEGIN

      counter21( reset, q, countmax )
    + counter4( reset, countmax, tbEdge1, tbEdge2, tbEdge3, tbEdge4 )
    + addrdec( isa.ale, isa.aen, isa.a,
        addrIsTD, addrIsTA, addrIsTC, addrIsTS )
    + readwrite(  addrIsTD, addrIsTA, addrIsTC, addrIsTS,
        isa.iow, isa.ior,
        writeAddr, writeData, writeCmd, writeSts,
        readData, readSts )

    + qcontroller( tbEdge1, tbEdge3, tbEdge4, reset, writeCmd,
        readData, isread, iswrite, Cf0, Rf1, shiftCf, shiftRf,
        shiftWf, forceIdle)

    + contrlogic( isa.rst, readData, readSts, writeSts, dataOC,
                          statusOC,  reset )
(* all buffers are clocked on the isa clock; this is transparent
 * in st. The isa-clock is buffered by contrlogic, but in st we
 * did not need it  *)

(* there are a few buffer cells that are instantiated several times,
like bufld - buffer with ld signal - and others that are *dedicated*
like buf0r etc. The combination digit-letter gives the stage and the
name of the fifo: stages 0 through 2 on fifos: a=address, d=data,
c=command, r=result(data)  *)

    + buf0r( isread, tbEdge4, tb.d, interDataR )            (* 0r *)
    + buf1r( shiftRf, shiftRf, dataOC, interDataR, isa.d )  (* 1r *)
    + buf0s( Cf0, Rf1, interStatus )                        (* 0s *)
    + buf1s( statusOC, interStatus, isa.d )                 (* 1s *)
    + bufld( writeAddr, isa.d, interAddr0 )                 (* 0a *)
    + bufld( shiftCf, interAddr0, interAddr1 )              (* 1a *)
    + bufld( tbEdge1, interAddr1, tb.a )                    (* 2a *)
    + bufld( writeData, isa.d, interDataW0 )                (* 0d *)
    + bufld( shiftCf, interDataW0, interDataW1 )            (* 1d *)
    + bufld( shiftWf, interDataW1, interDataW2 )            (* 2d *)
    + buf3d( tbEdge3, iswrite, interDataW2, tb.d )          (* 3d *)
    + buf0c( writeCmd, isa.d(0), isa.d(1), isa.d(2),
                          isa.d(3), interCmd0 )             (* 0c *)
    + bufld( shiftCf, interCmd0, interCmd1 )                (* 1c *)
    + buf2c( tbEdge1, forceIdle, iswrite, isread, interCmd1, tb.c  )
    )
  END;

  whole: T =
     STATIC
```

```
      cmdIdle: INTEGER = 3;
    STATE
      devaddr: INTEGER = 32H;
      isas: ISAsoft;
      isah: ISAhard;
      tb: TB;
      init, initClocks: BOOLEAN = TRUE, TRUE;
      tbEdge1, tbEdge2, tbEdge3, tbEdge4: BOOLEAN =
        FALSE, FALSE,   FALSE,   FALSE;
      p: BOOLEAN = FALSE;
  BEGIN
    << initClocks ->
      initClocks, tb.c,           isah.ior, isah.iow, isah.aen :=
      FALSE, IntToBA( cmdIdle, 4 ), TRUE,     TRUE,     TRUE
    >>
  + << NOT initClocks >>
  * (   (* "drives" the whole system *)
        driver(isas, init, iofile, count )

      + << NOT init >>
      * (
         isaCell( isas, isah )
       + bcCell( isah, tb, tbEdge1, tbEdge2, tbEdge3, tbEdge4 )
       + devCell( tbEdge2, tbEdge3, tbEdge4, tb, devaddr )
        )
    )
  END;
END.

(************************************************)
(*
* the definition module for an ISA simulator
*
*)
DEFINITION MODULE isa;
  FROM bc IMPORT ISAsoft, ISAhard, TB;
  EXPORT isaCell;
TYPE
  ISACell = CELL( sw: ISAsoft; hw: ISAhard );

STATIC
  isaCell: ISACell;
END.
(************************************************)
(*
* the implementation module for the ISA simulator
* this implements the standard ISA 8-bit IO cycle
*)
IMPLEMENTATION MODULE isa;
  FROM strings IMPORT IntToString;
  FROM stringIO IMPORT WriteString;
  FROM useful IMPORT IntToBA;
  FROM useful IMPORT BAToInt;
  FROM driver IMPORT OpToString;
  IMPORT con;

STATIC
  inB, outB: BOOLEAN = FALSE, TRUE;
  idle, addrSet, dataSet: INTEGER =
   0,    1,       2,
  waiting1, waiting2, waiting3, done: INTEGER =
   3,       4,        5,        6;
  TrainBase: INTEGER = 300H;
  TrainAddr,   TrainData,   TrainCmd,    TrainStatus: INTEGER =
  TrainBase+4, TrainBase+0, TrainBase+8, TrainBase+0CH;
  isaCell: ISACell =
  STATE
    state: INTEGER;
    reset: BOOLEAN = TRUE;

  BEGIN
    << reset -> hw.rst, reset := TRUE, FALSE >>
       || << NOT reset -> hw.rst := FALSE >> * (
     (* ISA initiates a transfer *)
    <<  sw.busy AND (state = idle)
      -> state, hw.a, hw.ale, hw.aen, hw.sbhe :=
         addrSet, IntToBA( sw.a, 10 ), TRUE, FALSE, TRUE >>
  + << state=addrSet -> state, hw.ale := dataSet, FALSE >>
    *(
      << sw.op = outB -> hw.iow, hw.d :=
                         FALSE, IntToBA( sw.d, 8 ) >>
    + << sw.op = inB -> hw.ior := FALSE >>
    )

      + << state=dataSet >>
        * (
        << sw.op = outB -> state := waiting3 >>
      + << sw.op = inB -> state := waiting1 >>
    )
      + << state=waiting1 -> state := waiting2 >>
      + << state=waiting2 -> state := waiting3 >>
      + << state=waiting3 ->
           state := done >>
      * (  (* ISA gets data it was waiting for *)
        << sw.op = inB -> sw.d, hw.ior := BAToInt(hw.d) , TRUE >>
      + << sw.op = outB -> hw.iow := TRUE >>
    )
      + << state=done ->
         state, sw.busy, hw.aen, hw.sbhe :=
         idle, FALSE,   TRUE,   FALSE >>
      )
  END;
END.


(************************************************)
(*
* Definition module for the cells used to model the behaviour of
*   performing inbyte and outbyte operations on an ISA bus.
*)
DEFINITION MODULE byteIO;
  FROM bc IMPORT ISAaddr, dataT, ISAsoft;
  EXPORT inByte, outByte;

TYPE
  byteIOcell = CELL( globalnew, localnew: BOOLEAN; data,
                     addr:INTEGER; isa: ISAsoft );
STATIC
  inByte : byteIOcell;
  outByte : byteIOcell;
END.

(************************************************)
(*
* Implementation module for the cells used to model the behaviour
* of performing inbyte and outbyte operations on an ISA bus.
*)
IMPLEMENTATION MODULE byteIO;
  FROM stringIO IMPORT WriteString;
  FROM strings IMPORT IntToString, BoolToString;
  IMPORT con;
STATIC
  inB, outB: BOOLEAN = FALSE, TRUE;

(*************************************************************)
(* outByte cell                                              *)
(*************************************************************)
(* When new is true, this cell puts commands onto isa,
 * waits for them to
 * be handled, and shows that we're done by clearing new. *)
  outByte : byteIOcell =
    STATE
      (* A flag to tell us if we're waiting for bus to finish. *)
      waiting : BOOLEAN = FALSE;
  BEGIN
    (* Don't do anything until we're signalled.
       We're signalled by two variables, globalnew & localnew, so
       that we can actually control the enabling of this cell from
       different levels *)
```

```
    << globalnew AND localnew >>
  * (
        (* If the ISA bus isn't busy, and we have a pending
         * request for an  inbyte, put the request on the bus. *)

        << NOT isa.busy AND NOT waiting ->
             waiting, isa.busy, isa.op, isa.a, isa.d :=
             TRUE,    TRUE,     outB,    addr,  data >>
        (* If the bus is no longer busy, and we were waiting, the
           data has been accepted by the target address. *)
    || << NOT isa.busy AND waiting
       -> globalnew, localnew, waiting := FALSE, FALSE, FALSE >>

  )

  END;

(*********************************************************)
(* inByte cell                                           *)
(*********************************************************)
(* When new is true, this cell asks the isa bus for data,
 *  waits for the request to be filled, and shows that we're done
 *  by clearing new. *)
  inByte : byteIOcell =
    STATE
      (* A flag to tell us if we're waiting for a response. *)
      waiting : BOOLEAN = FALSE;
  BEGIN
    (* Don't do anything until we're signalled.
       We're signalled by two variables, globalnew & localnew, so
       that we can actually control the enabling of this cell from
       different levels *)

    << globalnew AND localnew >>
  * (
 (* If the ISA bus isn't busy, and we have a pending request for an
  * inbyte, put the request on the bus. *)
    << NOT isa.busy AND NOT waiting ->
       waiting, isa.busy, isa.op, isa.a := TRUE, TRUE, inB, addr >>

 (* If the bus is no longer busy, and we were waiting, the target
  * address has supplied the data. *)
 || << NOT isa.busy AND waiting ->
       globalnew, localnew, data, waiting :=
         FALSE,   FALSE,    isa.d, FALSE >>
   )
  END;
END.


(*************************************************)
(*
* the definition module for the ''driver'' (a simulator of the
* software controlling the train set):
*)
DEFINITION MODULE driver;
  FROM bc IMPORT ISAsoft;
  EXPORT driver, CmdToString, AddrToString, OpToString;

TYPE
  IntToBool = FUNCTION(a: INTEGER): BOOLEAN;
  Status = RECORD
             CmdFifoFull: BOOLEAN;
             ResultFifoFull: BOOLEAN
           END;
  IntArray(n: INTEGER) = ARRAY(i: INDEX(n)): INTEGER;

  ReadCell = CELL( STATIC iofile: STRING; addr, data: IntArray;
                   STATIC count: INTEGER; done: BOOLEAN );
  ResetCell = CELL(init: BOOLEAN; isa: ISAsoft; doByteOp: BOOLEAN);
  StatusCell = CELL(state: INTEGER; s: Status; isa: ISAsoft;
                    doByteOp: BOOLEAN);
  IssueCell = CELL(state: INTEGER; isa: ISAsoft; addr, data:
               IntArray; doByteOp: BOOLEAN; done: INTEGER);
  ReceiveCell = CELL(state: INTEGER; isa: ISAsoft;
                     doByteOp: BOOLEAN);
  SendCell = CELL(isa: ISAsoft; addr, data: IntArray;
             doByteOp: BOOLEAN; index: INTEGER);
  RetireCell = CELL(isa: ISAsoft; doByteOp: BOOLEAN);
  DriverCell = CELL(isa: ISAsoft; init: BOOLEAN;
             STATIC iofile: STRING; STATIC count: INTEGER );
  S2S = FUNCTION(s: INTEGER ): STRING;
  C2S = FUNCTION(c: INTEGER): STRING;
  I2S = FUNCTION(i: INTEGER): STRING;
  B2S = FUNCTION(b: BOOLEAN): STRING;

STATIC
  read: ReadCell;
  reset: ResetCell;
  checkStatus: StatusCell;
  issue: IssueCell;
  receive: ReceiveCell;
  send: SendCell;
  retire: RetireCell;
  driver: DriverCell;
  CmdToString: C2S;
  AddrToString: I2S;
  OpToString: B2S;
END.
(*************************************************)
(*
* the implementation module for the ''driver'' (a simulator of the
* software controlling the train set):
*)
IMPLEMENTATION MODULE driver;
  FROM random IMPORT MakeRandom;
  FROM useful IMPORT IntToInt, bit;
  FROM byteIO IMPORT inByte, outByte;
  FROM bc IMPORT ISAaddr, dataT, TBaddr, cmdT;
  FROM IO IMPORT ReadInt, ReadBool;
  IMPORT con;

STATIC
  TrainBase: ISAaddr = 300H;
  TrainAddr,   TrainData,   TrainCmd,    TrainStatus: ISAaddr =
  TrainBase+4, TrainBase+0, TrainBase+8, TrainBase+0CH;
  inB, outB: BOOLEAN = FALSE, TRUE;
  cmdIdle, cmdWrite, cmdRead, cmdStart, cmdStop: cmdT =
  3,       4,        5,       6,        7;
  cmdGreset, cmdDreset, cmdMreset: cmdT =
    0,         1,         2;

  (* commandfifo full *)
  STATIC cff: IntToBool =  BEGIN bit( a, 0 ) END;
  (* resultfifo full *)
  STATIC rff: IntToBool =  BEGIN bit( a, 1 ) END;

  STATIC
 reset: ResetCell =
     STATE doOutByte: BOOLEAN=FALSE;
           isaAddr, isaData: INTEGER = TrainStatus, 0;
    BEGIN
     << init -> doOutByte, doByteOp, init := TRUE, TRUE, FALSE >>
      || outByte(doByteOp, doOutByte, isaData, isaAddr, isa )
     END;

 read: ReadCell =
       STATE cnt, a, d: INTEGER = 0, 0, 0;
    BEGIN
        ( ReadInt( a, iofile, "isaAddr")
        * ReadInt( d, iofile, "isaData")
        * << cnt := cnt+1 >>
        )
        + << NOT done >> * (
              << cnt > 0  -> addr(cnt-1), data(cnt-1) := a, d >>
            + << cnt = count -> done := TRUE >>
             )
    END;

 checkStatus: StatusCell =
```

```
    STATE
      doInByte: BOOLEAN;
      step: INTEGER = 0;
      isaAddr, isaData: INTEGER;
  BEGIN
       << (step = 0)  AND (NOT doByteOp)->
          step, doInByte, doByteOp, isaAddr :=
             1, TRUE,       TRUE,  TrainStatus
       >>
    (* next step, inbyte is done, the data is in isaData *)
      || << (step = 1) AND (NOT doByteOp) ->
          s.CmdFifoFull, s.ResultFifoFull, step, state :=
          cff(isaData),  rff(isaData), 0, 2
       >>
    (* state set to 2: "next" for send, "start" for retire *)
   || inByte(doByteOp, doInByte, isaData, isaAddr, isa )
   END;

issue: IssueCell =
    STATE doOutByte: BOOLEAN=FALSE;
          step: INTEGER = 0;
          isaAddr, isaData: INTEGER;
  BEGIN
  (* the order in which data and address are sent doesn't
   * matter *)
   << step = 0 -> step := 1 >>
       * WriteString( BEGIN con.cat2(
          "issuecell starting new command with index =",
          IntToString( done, 10 ))END)

    || << (step = 1) AND (NOT doByteOp) -> (* output address *)
       step, doByteOp, doOutByte, isaAddr, isaData :=
       2,    TRUE,      TRUE, addr(3*done), data(3*done)
    >>

    || << (step = 2) AND (NOT doByteOp) -> (* output data *)
       step, doByteOp, doOutByte, isaAddr, isaData :=
       3,    TRUE,      TRUE, addr(3*done+1), data(3*done+1)
    >>

    || << (step = 3) AND (NOT doByteOp) -> (* output command *)
       step, doByteOp, doOutByte, isaAddr, isaData  :=
       4,    TRUE,      TRUE, addr(3*done+2), data(3*done+2)
    >>

    || << step = 4 >> * (
        << NOT doByteOp ->
         done, step, state := done+1, 0, 1 >>
      )
   || outByte(doByteOp, doOutByte, isaData, isaAddr, isa )
  END;


receive: ReceiveCell =
   STATE doInByte: BOOLEAN=FALSE;
       step: INTEGER = 0;
       isaAddr, isaData: INTEGER;
   BEGIN
     << (step = 0)  AND (NOT doByteOp) ->
       step, doInByte, doByteOp, isaAddr :=
       1,    TRUE,      TRUE, TrainData
     >>
 ||  << (step = 1) AND (NOT doByteOp)
      -> step, state := 0, 1 >> (* state := received *)
 || inByte(doByteOp, doInByte, isaData, isaAddr, isa )

   END;

send: SendCell =
 STATIC
   start, issued, next: INTEGER = 0, 1, 2;
 STATE
   state: INTEGER = start;
   status: Status;
 BEGIN
      << state = start >>
       * issue(state, isa, addr, data, doByteOp, index )
   || << state = issued >> * (
       checkStatus(state, status, isa, doByteOp )
       )
   || << state = next >>
       * ( << status.CmdFifoFull >>
           * checkStatus(state, status, isa, doByteOp )
       + << NOT status.CmdFifoFull ->
           state := start >>
       )
 END;

retire: RetireCell =
 STATIC    (* start=2 so that checkstatus always returns 2 *)
   start, newdata, received: INTEGER = 2, 0, 1;
 STATE
   state: INTEGER = start;
   status: Status;
 BEGIN
      << state = start >>
       * ( << NOT status.ResultFifoFull >>
           * checkStatus(state, status, isa, doByteOp )
       + << status.ResultFifoFull->
           state := newdata >>
       )
   || << state = newdata >>
       * receive(state, isa, doByteOp)
   || << state = received >> * (
        checkStatus(state, status, isa, doByteOp )
       )
 END;

driver : DriverCell =
 STATE
   status: Status;
   doByteOp, done: BOOLEAN=FALSE, FALSE;
   step: INTEGER = 0;
   addr, data: IntArray(50);
 BEGIN

   reset(init, isa, doByteOp)
   + << NOT init >>
   *(
      << step = 0 >>
       * read( iofile, addr, data, count, done )
   || << done >>
      * (    << step < (count DIV 3) >>
       * send(isa, addr, data, doByteOp, step )
        || retire( isa, doByteOp )
        )
    )
 END;
END.
\end{verbatim}
```

Appendix B

ST cell instantiation diagrams

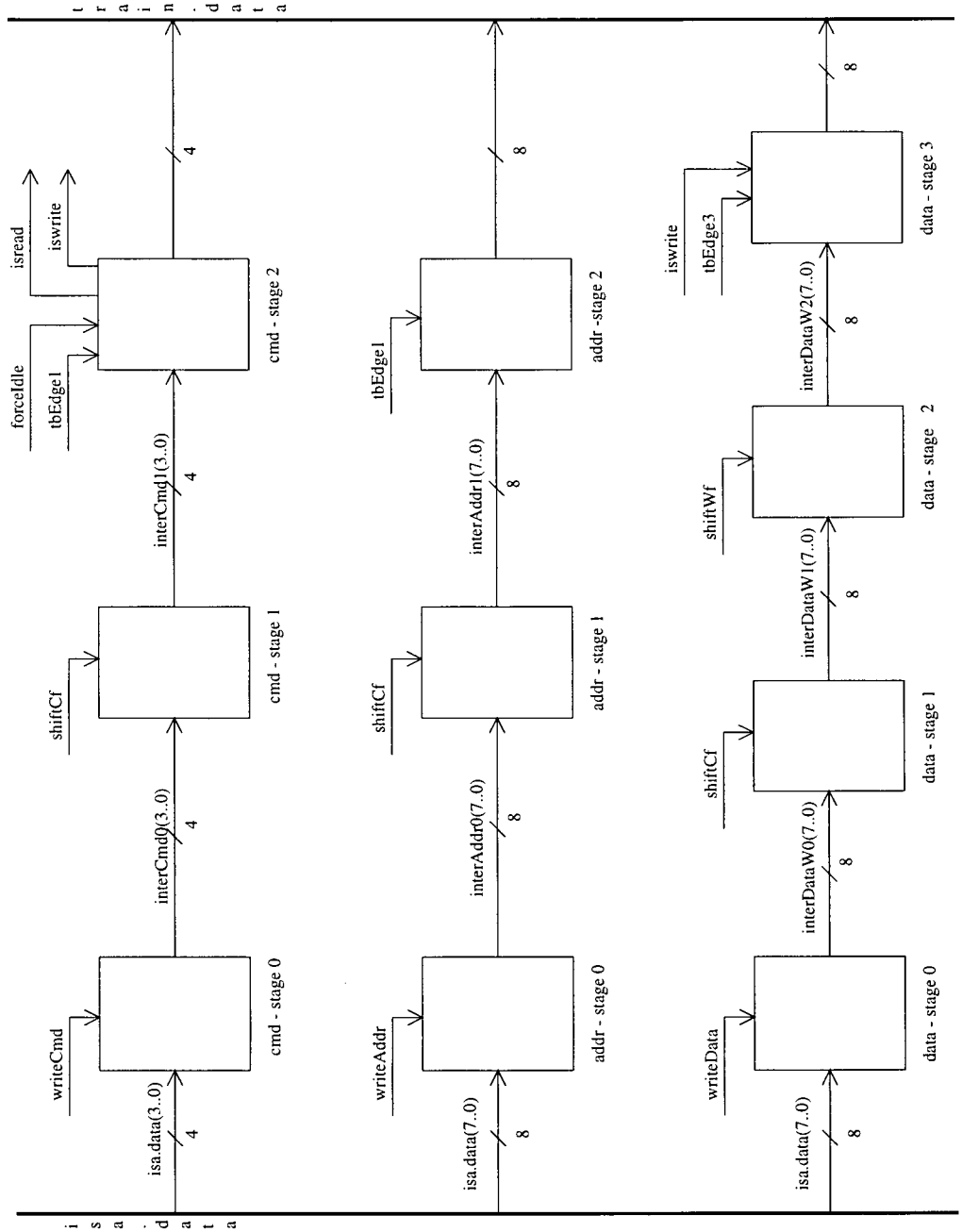

Figure B.1: The "out-FIFOs": Cmd, Addr and Data-Write

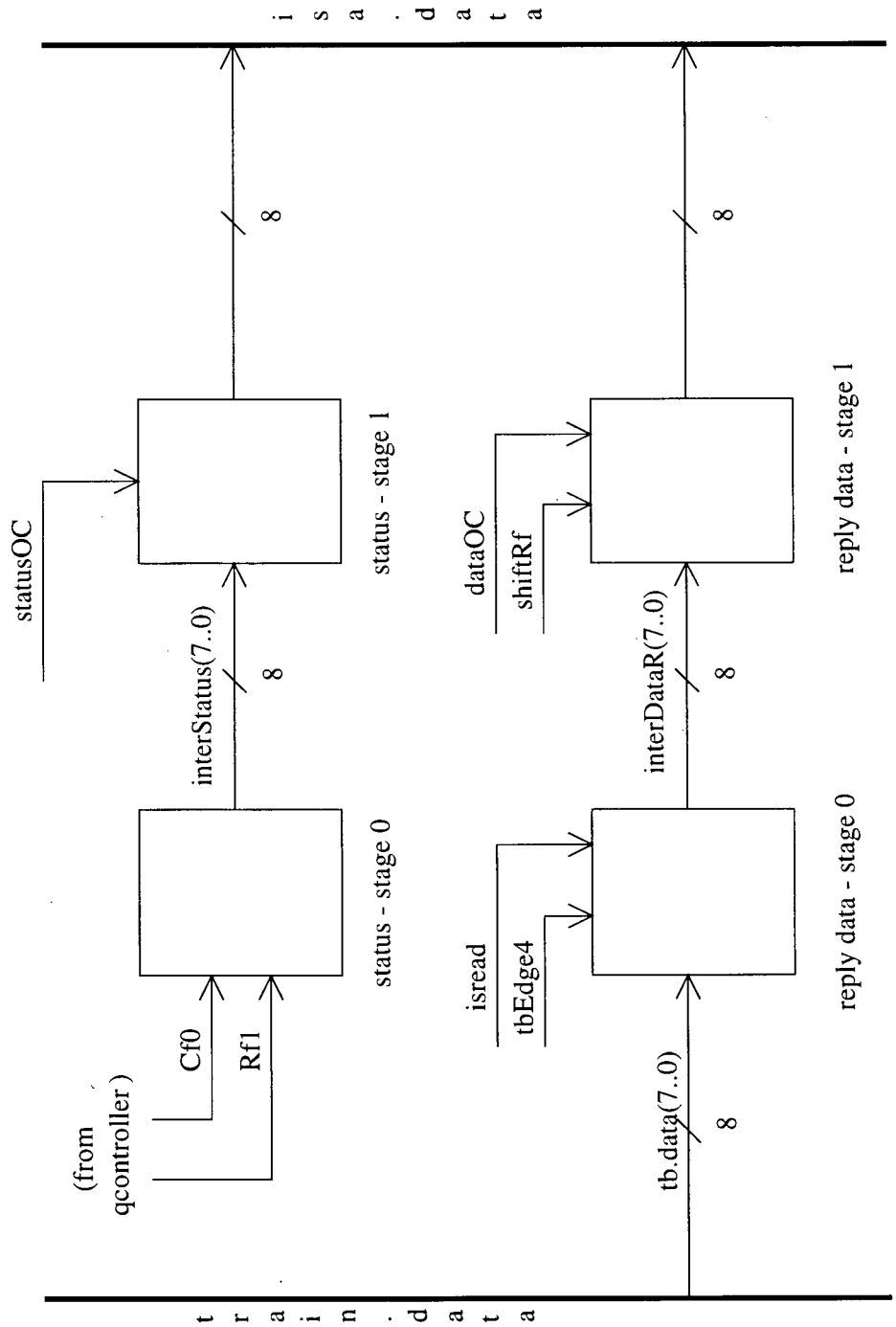

Figure B.2: The "in-FIFOs": Status and Data-Read

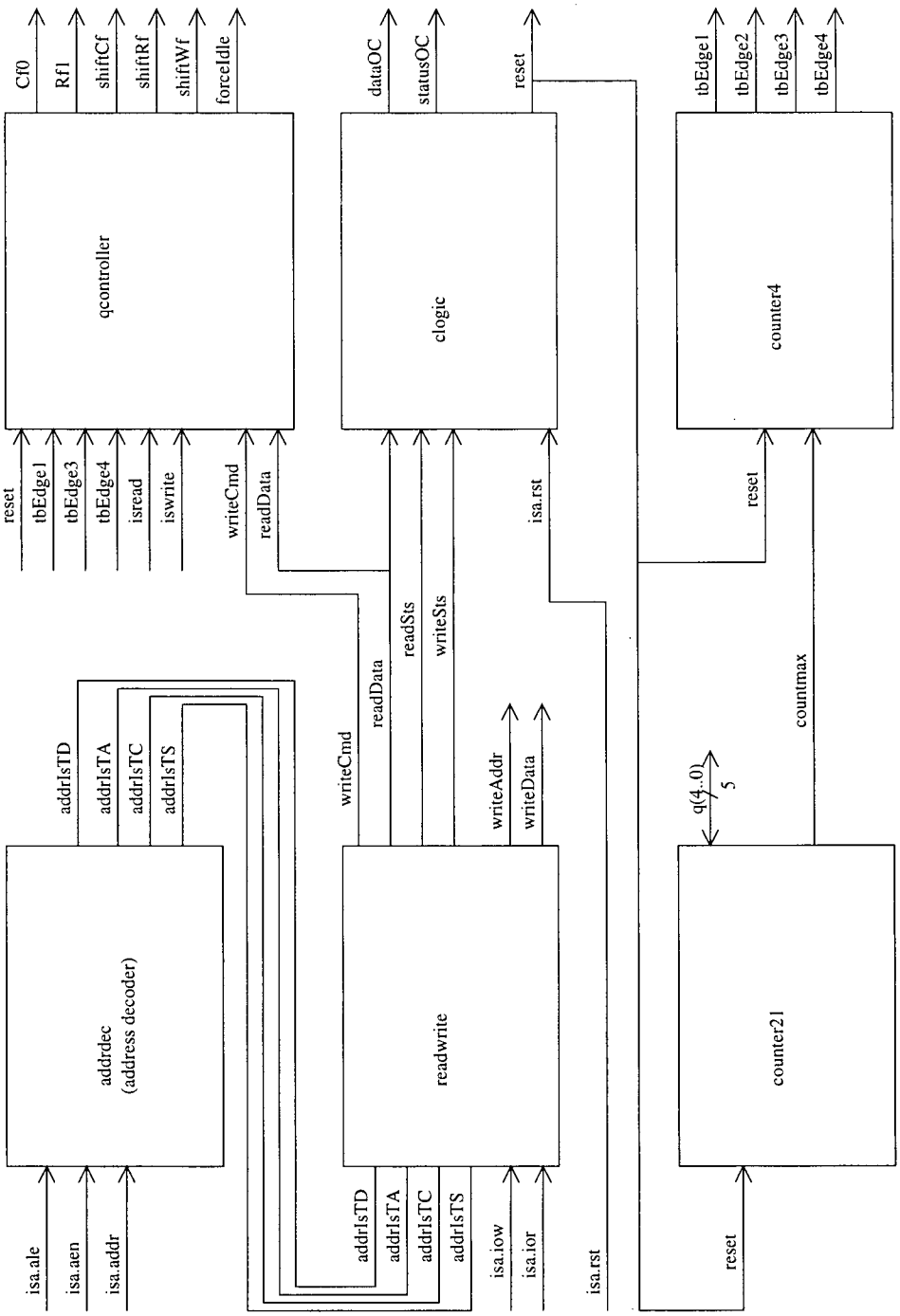

Figure B.3: The Control Logic Cells

Appendix C

VHDL code

```
----------------------------------------
-- counter mod 21
----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
library SYNOPSYS;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity counter21 is
    port( clk, reset: in std_logic;
          count21: out std_logic );
end counter21;

architecture rtl of counter21 is
        signal q: std_logic_vector(4 downto 0);
begin
        process(clk, reset)
        begin
          if (reset='1') then
                count21 <= '0';
                q <= "00000";
          elsif (clk'event and clk='1') then
                if (q="10100") then
                        count21 <= '1';
                        q <= "00000";
                else
                        count21 <= '0';
                        q <= q+1;
                end if;
          end if;
        end process;
end rtl;

----------------------------------------
-- counter mod 4
----------------------------------------
-- ev1 through ev4 are all 1 clk period wide
library IEEE;
library SYNOPSYS;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_1164.all;

entity counter4 is
    port( clk, reset, inc: in std_logic;
          ev1, ev3, ev4: out std_logic;
          bclk1, bclk2: out std_logic);
end counter4;

architecture rtl of counter4 is
  signal clk1, clk2: std_logic;
begin
  process( clk, reset )
  begin
    if (reset = '1') then
     (clk1, clk2, ev1, ev3, ev4) <= std_logic_vector'("00000");
    else                    --(reset = '0')
      if (clk'event and clk = '1') then
         if( inc = '1') then
           clk1 <= not clk2;
           clk2 <= clk1;
           bclk1 <= clk1;
           bclk2 <= clk2;
         end if;
         ev1 <= inc and not reset and not clk1 and not clk2;
         --ev2 <= inc and not reset and   clk1 and not clk2;
         ev3 <= inc and not reset and     clk1 and     clk2;
         ev4 <= inc and not reset and not clk1 and     clk2;
      end if;
    end if;
  end process;
end rtl;
```



```
----------------------------------------
-- address decoder
----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

library SYNOPSYS;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity addrdec is
  port( reset, ale, aen: in std_logic;
        addr: in std_logic_vector( 9 downto 0);
        seldata, seladdr, selcmd, selsts: out std_logic );
end addrdec;

architecture rtl of addrdec is
begin
  process( reset, ale )
    subtype address is std_logic_vector(9 downto 0);
    constant trainData: address := "1100000000";       --X"300"
    constant trainAddr: address := "1100000100";       --X"304"
    constant trainCmd:  address := "1100001000";       --X"308"
    constant trainSts:  address := "1100001100";       --X"30C"
  begin
    if (reset='1') then
      (seldata, seladdr, selcmd, selsts
                                    <=std_logic_vector'("0000");
    else                    -- (reset ='0')
     if (aen='0') then
      if (ale='0') then
        case addr is
          when trainData =>
            (seldata, seladdr, selcmd, selsts)
                                    <= std_logic_vector'("1000");
          when trainAddr =>
            (seldata, seladdr, selcmd, selsts)
                                    <= std_logic_vector'("0100");
          when trainCmd =>
            (seldata, seladdr, selcmd, selsts)
                                    <= std_logic_vector'("0010");
          when trainSts =>
            (seldata, seladdr, selcmd, selsts)
                                    <= std_logic_vector'("0001");
          when others =>
            (seldata, seladdr, selcmd, selsts)
                                    <= std_logic_vector'("0000");
        end case;
      end if;    --ale
     end if;     --aen
    end if;      --reset
  end process;
end rtl;

----------------------------------------
-- readwrite
----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity readwrite is
  port( reset, seldata, seladdr, selcmd, selsts,
                                        iow, ior: in std_logic;
        writeaddr, writedata, writecmd, writests: out std_logic;
        readdata, readsts: out std_logic );
end readwrite;

architecture rtl of readwrite is
begin
  process( reset, seldata, seladdr, selcmd, selsts, iow, ior )
  begin
     writeaddr <= not reset and seladdr and not iow;
     writedata <= not reset and seldata and not iow;
```

```
      writecmd  <= not reset and selcmd and not iow;
      writests  <= not reset and selsts and not iow;
      readdata <= not reset and seldata and not ior;
      readsts <= not reset and selsts and not ior;
  end process;
end rtl;

----------------------------------------
-- contrlogic
----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity contrlogic is
  port( isaclk, isareset, readdata, readsts,
                              writests: in std_logic;
        clock, reset, dataoe, statusoe: out std_logic );
end contrlogic;

architecture rtl of contrlogic is
begin
  process( isaclk, isareset, readdata, readsts, writests )
  begin
    clock <= isaclk;
    reset <= isareset or writests;
    dataoe <= readdata and not isareset;
    statusoe <= readsts and not isareset;
  end process;
end rtl;

----------------------------------------
-- queue controller
----------------------------------------
-- the version without using functions

library IEEE;
use IEEE.std_logic_1164.all;

entity qcontroller is
  port( reset, clk, ev1, ev3, ev4: in std_logic;
        writecmd, readdata: in std_logic;
        isread: in std_logic;
        shiftrf, forceidle: out std_logic;
        cf0, cf1, rf0, rf1, wf2,
                          shiftcf, shiftwf: inout std_logic );
end qcontroller;

architecture rtl of qcontroller is

-- I initially used functions; after I saw however how differently
-- different syntaxes are synthesized, I decided to write
-- everything as low-level as possible.
--
--    function issuecmd( cf1, rf0: std_logic) return boolean is
--    begin
--      return (cf1='1' and rf0='0');
--    end;
--
--    function advancefifoc( cf0, cf1: std_logic) return boolean is
--    begin
--      return (cf0='1' and cf1='0');
--    end;
--
--    function advancefifor( rf0, rf1: std_logic) return boolean is
--    begin
--      return (rf0='1' and rf1='0');
--    end;
--
--    function availdata( isread, ev4: in std_logic )
--                return boolean is
--    begin
--      return ( isread='1' and ev4='1' );
--    end;
```



```
begin
  process( reset, clk )
  begin
    if (reset='1') then
        forceIdle <= '1';
        (cf0, cf1, rf0, rf1, wf2) <= std_logic_vector'("00000");
        (shiftrf, shiftcf, shiftwf) <= std_logic_vector'("000");
    else
     if (clk'event and clk='1') then
        if (writecmd='0' and (cf0='1' and cf1='0')) then
                cf0 <= '0';
                cf1 <= cf0;      --old cf0 value!
        elsif (writecmd='1' ) then
                cf0 <= '1';
        end if;
        if (not(cf0='1' and cf1='0') and (cf1='1' and rf0='0')
                                                and ev1='1') then
                cf1 <= '0';
        end if;
        shiftcf <= cf0 and not cf1;

        if (ev3='1') then wf2 <= '0';
        elsif (shiftwf='1' and shiftcf='0') then wf2 <='1';
        end if;
        shiftwf <= cf1 and not wf2;

        if ((rf0='1' and rf1='0') and
                                not (isread='1' and ev4='1')) then
                rf0 <= '0';
        elsif ((isread='1' and ev4='1')) then
                rf0 <= '1';
        end if;
        shiftrf <= rf0 and not rf1;
        if (readdata='1' and not (rf0='1' and rf1='0')) then
                rf1 <= '0';
        elsif ((rf0='1' and rf1='0') and
                                not (isread='1' and ev4='1')) then
                rf1 <= rf0;
        end if;

        if (cf1='1' and rf0='0') then
                forceidle <= '0';
        else forceidle <= '1';
        end if;

     end if;     -- if clk
    end if;      -- if reset
  end process;
end rtl;

----------------------------------------
-- buf2 (2 bits, loads on every clk edge)
----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity buf2 is
  port( clk: in std_logic;
        in0, in1: in std_logic;
        outs: out std_logic_vector(7 downto 0) );
end buf2;

architecture rtl of buf2 is
begin
  process( clk )
  begin
    if ( clk'event and clk='1' ) then
      outs <= (0=>in0, 1=>in1, others=>'0');
    end if;
  end process;
end rtl;

----------------------------------------------------
-- buf4spec (special 4 bit buffer, with forceidle)
```

```
-------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity buf4spec is
  port( clk, cs, force: in std_logic;
        ins: in std_logic_vector(3 downto 0);
        outs: out std_logic_vector(3 downto 0);
        iswrite: out std_logic;
        isread: out std_logic );
end buf4spec;

architecture rtl of buf4spec is
begin
  process( clk )
    subtype cmd_vector is std_logic_vector(3 downto 0);
    constant cmdidle: cmd_vector := "0011";

    function iswritefn( force: std_logic; ins: in cmd_vector)
                                          return std_logic is
    begin
      if (force='0' and ins(0)='0') then
        return '1';
      else
        return '0';
      end if;
    end;

    function isreadfn( force: std_logic; ins: in cmd_vector)
                                            return std_logic is
    begin
      if (force='0' and ins(0)='1') then
        return '1';
      else return '0';
      end if;
    end;

  begin
    if (clk'event and clk='1') then
      if (cs='1') then
        if (force='0') then
                outs <= ins;
        else
                outs <= cmdidle;
        end if;
        iswrite <= iswritefn(force, ins);
        isread <= isreadfn(force, ins);
      end if;
    end if;
  end process;
end rtl;

-------------------------------------------
-- buf8ldoe (ld and oe) -
-------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity buf8ldoe is
  port( clk, ld, oe: in std_logic;
        ins: in std_logic_vector(7 downto 0);
        outs: out std_logic_vector(7 downto 0) );
end  buf8ldoe;

architecture rtl of buf8ldoe is
        signal temp: std_logic_vector(7 downto 0);
begin
  process( clk )
  begin
    if ( clk'event and clk='1' ) then
      if (ld='1') then
        temp <= ins;
        if (oe='0') then
                outs <= "ZZZZZZZZ";
        else
                outs <= ins;
        end if;
      else      -- ld='0'
        if (oe='1') then
                outs <= temp;
        else
                outs <= "ZZZZZZZZ";
        end if;
      end if;
    end if;
  end process;
end rtl;

-------------------------------------
-- buf8oe (8 bits, with oe)
-------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity buf8oe is
  port( clk, oe: in std_logic;
        ins: in std_logic_vector(7 downto 0);
        outs: out std_logic_vector(7 downto 0) );
end buf8oe;

architecture rtl of buf8oe is
begin
  process( clk )
  begin
    if (clk'event and clk='1') then
      if (oe='1') then
        outs <= ins;
      else
        outs <= "ZZZZZZZZ";
      end if;
    end if;
  end process;
end rtl;

-------------------------------------
-- buf4ld (4 bits, with ld)
-------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity buf4ld is
  port( clk, ld: in std_logic;
        ins: in std_logic_vector(3 downto 0);
        outs: out std_logic_vector(3 downto 0) );
end buf4ld;

architecture rtl of buf4ld is
begin
  process( clk )
  begin
    if ( clk'event and clk='1' ) then
      if ( ld='1' ) then
        outs <= ins;
      end if;
    end if;
  end process;
end rtl;

-------------------------------------
-- buf8ld (8 bits, with ld)
-------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity buf8ld is
```

```
  port( clk, ld: in std_logic;
        ins: in std_logic_vector(7 downto 0);
        outs: out std_logic_vector(7 downto 0) );
end buf8ld;

architecture rtl of buf8ld is
begin
  process( clk )
  begin
    if (clk'event and clk='1') then
      if (ld='1') then
        outs <= ins;
      end if;
    end if;
  end process;
end rtl;

----------------------------------------
-- buf8csld (8 bits, with cs and ld)
----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity buf8csld is
  port( clk, ld, cs: in std_logic;
        ins: in std_logic_vector(7 downto 0);
        outs: out std_logic_vector(7 downto 0) );
end buf8csld;

architecture rtl of buf8csld is
begin
  process( clk )
  begin
    if (clk'event and clk='1') then
      if (cs='1' and ld='1') then
        outs <= ins;
      end if;
    end if;
  end process;
end rtl;

-- I needed this multiplexer because I had two tristate buffers
-- alternatively driving the ISA data lines; the synthesizing
-- CAD tools did not know how to deal with that so I had to
-- insert a multiplexer for the two output buses of those drivers
-- to thus connect them to the ISA data bus.
----------------------------------------
-- mux3state (8 bits)
----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;

entity mux3state is
  port( clk, oe1, oe2: in std_logic;
        ins1, ins2: in std_logic_vector(7 downto 0);
        outs: out std_logic_vector(7 downto 0) );
end mux3state;

architecture rtl of mux3state is
begin
  process( clk )
  begin
    if (clk'event and clk='1') then
      if (oe1='1') then
        outs <= ins1;
      elsif (oe2='1') then
        outs <= ins2;
      else
        outs <= "ZZZZZZZZ";
      end if;
    end if;
  end process;
end rtl;
```



```
----------------------------------------
-- now the whole controller
----------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
library SYNOPSYS;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity bc is
  port( isaclk, isareset,
        isaaen, isaale,
        isaiow, isaior: in std_logic;
        isaaddr: in std_logic_vector(9 downto 0);
        isadata: inout std_logic_vector(7 downto 0);

        tbaddr: out std_logic_vector(7 downto 0);
        tbdata: inout std_logic_vector(7 downto 0);
        tbcmd: out std_logic_vector(3 downto 0);
        tbclk1, tbclk2: out std_logic;
          -- this only for debugging purposes:
        writedata, writeaddr, writecmd: inout std_logic;
        readdata, readsts: inout std_logic;
        ev1, ev3, ev4: inout std_logic;
        shiftrf, shiftcf, shiftwf: inout std_logic;
        cf0, cf1, rf0, rf1, wf2: inout std_logic;
        statusoe, dataoe: inout std_logic);
end bc;

architecture struct of bc is

  component counter21
    port( clk, reset: in std_logic;
          count21: out std_logic);
  end component;

  component counter4
    port( clk, reset, inc: in std_logic;
          ev1, ev3, ev4: out std_logic;
          bclk1, bclk2: out std_logic);
  end component;

  component addrdec
    port( reset, ale, aen: in std_logic;
          addr: in std_logic_vector( 9 downto 0);
          seldata, seladdr, selcmd, selsts: out std_logic );
  end component;

  component readwrite
    port( reset, seldata, seladdr, selcmd,
                                selsts, iow, ior: in std_logic;
          writeaddr, writedata, writecmd,
                                writests: out std_logic;
          readdata, readsts: out std_logic );
  end component;

  component qcontroller
    port( reset, clk, ev1, ev3, ev4: in std_logic;
          writecmd, readdata: in std_logic;
          isread: in std_logic;
          shiftrf, forceidle: out std_logic;
          cf0, cf1, rf0, rf1, wf2,
                        shiftcf, shiftwf: inout std_logic );
  end component;

  component contrlogic
    port( isaclk, isareset, readdata, readsts,
                                        writests: in std_logic;
          clock, reset, dataoe, statusoe: out std_logic );
  end component;

  component buf8csld
  port( clk, ld, cs: in std_logic;
```

```
      ins: in std_logic_vector(7 downto 0);
      outs: out std_logic_vector(7 downto 0) );
end component;

component buf2
port( clk: in std_logic;
      in0, in1: in std_logic;
      outs: out std_logic_vector(7 downto 0) );
end component;

component buf4spec
port( clk, cs, force: in std_logic;
      ins: in std_logic_vector(3 downto 0);
      outs: out std_logic_vector(3 downto 0);
      iswrite: out std_logic;
      isread: out std_logic );
end component;

component buf8ldoe
port( clk, ld, oe: in std_logic;
      ins: in std_logic_vector(7 downto 0);
      outs: out std_logic_vector(7 downto 0) );
end component;

component buf8oe
port( clk, oe: in std_logic;
      ins: in std_logic_vector(7 downto 0);
      outs: out std_logic_vector(7 downto 0) );
end component;

component buf4ld
port( clk, ld: in std_logic;
      ins: in std_logic_vector(3 downto 0);
      outs: out std_logic_vector(3 downto 0) );
end component;

component buf8ld
port( clk, ld: in std_logic;
      ins: in std_logic_vector(7 downto 0);
      outs: out std_logic_vector(7 downto 0) );
end component;

component mux3state
port( clk, oe1, oe2: in std_logic;
      ins1, ins2: in std_logic_vector(7 downto 0);
      outs: out std_logic_vector(7 downto 0) );
end component;

signal reset, clk,
       --shiftrf, shiftcf, shiftwf, cf0, rf1,
       forceidle, iswrite, isread,
       --writedata, writeaddr, writecmd, readdata, readsts,
       writests,
       --dataoe, statusoe,
       countmax,
       --ev1, ev3, ev4,
       seldata, seladdr, selcmd, selsts: std_logic;
signal intercmd0, intercmd1: std_logic_vector(3 downto 0);
signal interaddr0, interaddr1, interdatar,
       interdataw0, interdataw1,
       interdataw2: std_logic_vector(7 downto 0);
signal interstatus: std_logic_vector(7 downto 0);
signal interdatar1: std_logic_vector(7 downto 0);

begin
  -- any initial values for signals ?? no, got reset

  u1: counter21 port map( clk => clk, reset => reset,
            count21 => countmax );

  u2:  counter4 port map( clk => clk, reset => reset,
            inc => countmax,
            ev1 => ev1, ev3 => ev3, ev4 => ev4,
            bclk1 => tbclk1, bclk2 => tbclk2);
```

```
  u3:  addrdec port map( reset => reset,
            ale => isaale, aen => isaaen,
            addr => isaaddr,
            seldata => seldata, seladdr => seladdr,
            selcmd => selcmd, selsts => selsts );

  u4:  readwrite port map( reset,
            seldata, seladdr, selcmd, selsts,
            isaiow, isaior,
            writeaddr, writedata, writecmd, writests,
            readdata, readsts );

  u5:  qcontroller port map( reset => reset, clk => clk,
            ev1 => ev1, ev3 => ev3, ev4 => ev4,
            writecmd => writecmd, readdata => readdata,
            isread => isread,
            shiftrf => shiftrf, forceidle => forceidle,
            cf0 => cf0, cf1=> cf1, rf0 => rf0, rf1 => rf1,
            wf2 => wf2,
            shiftcf => shiftcf, shiftwf => shiftwf );

  u6:  contrlogic port map( isaclk, isareset, readdata,
          readsts, writests,
          clk, reset, dataoe, statusoe );

  u7:  buf8csld port map( clk => clk,           --   buf0r
           cs => isread, ld => ev4,
           ins => tbdata, outs => interdatar );

-- u8 and u10 are not needed in the version with the multiplexer
--  u8:  buf8ldoe port map( clk => clk,          -- buf1r
--         ld => shiftrf, oe => dataoe,
--       ins => interdatar, outs => testisadata);

  u9:  buf2 port map( clk => clk,               -- buf0s
          in0 => cf0, in1 => rf1,
          outs => interstatus );

--  u10:  buf8oe port map( clk => clk,          -- buf1s
--        oe => statusoe,
--        ins => interstatus,
--        outs => testisadata );

  u11:  buf8ld port map( clk => clk,            -- buf0a
          ld => writeaddr,
          ins => isadata, outs => interaddr0 );

  u12:  buf8ld port map( clk => clk,            -- buf1a
          ld => shiftcf,
          ins => interaddr0, outs => interaddr1 );

  u13:  buf8ld port map( clk => clk,            -- buf2a
          ld => ev1,
          ins => interaddr1, outs => tbaddr );

  u14:  buf8ld port map( clk => clk,            -- buf0d
          ld => writedata,
          ins => isadata, outs => interdataw0 );

  u15:  buf8ld port map( clk => clk,            -- buf1d
          ld => shiftcf,
          ins => interdataw0, outs => interdataw1 );

  u16:  buf8ld port map( clk => clk,            -- buf2d
          ld => shiftwf,
          ins => interdataw1, outs => interdataw2 );

  u17:  buf8ldoe port map( clk => clk,          -- buf3d
          ld => ev3, oe => iswrite,
          ins => interdataw2, outs => tbdata );

  u18:  buf4ld port map( clk => clk,            -- buf0c
          ld => writecmd,
          ins => isadata(3 downto 0), outs => intercmd0 );
```

```
  u19:  buf41d port map( clk => clk,              -- buf1c
          ld => shiftcf,
          ins => intercmd0, outs => intercmd1 );

  u20:  buf4spec port map( clk => clk,            -- buf2c
          cs => ev1, force => forceidle,
          ins => intercmd1, outs => tbcmd,
          iswrite => iswrite, isread => isread );

  u21:  buf81d port map( clk => clk,              -- buf1r
          ld => shiftrf,
          ins => interdatar, outs => interdatar1 );

  u22:  mux3state port map( clk => clk,
        oe1 => statusoe, oe2 => dataoe,
        ins1 => interstatus,  ins2 => interdatar1,
        outs => isadata );
end struct;
```

Appendix D

Board Schematics

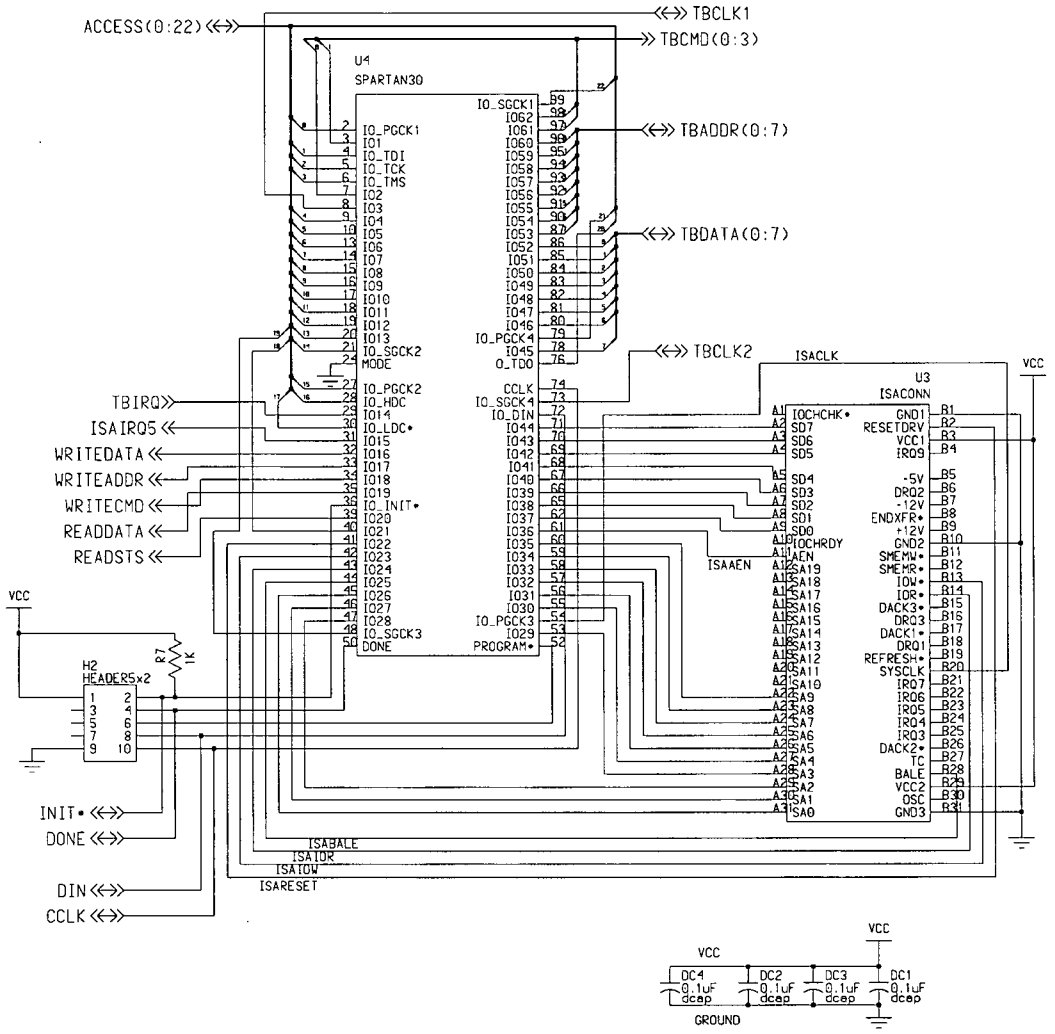

Figure D.1: Board schematic sheet 1

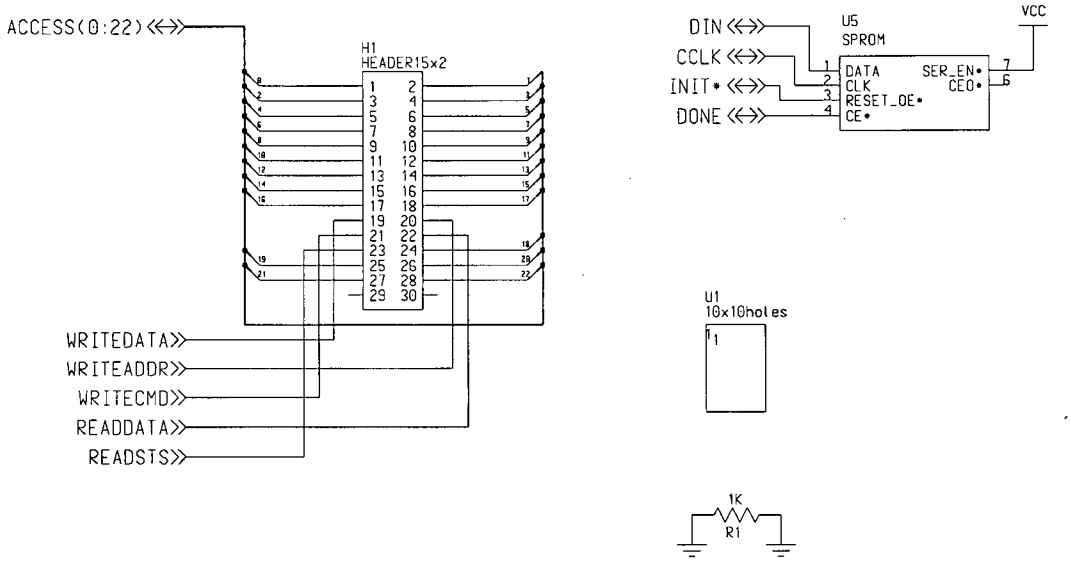

Figure D.2: Board schematic sheet 2

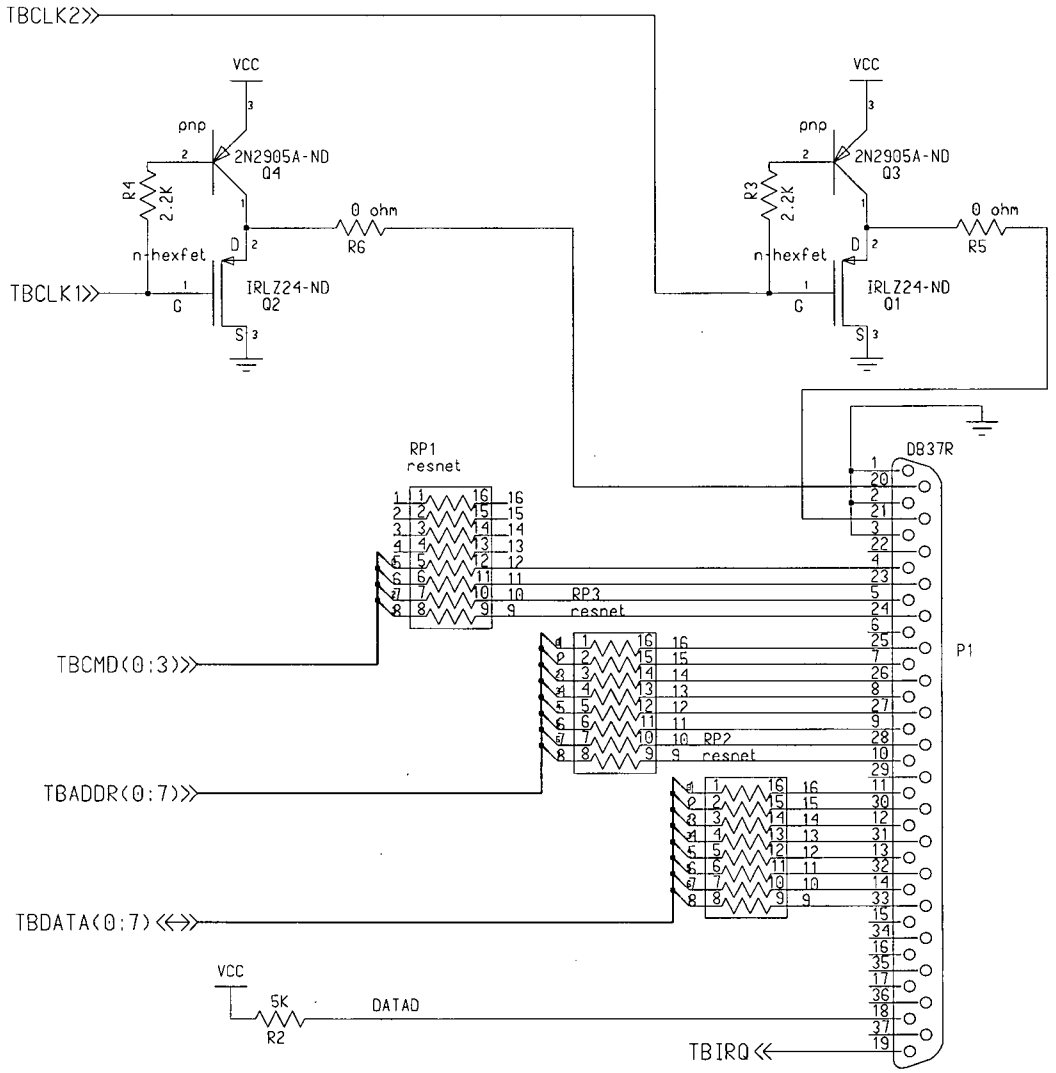

Figure D.3: Board schematic sheet 3

Appendix E

Board Design Details

A view of the board with the component geometries is shown in figure E.1.

The list of the components, as generated by the PC-board layout tools (Mentor Graphics), is given in figure E.2.

The train-bus will have 8 wires for ground, and 7 spare wires for future expansions and/or experiments. These include probably 1 wire for response (the one labelled DATAD in the schematics, figure D.1), to allow bus-repeaters to be used in large designs; and probably one wire for interrupt requests (one of the trainbus lines is connected to the FPGA on the board because I also planned a connection from the FPGA to ISA-IRQ5, so that if we later want to implement this, we may do it without hardware changes. Unfortunately I noticed I forgot this last connection, which was not necessary for the current version.

Also, the board currently has the pin holes for a configuration EEPROM to keep the data for programming the FPGA, so that it will not be necessary to always use the download cable to program the FPGA after power shutdown. The actual memory device is not soldered on, but the schematics were conceived having this operating mode in mind (the current version has the MODE-pin of the FPGA bent up, i.e., unconnected). The device I was planning to use is ATMEL's AT17C256A (application note on "FPGA Configuration EEPROM Programming Specification", Atmel, 1998).

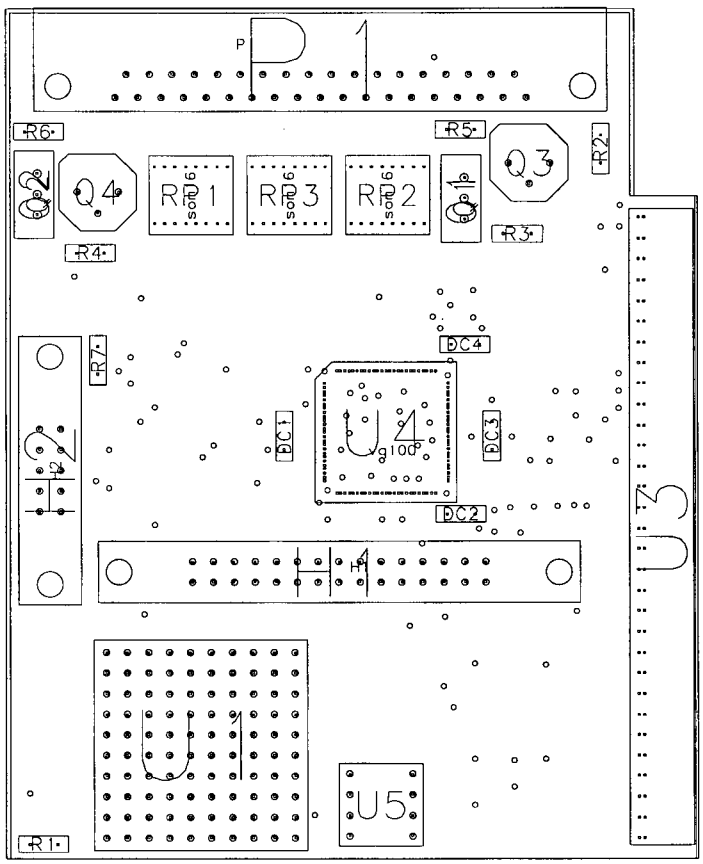

Figure E.1: The controller board with components

```
# Reference  Part_number  Symbol  Geometry  Properties
DC1     dcap          dcap          rc1206    (VALUE,"0.1uF")
DC2     dcap          dcap          rc1206    (VALUE,"0.1uF")
DC3     dcap          dcap          rc1206    (VALUE,"0.1uF")
DC4     dcap          dcap          rc1206    (VALUE,"0.1uF")
H1      header15x2    HEADER15x2    header15x2
H2      header5x2     HEADER5x2     header5x2
P1      tbconn        DB37R         db37r
Q1      nhexfet       n-hexfet      to220ab   (VALUE,"IRLZ24-ND")
Q2      nhexfet       n-hexfet      to220ab   (VALUE,"IRLZ24-ND")
Q3      pnp           pnp           to39      (VALUE,"2N2905A-ND")
Q4      pnp           pnp           to39      (VALUE,"2N2905A-ND")
R1      res           resistor      rc1206    (VALUE,"1K")
R2      res           resistor      rc1206    (VALUE,"5K")
R3      res           resistor      rc1206    (VALUE,"2.2K")
R4      res           resistor      rc1206    (VALUE,"2.2K")
R5      res           resistor      rc1206    (VALUE,"0 ohm")
R6      res           resistor      rc1206    (VALUE,"0 ohm")
R7      res           resistor      rc1206    (VALUE,"1K")
RP1     resnet        resnet        som16
RP2     resnet        resnet        som16
RP3     resnet        resnet        som16
U1      10x10         10x10holes    10x10holes
U3      isaconn       ISACONN       isaconn
U4      spartan30     SPARTAN30     vq100
U5      sprom         SPROM         dip8
```

Figure E.2: A list of the components placed on the controller board