Computer System Modeling

At The Hardware Platform Level

by

Amir Saghir

A thesis submitted to the

Faculty of Graduate Studies and Research in partial fulfilment of the requirements for the degree of

Master of Applied Sciences

In Electrical Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

September 2002

The undersigned hereby recommend to the Faculty of Graduate Studies and Research the acceptance of the thesis

"Computer System Modeling At The Hardware Platform Level"

Submitted By

Amir Saghir

In Partial Fulfillment of the Requirements for the Degree of Master of Applied Science

Dr. Gabriel Wainer	**Dr. Trevor Pearce**
(Thesis Co-supervisor)	**(Thesis Co-supervisor)**

Dr. Rafik Goubran
(Department Chair, Systems and Computer Engineering)

Department of Systems and Computer Engineering
Faculty of Engineering, Carleton University

September 2002

Abstract

Modeling and simulation plays an important role in the development of computer systems. This research work focuses on modeling general computer hardware platforms using the Discrete Event system Specification (DEVS) modeling formalism, and mapping these models onto the High Level Architecture (HLA) for simulation interoperability.

An HLA-based system level framework for a general hardware platform is proposed, in which hardware component details are modeled using DEVS atomic models. To verify the proposed framework, a case study is presented for a simplified Intel 8088 processor based platform. The components modeled in this case study are a processor, a memory module, an interrupt controller, a bus controller and a timer.

This research has developed an approach that flattens the modeling hierarchy by using only DEVS atomic models mapped onto the HLA. As a result, the simulation support layer normally present to handle coupled models is not needed. The general hardware platform framework influences aspects of component models, and these aspects are identified and related to their implications to the HLA. This work has laid some groundwork leading towards a simulation tool for hardware platform modeling and simulation that can be used by system designers during the product development phase.

Acknowledgements

I express my heartiest and most sincere gratitude to my supervisors Dr. Trevor Pearce and Dr. Gabriel Wainer for their help, supervision, and full support towards the completion of this thesis.

Needless to say, I am grateful to my parents and my dear wife for providing me all the emotional support and encouragement.

TABLE OF CONTENTS

List of Acronyms

BIU	Bus Interface Unit
CU	Control Unit
CS	Code Segment
DEVS	Discrete Event System Specification
DS	Data Segment
EA	Effective Address
EU	Execution Unit
FED	Federation Execution Data
FedExec	The Federation Executive
FOM	Federation Object Model
HLA	High level Architecture
IF	Internal Flags
INTA	Interrupt Acknowledge
INTR	Interrupt Request
IP	Instruction Pointer
LBTS	Lower Bound Time Stamp
LibRTI	The RTI library
LSB	Least Significant Byte
MSB	Most Significant Byte
OMDT	Object Model Development Tool
RO	Receive Order

RTI	Run time Infrastructure
RtiExec	The RTI Executive
SOM	Simulation Object Model
SP	Stack Pointer
SS	Stack Segment
TSO	Time Stamped Order

List of Figures

List of Tables

CHAPTER 1

Introduction

The use of embedded computer systems to solve application problems requires an understanding of both the hardware platform involved and the software that customizes the hardware for the particular application. Modeling and simulation is used increasingly in developing the hardware and software of such systems. Modeling is required to represent a system for the purpose of studying the system. A simulation is the imitation of the operation of a system over time. The behavior of a system as it evolves over time is studied by developing a simulation model [3]. The models created by hardware engineers while developing hardware components often contain details that are irrelevant to software developers. Furthermore, the simulation of the hardware component models is often fine-grained and computationally intensive, since the states of all of the individual transistors in the components must be addressed. Software developers would prefer a programmer's model of the hardware platform, which would abstract away irrelevant hardware details and focus only on the information relevant to software development. Ideally, the more abstract programmer's models would have larger-grained simulations that are less computationally expensive than the hardware models.

The goal of this research is to develop a generic framework that can be extended to model and simulate specific computing platforms by introducing platform-specific details.

Software developers can use these models to develop and test their software for computing platforms. For modeling platform components, the DEVS (Discrete event system specification) formalism is used. The simulation framework for these models is defined, following the specifications of the HLA (High-level architecture) standard. The rationale for adopting the DEVS/HLA approach is explained in chapter 4.

This research is seen as an initial step in a longer-term research project that will (later) include the development of tools to simplify the introduction of platform-specific details by handling the formal aspects associated with DEVS and the HLA (such tools would allow designers to focus more attention on the platform-specific details by relieving the designer of the unnecessary DEVS and HLA clutter).

A brief synopsis of the structure of the thesis document is as follows. Chapter 2 provides the background information about the DEVS formalism and the HLA. It briefly describes different classes of the DEVS formalism and key features of the HLA that are required to manage various aspects of the simulator. Chapter 3 reviews the latest ideas in the DEVS formalism and a subset of the tools developed to implement the DEVS formalism. It also describes a layered approach to use DEVS and the HLA for a simulator. Chapter 4 states the research motivation, a critical analysis of the facts of chapter 3, the research question under investigation, the approach adopted to answer the question and the scope of this research. Chapter 5 describes the proposed framework of the simulator and the general models of basic hardware platform components. These general models have a list of parameters, which can be used for defining the functionality of these components. Chapter 6 shows how this simulator can be used for a specific hardware platform. Execution of an example program (a few instructions of the 8088 processor) is simulated

in order to illustrate the proposed DEVS/HLA approach. It also describes the flow charts of each hardware component used in the case study. Chapter 7 shows the results of the case study. Excerpts of some component logs are discussed and the message flow between different components is explained. These logs and message flows are helpful in verifying the simulator. Chapter 8 concludes the research. It discusses the contributions and suggests some future research topics. The conclusion is followed by the references and two appendices. Appendix A refers to the software code for each hardware component involved in the case study. Appendix B refers to each component's log in the case study.

CHAPTER 2

Background for the DEVS and the HLA

In this section a system development process is discussed with emphasis on the need for modeling as an initial step in development. Computer systems being discrete in nature require an appropriate modeling formalism and a simulation framework to get the system analyzed. This section describes the Discrete Event System Specification (DEVS) formalism to model various components of a computer system and the High-level architecture (HLA) as a simulation framework standard.

2.1 System Development Method

The products based on computer systems are increasing tremendously. In order to verify the required features of these computer systems, modeling is becoming an important part of product development. Modeling has various levels of abstraction and complexity that are normally used during the development cycle of a system.

Computer systems are discrete in nature. Much research effort has been made in order to find the most appropriate modeling formalism for such systems. One of the current approaches is the use of computer simulation to analyze these models and explore the

properties of these systems. This modeling and simulation methodology is playing an important role in the development of computer platforms and systems.

2.2 The DEVS Formalism

The DEVS (Discrete Event System Specification) is a system's formalism, with a well-defined concept of modularity and coupling of components. The DEVS formalism focuses on the changes of variable values and generates time segments that are piecewise constant. In essence the formalism defines how to generate new values for variables and the times the new values should take effect. An important aspect of the formalism is that the time intervals are continuous. There are two major classes from which all the user-defined models can be developed – atomic and coupled.

A brief description of the DEVS Atomic model and Coupled model is as below. [4]

2.2.1 Atomic Model

An atomic model directly specifies the system's response to events on its input ports, state transitions, and the generation of events on its output ports. It is defined as [4]:

$AM = < IO, X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta >$

Where

IO is the model's interface (input/output ports).

X is the input events set.

S is the state set.

Y is the output events set.

δ_{int} is the internal transition function.

δ_{ext} is the external transition function.

δ_{con} is the confluent transition function

λ is the output function.

ta is the time advance function.

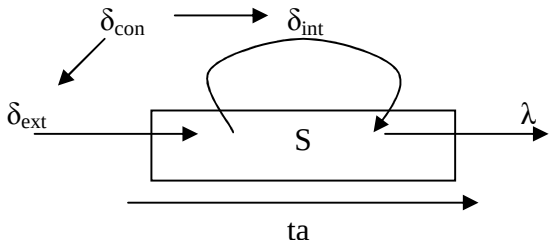


Figure 1: Systematic representation of a basic (atomic) model.

Figure 1 shows a graphical representation of an atomic model. Each model is provided with an interface consisting of input and output ports to communicate with other models. When an input arrives the instance variable S is updated. The δ_{ext} function accepts the input and changes the instance variable. After an elapsed time, given by the ta function, the system checks the internal variables and makes internal changes to bring the system to the next state. The function that performs all this is the δ_{int} function. The δ_{con} function decides what to do when both external and internal events occur together. It might, for example, decide the order between the δ_{ext} and the δ_{int} functions. The λ function produces the output Y from the instance variables. The ta function returns the time to the next internal event.

2.2.2 Coupled Model

A DEVS coupled model is composed of several atomic or coupled sub-models. It is defined as [4]:

CM = < IO, X, Y, D, {Mi}, {Ii}, {Zij} >

Where

IO is the model's interface (input/output ports).

X is the set of input events.

Y is the set of output events.

D is an index for the components of the coupled model.

Mi is a basic DEVS (that is, an atomic or coupled model).

Ii is the set of influencees of model i (that is, the models that can be influenced by outputs of model i).

Zij: Yi→ Xj is i to j translation function.

We can see that coupled models are defined as a set of basic components (atomic or coupled), which are interconnected through the models' interfaces. The influencees of a model define the models to which outputs must be sent. The translation function is in charge of converting the outputs of a model into inputs for the others. To do so, an index of influencees is created for each model (Ii). This index defines that the outputs of the model Mi are connected to the inputs in the model Mj, where j is an element of the set Ii.

2.3 High Level Architecture (HLA)

The HLA was developed by the US Department of Defense (DoD) based on a process involving government, industry and academia. The High Level Architecture (HLA) provides a general framework within which simulation developers can structure and describe their simulation application. In particular, the HLA addresses two key issues: promoting interoperability between simulations and aiding the reuse of models. In the terminology of the HLA [1]

- The combined simulation system created from the constituent simulation is a federation.
- Each simulation that is combined to form a federation is a federate.

The baseline of the HLA, defined in IEEE Standard 1516, includes the following:

- The HLA Rules define the responsibilities and relationships among the components of an HLA federation. [5]
- The HLA Interface Specification provides a specification of the functional interface between the HLA federates and the HLA Runtime Infrastructure (RTI) (RTI is discussed in section 2.3.1). This specification defines all the RTI services and identifies “callback” functions that must be provided by each federate. [6]
- The HLA Object Model Template (OMT) provides a common presentation format for HLA Simulation and Federation Object Models (SOM/FOM) [7]. A SOM defines the objects and interactions within a federate, while a FOM defines object models, communication between federates, condition for data updates and other information for interoperability purposes. The Federation Execution Data (FED), which is

required as an input to the RTI, is a subset of a FOM along with the specification of some default values for transport properties of data. In brief, the OMT provides an interface between the models and the RTI/HLA.

2.3.1 Runtime Infrastructure (RTI)

The RTI is a middleware that provides common services to federates and/or federations. It is an implementation of the HLA Interface Specification. The RTI software can be executed on a standalone computer or distributed over a network.

2.3.1.1 Components of the RTI

Figure 2 shows the major components of the RTI and each component is briefly discussed after the figure.

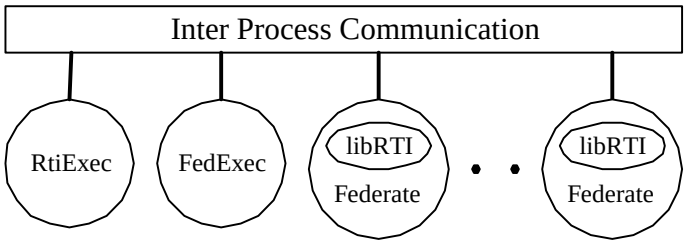


Figure 2: RTI Major Components. [8]

The federation executive (FedExec) manages multiple federates within a federation. It allows federates to join and to resign, and facilitates data exchange between participating federates. Each federate joining the federation is assigned a federation wide unique handle.

The RTI Executive (RtiExec) manages multiple federation executions in a network. It helps in initializing RTI components for each federation executive (FedExec). It also ensures that each FedExec has a unique name.

The RTI library (libRTI) provides the RTI services specified in the HLA Interface Specification to federate developers.

2.3.1.2 Management Areas of RTI

The HLA Interface Specification divides the services provided by the RTI into six management areas. Table 1 summarizes the objectives of each of the management areas:

Management Area	**Activities Supported**
Federation Management	Manages federation execution. Initializes name space, transportation, routing spaces etc.
Declaration Management	Specifies the data a federate sends and/or receives.
Object Management	Creates, modifies and deletes objects and interactions. Facilitates object registration and distribution. Coordinates attribute updates among federates. Accommodates various transportation and time management schemes.
Ownership Management	Supports transfer of ownership for individual object attributes. Offers both “push” and “pull” based transactions.

Time Management	Establishes or associates events with federate time. Regulates interactions, attribute updates, object reflection or object deletion by federate time scheme. Supports interaction between federates having different time schemes.
Data Distribution Management	Supports efficient routing of data.

Table 1: RTI Management Areas partitioned in FedExec life cycle [8]

CHAPTER 3

State of the Art in the DEVS/HLA

This section briefly describes the research work in the area of the DEVS formalism and a technique used to map the DEVS model formalism onto the HLA simulation framework.

3.1 Implementation of the DEVS formalism

The DEVS formalism is a well-defined means of expressing hierarchical, modular models in discrete event simulation. A DEVS model is a state machine, and the state of the model is changed by external or internal events with elapsed time. The following is a subset of the tools developed to map the DEVS formalism to simulation engines.

The *DEVS-C++* tool is based on the DEVS formalism. It is a modular hierarchical discrete event simulation environment implemented in the object-oriented C++ language [9]. The DEVS-C++ contains three libraries: container, devs and devsHLA. The container library provides methods, which are used to organize the interacting objects (e.g. atomic models). The devs library provides methods to implement the functions as δ_{ext}, δ_{int}, λ, ta and port information to be defined by the user. The devsHLA library provides easy access to the DEVS/HLA environment and is discussed in section 3.2.1.

The *CD++* toolkit is developed with the goal of developing and simulating models based on the DEVS and Cell-DEVS [11] paradigms. The core of the toolkit is the CD++ environment [11], which implements the DEVS and Cell-DEVS theories.

The *DEVSim++* is another tool that provides the ability to develop discrete event models using the hierarchical composition technology within the DEVS framework [12]. For simulation, DEVSim++ implements hierarchical scheduling in abstract simulators of atomic and coupled models.

3.2 DEVS Implementation over the HLA/RTI

B.P.Zeigler [2] has designed and developed an HLA compliant simulation environment using DEVS as a modeling tool and a way to map the DEVS models to the HLA as shown in figure 3. The strategy underlying the mapping of the DEVS to the HLA is to exploit the information contained within the DEVS models to automate as much as possible of the programming work required for constructing the HLA compliant simulations [9]. The final goal is to facilitate a bi-directional transfer of information between the OMT Development Tool (OMDT) that captures OMT information and the DEVS model description. Figure 3 shows the layered approach and each layer is discussed in the sub-sections below.

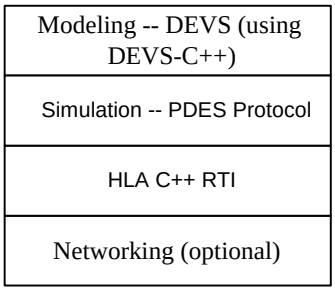


Figure 3: Layered approach to implement the DEVS/HLA. [2]

3.2.1 DEVS Modeling using DEVS-C++

The top most DEVS layer of figure 3 is used for defining the models. The DEVS-C++ tool [9], as discussed in section 3.1, has a devsHLA library required for interfacing between DEVS and the HLA/RTI in a C++ environment. The devsHLA library contains methods for attribute updates, attribute reflections, interaction updates, interaction receive, object discovery and quantizers. A quantizer object is associated with each attribute that the modeler would like to publish. A quantizer is a demon that checks for the attribute value crossing the threshold. Quantizers are used to reduce message update traffic and the size of the quantizers is directly related to the accuracy and the speed of computation required.

3.2.2 The Parallel DEVS (PDES) Protocol

The Parallel and Distributed Discrete Event Simulation (PDES) protocol layer of figure 3 introduces a simulation engine, which takes care of all the time and data interactions within the DEVS models. It also acts as a message translator between the DEVS models and the HLA/RTI.

There are three approaches to mapping the DEVS formalism into the PDES protocols: Conservative, Optimistic and the Parallel DEVS. Each approach is briefly discussed below.

The *conservative scheme* [9] processes events in strict time stamped order. Conservative schemes must somehow arrange for the potential for input events with earlier time stamps to be conveyed to affected models. This can be done through “lookahead” in which each

model provides a time in the immediate future up to which it promises not to send input events. The minimum of such blackout times at any model/component, called the Lower bound time stamp (LBTS), is the time up to which it can safely process its time-stamped inputs. Thus simulation proceeds incrementally governed by the lookahead, which is the interval that a model/component adds to its current LBTS to obtain the blackout time sent to other models/components.

The *optimistic scheme* [9] permits temporary time-stamped order violation that must be repaired before the final simulation output is presented. It allows models/components to march forward in local time and process their input and output queues as fast as they can. But from time to time a model/component can receive order messages with old time-stamps. To rectify this situation, queues of already processed inputs and their outputs are maintained so that the situation can be restored to what it was just before the arrival of the old time-stamped message. This scheme has an extensive apparatus of overheads.

The *Parallel DEVS scheme* [9] differs from the other schemes in that there is a coordinator to synchronize the simulation cycle through its steps (see Figure 4). Unlike other schemes it does not have overheads of lookahead and local time rollbacks. In parallel DEVS, the coordinator collects all times of next events from the component simulators. It sends the minimum of these times back to the components, thereby allowing them to determine whether they are imminent (component with the least time for next event), and if so to generate output. More than one component may be imminent, and the outputs of all imminent components are sorted and distributed to others according to the coupling rules. The transition functions of the imminent components, as well as all other recipients of inputs, are then applied. Which transition is applied, depends on the

state and input of a component. The resulting changes in states may cause new values for time advances and these are sent to the coordinator. The cycle then repeats.

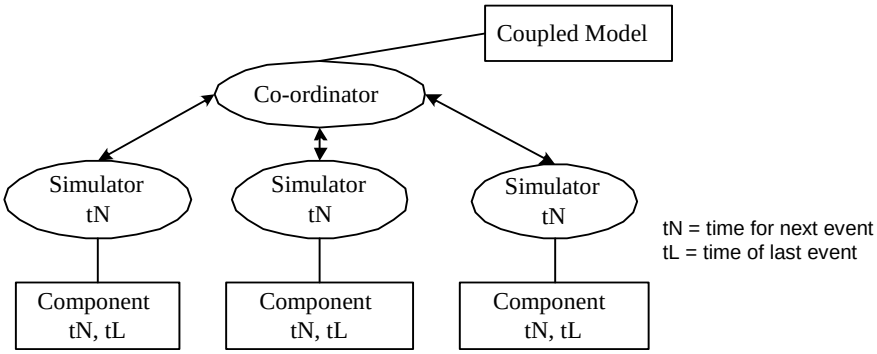


Figure 4: The Parallel DEVS simulation protocol [2]

3.2.3 Mapping of the PDES Protocol to the HLA/RTI

As shown in figure 3, B.P.Zeigler's approach to implement the PDES protocol in the HLA is as follows.

The DEVS simulation protocol is implemented with an explicit coordinator in the DEVS/HLA approach as illustrated in figure 5. Here, a separate Time Manager federate is allocated to the coordinator. This technique exploits quantizers (defined in section 3.2.1) to efficiently share the times of next events among DEVS federate and the coordinator. Only changes in local or global tN (minimum of local tNs) greater than the quantizer threshold are sent from one federate to the other using the RTI.

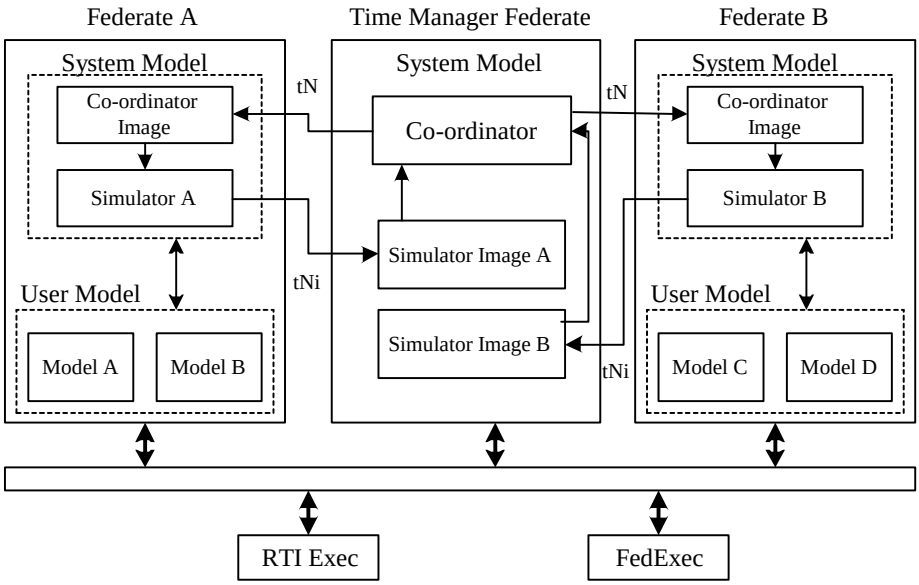


Figure 5: Coordinator as a Federate (the PDES protocol mapping into the HLA) [2]

CHAPTER 4

Modeling Hardware Platforms

In this chapter research motivation is discussed followed by a brief research statement and a critical analysis of the current solution to the problem as stated in chapter 3. Later, in this chapter, a solution to the research problem is proposed along with the scope of work, the contributions that are made and the presentation of the solution are discussed.

The development of systems involving computer/hardware platforms has increased tremendously. An important step during the development phase is the modeling and simulation at various levels of abstraction [10]. Simulation results play an important role in assessing the properties of the systems at various levels of abstraction.

Chapter 3 discusses a general approach to model and simulate different systems using the DEVS and the HLA/RTI. This approach can also be used in modeling and simulating computer systems and hardware platforms. A critical analysis of the technique/approach discussed in chapter 3 is as follows.

- The software tools for the DEVS implementations (section 3.1) are focused for hierarchical models (the DEVS coupled class). Hence they have complicated simulation and coordination engines in order to manage/synchronize different components. These engines perform well in the DEVS environment, but require additional message translation functionality to work with the HLA/RTI layer.

- The DEVS/HLA layered approach, discussed in section 3.2, is shown in figure 6. The two layers of simulator middleware (i.e., the RTI and the DEVS) are on top of each other. The DEVS layer is required to support the DEVS coupled models.

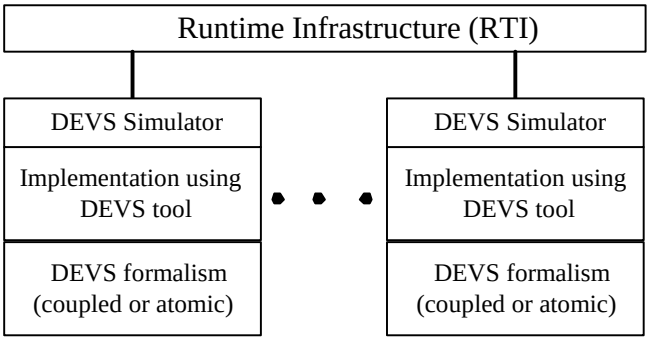


Figure 6: The block diagram for basic approach from chapter 3.

- B.P.Zeigler's work (chapter 3) adapts the DEVS-C++ tool to the HLA/RTI structure. This is quite useful to reuse old simulations developed under the DEVS-C++ tool but suffers from the extra layer of DEVS middleware.

The motivation of this research work is to model various components of computer systems and define a simulation framework so that these components can interact with each other and give some useful results at various levels of abstractions. Another motivation is that the components could be reused in other simulations.

The research problem is to define a framework for a general hardware platform model, of a single master computer system, and to map the model onto the High Level Architecture (HLA) simulation guideline.

To address the above research problem, the DEVS formalism is used to model a hardware platform. This formalism is used, as the DEVS is an increasingly accepted paradigm for understanding and supporting the activities of modeling and simulation. In

order to provide a simple solution to the research question, only the DEVS atomic class is used to model components. The rationale behind this approach is as follows.

- Atomic models can be controlled directly from the RTI, therefore an extra layer of DEVS middleware to support coupled models is not required. This approach effectively flattens the DEVS modeling hierarchy.
- Atomic models are a simple concept, and easily reusable for future developments.

The suggested layered approach to address the research problem is shown in Figure 7.

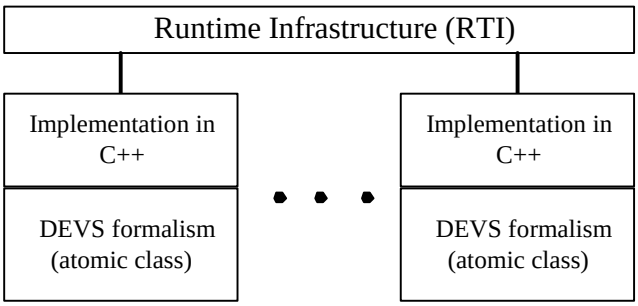


Figure 7: A Model of the proposed solution.

The scope of this research is to

- Define a simulation framework for a general hardware platform. The general hardware components to be modeled are a Processor module, a Memory module, a Bus controller module, an Interrupt module and a Timer module. Being a single master (Processor) platform, a bus arbiter is not modeled. The Processor module is further divided into three units. An Execution Unit (EU) that is responsible for decoding and executing all instructions, a Bus Interface Unit (BIU) that is responsible for performing all external bus operations and a Control Unit (CU) that is responsible for minimum/maximum mode and interrupt signals from other devices. Any required

hardware component can be modeled and added, as a future work, below the simulation (RTI) layer.

- Identify the general attributes of each hardware component in order to use these models in the initial development phase of the computer systems. The level of abstraction for these components is in the unit of bus cycles. More detailed time breakdown within a bus cycle and detailed bus level signal implementation are outside the scope of this work and may not be modeled accurately. For the execution unit (EU) of the processor the timing details are implemented up to a clock cycle.
- Use the DEVS formalism to apply the modeling and mapping results to a case study involving hardware platform components. For this case study, a Processor (a simplified version of Intel 8088), a basic Memory unit, a bus controller (a simplified version of Intel 8288), a basic Interrupt controller and a basic Timer unit are modeled.
- Execute simple operational codes as a main program and an interrupt routine on this simulator for the case study and obtain results to verify the operation of the simulator.
- Draw conclusions for the proposed solution's (discussed above) implementation in the case study.
- Summarize the contributions made by this research and the case study.
- Identify possible future research topics.

The following paragraphs identify the contributions that are made, and the approach to presenting the proposed solution.

As a contribution, a mapping of computer/hardware platforms to the HLA/RTI is developed. An approach is defined for the DEVS atomic models to work under the HLA

framework without using any additional DEVS simulators. A simulator is developed, as a case study, by modeling the hardware platform components (as discussed above in the scope of the research) using the DEVS formalism and making all the interactions between these components using the HLA framework.

In order to present the solution, the body of the thesis is organized in chapters starting from the system level discussion of hardware platform modeling using the HLA, defining the DEVS models for each hardware component, implementing the design as a case study for a simple (Intel 8088 based) platform by developing a simulator and analyzing the simulator's results to verify the proposed solution/design.

As discussed in the scope of the research, chapter 4, several modules are modeled for the platform. To reduce the amount of low-level detail only the memory module is presented in depth. For the complete implementation of all modules, appendix A.2 refers to the software code attached with the thesis. The reason for selecting the memory module, as a reference is that it is a simple module and also an important part of hardware platforms.

CHAPTER 5

Simulation Framework and DEVS Atomic models

This chapter includes the system level framework suggested to approach the solution. The discussion below starts from a system level block diagram and is later divided into two parts. In the first part the simulation flow of a Federate is described, and in the second part the DEVS atomic model for each federate is explained.

5.1 System Level Block diagram

Figure 8 shows a system level block diagram for a simple hardware platform simulator. It contains the following basic hardware components: Processor module, Bus controller module, Memory module, Interrupt controller module and Timer module. The Processor module is further divided into three units: BIU, EU and CU. For modeling a single master hardware platform (chapter 4), a bus arbiter module is not required. All these components/modules are modeled using DEVS atomic models and are discussed in detail later in this chapter. In terms of the HLA, each individual component is a federate, and all federates collectively form a federation. The Runtime Infrastructure (RTI) provides the simulation platform for this federation.

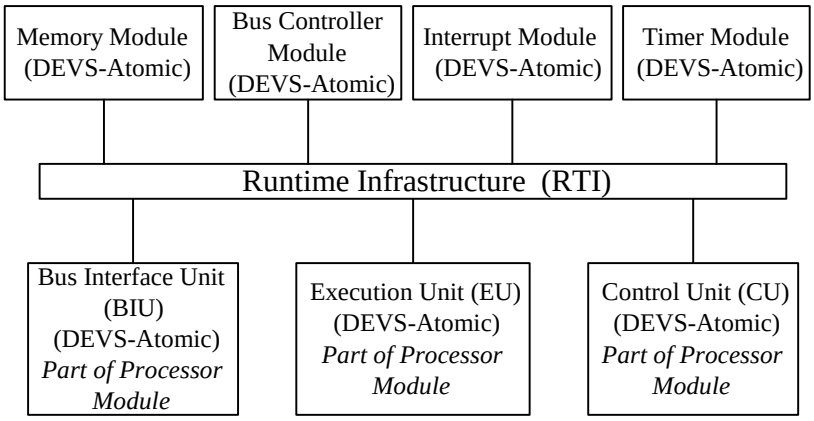


Figure 8: System-level block diagram for Hardware platform simulator.

5.1.1 System Level Flow Diagram for a Federate

Figure 9 shows a basic simulation flow of a federate. On the left side, the figure is labeled with numbers, which correspond to the steps in the flow chart in that row. Each step is discussed in this section by using the label references.

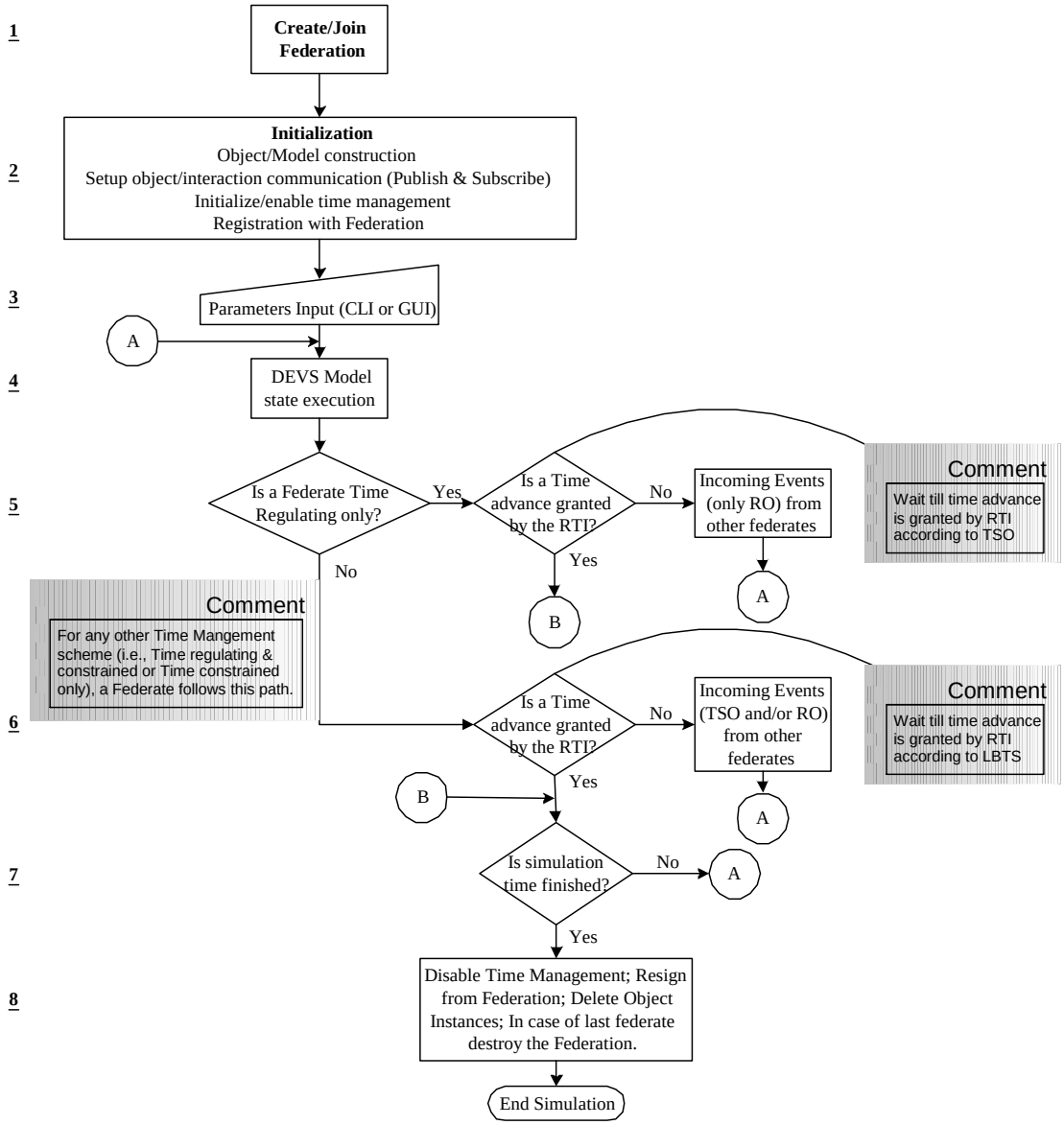


Figure 9: Simulation flow chart of a federate.

Step 1 is the initial step of the simulation in which a federate creates or joins a federation. If the federation does not exist, a new one is first created and then the federate is added to it. If a federation already exists, then the federate joins that federation.

At step 2, the federate is initialized. Initialization consists of the Object/Model construction, defining the attributes or/and parameters as to be published or/and

subscribed (can be re-defined during simulation), defining the federate as time regulating or/and time constrained (can be re-defined during simulation) and the registration of the federate with the federation executive. The SOM/FOM (defined in section 2.3) of each federate must also contain these published and/or subscribed attributes and/or parameters. The registration process registers the object with the federation execution, which returns an HLA object handle – an identifier for all interactions within the federation.

Step 3 allows a user to input parameters for a federate before starting the simulation loop. This step is optional, as some federates do not need any user input.

Step 4 is the first step in the simulation loop. In this step, a federate executes state transition functions according to its DEVS atomic model. The DEVS atomic models are discussed later in this chapter.

Following step 4, a federate will perform one of steps 5 or 6, depending upon its time management scheme. Step 5 is performed if a federate is time regulating (capable of sending Time Stamped Order (TSO) messages) only. The RTI grants a time advance according to the TSO message time. On the other hand if there is some incoming Receive Order (RO) message, a message with no time stamps, from another federate then the RTI delivers this message to the federate without granting a time advance. Either a time advance is granted or not, the simulation proceeds and the DEVS model transitions from one state to another.

Step 6 is performed if a federate is using either the time regulating & time constrained (capable of receiving TSO messages) both or the time constrained only scheme. The RTI grants a time advance according to the Lower Bound Time Stamp (LBTS). The LBTS is the maximum time to which a time constrained federate may advance. The LBTS is

determined by the RTI after getting the lookahead (a time period during which a federate does not send out any message) values for each federate and TSO messages from all the time regulating federates. On the other hand if there is some incoming message (either TSO or RO) from another federate then the RTI delivers this message to the federate without granting a time advance. If a time advance is granted, the federate proceeds to step 7. If a time advance is not granted, the federate returns to the top of the simulation loop for another DEVS model state transition.

At step 7, the federate checks whether the simulation loop will continue or stop. At step 8, a federate removes itself from the federation by disabling time management, resigning from the federation, deleting object instances, and, in the case where it is the last federate, destroys the federation.

5.1.2 DEVS Atomic Models for each Federate

Here, DEVS atomic models for each module in Figure 8 are discussed. The memory module is discussed in detail with external/internal transition functions, time advance function and the output function. In the following sub-sections, input and output ports refer to the DEVS models as discussed in section 2.2 should not be confused with similar hardware terminology.

5.1.2.1 MEMORY MODULE

The level of abstraction, as discussed in the scope of work chapter 4, can be achieved at bus cycles level. So all of the detailed signals within a bus cycle are not discussed in the

model. The parameter that a user can set, as discussed in step 3 of Figure 9, is the following.

Parameter

- Memory size (MS). This parameter has an integer value (MS > 0) and is defined in terms of bytes.

Model

Memory = < IO X S Y δ_{int} δ_{ext} δ_{con} λ ta >

IO – Input and output ports

- Input ports:
  - Address bus: The size of address bus (AB_Size) is an integer (AB_Size > 0) and is defined in the processor module's parameters section 5.1.2.2. To support access to all memory locations, MS ≤ 2^{AB_Size}.
  - Data bus: The size of the data bus (DB_Size) is an integer (DB_Size > 0) and is defined in the processor module's parameters section 5.1.2.2.
  - Read control signal: This is a single bit signal. 1 = Read, 0 = no operation specified.
  - Write control signal: This is a single bit signal. 1 = Write, 0 = no operation specified.
- Output port:
  - Data bus: The size of the data bus is DB_Size as above.
  - Data acknowledge signal (for Motorola 68000 series): This is a single bit signal. 1 = Acknowledge, 0 = no operation specified.

X – Input event set

- Address on address bus: A binary value, in the range of $\{0,\ldots, 2^{AB\_Size}-1\}$. In order to point to a memory location, the address must be < MS.
- Data on data bus: A binary value in the range of $\{0,\ldots, 2^{DB\_Size}-1\}$.
- Read control signal $\in \{0,1\}$.
- Write control signal $\in \{0,1\}$.

Y – Output event set

- Data to data bus: A binary value as above.
- Data acknowledge signal (DTACK) informs the processor that the bus cycle has ended during an asynchronous processor's mode e.g., Motorola 68000 series processors. DTACK $\in \{0,1\}$.

S – State set

- Read state: In this state the memory module waits until it sends out data on the data bus acquired from the memory location as addressed by the address bus. At this level of abstraction (chapter 4), the delay function associated with this state is one bus cycle.
- Write state: In this state the memory module writes data, provided by the data bus, to a memory location which is addressed by the address bus. The module then waits until the time delay associated with this state is elapsed. At this level of abstraction (chapter 4), the delay function associated with this state is one bus cycle.
- Wait state till next δ_{ext}. The delay function associated with this state $\in R^{+}$.

δ_{ext} – External transition function

- The δ_{ext} starts with the arrival of read/write signal on the input port of the module. In case of write signal the δ_{ext} acquires the information from the address and data input ports. In case of read signal the δ_{ext} only acquires the information from the address input port. As an example the state diagram for memory read and memory write during a minimum operation mode is shown in
- Figure 10.

ta – Time advance function

- The ta function introduces the time delay before scheduling the next output function λ.
- Figure 10 shows an example where the ta function is introduced in the read state before carrying out the λ function.

λ – Output function

- The λ outputs the data on the output port (data bus) during the read cycle. For the case of Motorola 68000 series processors the DTACK signal is also sent out on the output port. As an example see
- Figure 10.

δ_{int} – Internal transition function

- The δ_{int} changes the internal state from read/write to the wait state. In the wait state the module stays until the next δ_{ext}. As an example see
- Figure 10.

δ_{con} – Confluent transition function

- The δ_{int} has higher priority than the δ_{ext}

State Diagram for minimum operation mode

Figure 10 shows a state machine diagram for the memory module. Being simpler, the minimum operation mode is used to show the transitions between different states of the module for memory read and memory write operations.

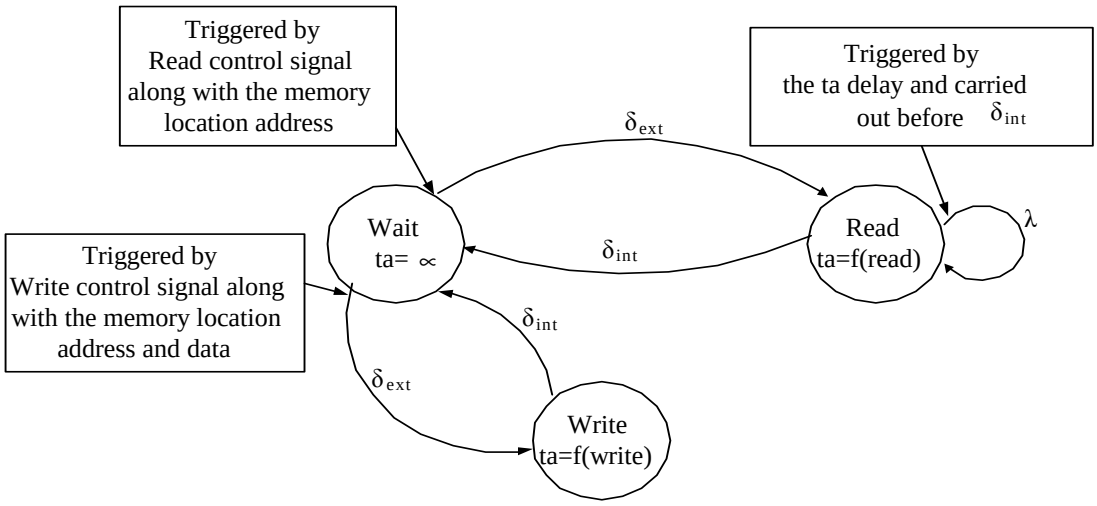


Figure 10: State diagram for memory read and write for minimum operation mode.

5.1.2.2 PROCESSOR MODULE

A general-purpose processor module is further divided into three sub-modules. Each sub-module can become a separate DEVS atomic model and communicate with the other modules using RTI services. Figure 11 shows a block diagram of the processor module.

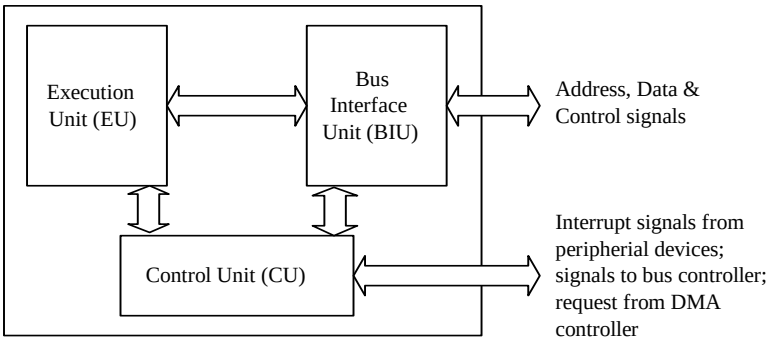


Figure 11: Block diagram showing sub-modules in a Processor model.

Processor's Parameters

As discussed in step 3 of Figure 9, a user can define the following processor's parameters.

- Duration of Bus/Machine cycle (e.g., 8086 ≈ 4 clock cycles).
- Instruction Queue size for Bus Interface Unit (if present).
- Size of External Data (DB_Size) and Address (AB_Size) buses.
- External data/address buses are multiplexed or separate (for multiplexed buses external de-multiplexing circuitry is compulsory).
- Internal size of buses and registers.
- Minimum/Maximum mode (this feature is Intel specific).
- Built in cache memory - if yes, cache size?

Each sub-module is modeled as follows.

5.1.2.2.1 Bus Interface Unit (BIU)

The BIU contains an instruction queue (first in first out), segment registers, instruction pointer and address/data/control buses. The main purposes of the BIU are to keep the instruction queue filled with instructions, to generate and accept the system control

signals and to act as a window between the execution unit (EU) and the memory/input-output (IO) devices. The DEVS atomic model for the BIU is as follows.

$$BIU = < IO\ X\ S\ Y\ \delta_{int}\ \delta_{ext}\ \delta_{con}\ \lambda\ ta >$$

X – Input event set

- Data on external data bus.
- Signal to indicate that the bus cycle has ended from the peripheral devices. This signal is required if a processor is operating in asynchronous mode e.g., Motorola 68000 series processor.
- Signals from the EU for data read/write (RD/WR) from memory or the IO devices.
- Address and Data from the EU via an internal bus.
- Signal from the EU to update instruction queue status/pointer.
- Signal from the control unit at the end of each instruction cycle indicating whether there is an interrupt request.
- In case of interrupt and jump, the execution unit shall indicate to the BIU to reset the instruction queue.
- Signal from control unit at the end of each bus cycle to indicate whether the system buses have been given under the control of DMA.
- Processor's parameters set by the user.

Y – Output event set

- Internal and external Address/data buses.
- Control signals to peripheral devices.

- Signal to control unit at the end of each bus cycle (this signal helps the control unit to act on the DMA's request).

S – State set

- Opcode fetching state.
    - Making physical address using code segment (CS) & instruction pointer (IP). This is Intel specific.
    - Read data.
    - Increment IP register.
    - Increment instruction queue status/pointer.
- Data read state from the memory/IO device.
- Data write state to the memory/IO devices.
- Interrupt state.
- Wait state until the processor gets control of the system buses.
- Wait state until next external transition function.

5.1.2.2.2 Execution Unit (EU)

The purpose of the EU is to decode and execute the instructions that are fetched from the instruction queue. The EU contains an arithmetic and logic unit (ALU) and a register array. The ALU performs arithmetic and logic operation on memory or register data. It also contains pointer or index registers used to address operand data located in the memory. The DEVS atomic model for the EU is as follows.

$$EU = < IO\ X\ S\ Y\ \delta_{int}\ \delta_{ext}\ \delta_{con}\ \lambda\ ta >$$

X – Input event set

- Signal from the BIU:
    - Arrival of an opcode in the queue.
    - A signal at the completion of each request.
- Signals from the control unit:
    - Interrupt request signal at the end of each instruction cycle.
    - A signal at the end of each bus cycle to indicate whether the control of system buses is given over to a DMA.
- Processor's parameters set by the user.

Y – Output event set

- Signal to the BIU to execute RD/WR operation according to the decoded instruction.
- Signal to the BIU to reset the instruction queue in the case of jump and interrupt statements.
- Signal to the control unit at the end of each instruction cycle (The control unit can only respond to an interrupt request signal at the end of an instruction cycle).

S – State set

- Opcode exection state. This state has following functionality.
    - Read the opcode from the instruction queue.
    - Decode the opcode.
    - Execute the opcode. During this stage the EU has a time elapse until the BIU finishes the read or write cycle.
    - Updates instruction queue pointer.
    - Maintains the status of flag register.

- Wait state until next external transition function.

5.1.2.2.3 Control Unit (CU)

The main function of the CU is to take care of the maskable interrupt request signal, to check the non-maskable interrupt (NMI), to generate signals for the bus controller and execution unit and to check any request from the DMA/controllers to control the external bus. The DEVS atomic model for the CU is as follows.

$$CU = < IO\ X\ S\ Y\ \delta_{int}\ \delta_{ext}\ \delta_{con}\ \lambda\ ta >$$

X – Input event set

- Signal from the BIU at the end of each bus cycle.
- Signal from the EU at the end of each instruction cycle.
- Signals from peripheral devices e.g., the DMA and the interrupt controller.
- Processor's parameters set by the user.

Y – Output event set

- Signal to the EU for the interrupt request and DMA access to the buses.
- Signal to the BIU for the interrupt request and DMA access to the buses.
- Status signal to bus controller, prior to the initiation of bus cycle. These status signals $S_0\ S_1\ S_2$ are required for Intel specific maximum operation mode. These three bits are decoded by the bus controller, which then (instead of the processor) generates the appropriate control signals to the peripheral devices e.g., memory RD/WR, IO port RD/WR, interrupt acknowledge and instruction fetch signals.

S – State set

- The state of an interrupt request. (It checks the signal from the interrupt controller, the state of the instruction cycle of the processor and the status of the interrupt flag. If all the conditions are in favor of executing the interrupt routine, it generates appropriate signals to the EU and the BIU).
- The state of an external bus control request. (It checks for a signal from the DMA controller and the state of machine/bus cycle of the processor. If the conditions are satisfied it generates appropriate signals to the EU and the BIU in order to accomplish the DMA request).
- Wait state until next external transition function.

5.1.2.3 BUS CONTROLLER MODULE

The bus controller module generates the control signals for the memory RD/WR from memory/input output devices, interrupt acknowledge and instruction fetch depending upon the status signal (S_0 S_1 S_2) [14] provided by the CU of the processor module. This Intel specific bus controller module also generates some bus level control signals, which are not in the scope of this work. The DEVS model of the bus controller is as follows.

$$BC = < IO\ X\ S\ Y\ \delta_{int}\ \delta_{ext}\ \delta_{con}\ \lambda\ ta >$$

X – Input event set

- Status signals S_0 S_1 S_2 from the processor module.

Y – Output event set

- Read/write control signals to the memory and input output devices.
- Interrupt acknowledge control signal.

S – State set

- Read state from the memory/input output devices.
- Write state to the memory/input output devices.
- Interrupt state. It sends out an interrupt acknowledge signal to the interrupt module.
- Wait state until next external transition function.

5.1.2.4 INTERRUPT CONTROLLER MODULE

Function

The basic function of this module is to acquire interrupt signals from the peripheral devices connected to the input ports, prioritize these requests and send the highest priority signal to the processor. It also sends out the interrupt type number on the data lines after getting the interrupt acknowledge signal from the processor or the bus controller.

Model

$$IM = < IO\ X\ S\ Y\ \delta_{int}\ \delta_{ext}\ \delta_{con}\ \lambda\ ta >$$

X – Input event set

- Interrupt acknowledge signal from the processor and/or the bus controller.
- Interrupt from external devices.
- Input data on the external address and data buses.
- RD/WR signal from the processor and/or the bus controller.

Y – Output event set

- Interrupt request (INTR) signal or priority signals (three signals for the case of Motorola 68000 series) to the processor.

- Data out to the system data bus (to output vector type).
- Data acknowledge signal (DTACK) informs an asynchronous processor that the bus cycle has ended e.g., Motorola 68000 series processors.

S – State set

- State of initiating an interrupt request. It prioritizes the incoming interrupt signals from the peripheral devices and sends out the interrupt request or the interrupt priority code to the processor (CU).
- State of sending vector type. The interrupt controller sends out the vector type on the data bus as a result of receiving the interrupt acknowledge signal.
- Wait state till next external transition function.

5.1.2.5 TIMER MODULE

This module generates a periodic signal to act as an interrupt source for the interrupt controller. As discussed in step 3 of Figure 9, a user can define the following attributes/parameters.

Parameters

- Duty cycle of the periodic signal.
- Number of interrupts to be generated.

Model

$$TM = < IO\ S\ Y\ \delta_{int}\ \delta_{ext}\ \delta_{con}\ \lambda\ ta >$$

Y – Output event set

- Periodic signal sent to the interrupt controller.

S – State set

- Signal generation state.
- Wait state for time elapse. The user can set this period.

CHAPTER 6

Case Study

6.1 Introduction

A hardware platform containing a processor (a simplified version of Intel 8088), a basic memory unit, a bus controller (a simplified version of Intel 8288), a basic interrupt controller and a basic timer unit is modeled using DEVS formalism. These modules interact with each other under a framework that is HLA compliant. This case study is for a synchronous platform, but no clock module is modeled. Each module assumes synchronous interactions and requests to schedule the future bus events as an integral number of clock ticks in the future.

The proposed simulation framework, as discussed in chapter 5, for hardware platform modeling using the DEVS/HLA approach is verified by the case study. This case study is done for the minimum and maximum operating modes of the processor. The user makes the selection of the operation mode (input through the keyboard) at the beginning of the execution of the processor federate as discussed in step 3 of Figure 9. For this case study, a simple simulator is developed to run the instruction codes shown in Tables 2 and 3. Later in this chapter the development of the simulator along with the hardware components' implementation are discussed.

6.1.1 Case study program

The example program generates 10 even numbers, from 0 to 18, and stores them in the memory module. The opcodes are shown in Table 2 along with the opcode values and the clock timing information [13]. Instruction execution times are determined by taking the number of clocks required per instruction plus any effective address (EA) time required for the operand. The EA for the indexed operand in the MOV [BX],CH instruction is 5 clocks.

Opcode (Hex)	**Program Instructions**	**Clock Cycle**
B500	MOV CH , 0	4
BF0000	MOV DI , 0	4
B102	MOV CL , 2	4
BB0000	xyz : MOV BX , 00	4
03DF	ADD BX , DI	3
882F	MOV [BX] , CH	9+EA = 9+5 = 14
47	INC DI	2
02E9	ADD CH , CL	3
83C70A	CMP 10 , DI	4
750D	JNE xyz	4 (when not executed) 16 (when jump executed)
F4	HLT	2

Table 2: Main program's opcodes for the case study

During the execution of the main program, the interrupt controller module sends interrupt signals to the processor, resulting in the execution of the interrupt routine (opcodes

shown in Table 3). The processor's interrupt behavior includes checking the interrupt flag; pushing the IP, CS and IF; sending the interrupt acknowledge to the interrupt controller; reading the vector type from the interrupt controller; calculating the starting address for the interrupt routine residing in the memory and clearing the instruction queue. The interrupt routine used in the case study increments a variable stored in memory. The opcodes for the interrupt routine along with the clock timing information are shown in Table 3 [13].

Op-code (Hex)	**Program Instructions**	**Clocks**
53	PUSH BX	15
BB0020	MOV BX , 32	4
8A07	MOV AL , [BX]	8+EA = 8+5 = 13
FEC0	INC AL	3
8807	MOV [BX] , AL	9+EA = 9+5 = 14
5B	POP BX	12
CF	IRET	32

Table 3: Interrupt routine's opcodes for the case study

6.2 Simulator Implementation

This section briefly discusses the system level block diagram for the simulator required for the case study. Later in this section each module/component is discussed in terms of the assumptions made for the implementation and flow charts for the C++ code.

6.2.1 System level Block diagram

Here the Bus interface unit (BIU), execution unit (EU) and control unit (CU) are combined as a single processor module and implemented as a DEVS atomic class. A bus controller module, a basic memory module, a basic interrupt controller module and a basic timer module are implemented as defined in the scope of the research, chapter 4. The implementation details of each module are discussed later in this chapter. Figure 12 shows an overall system level block diagram implemented for the case study.

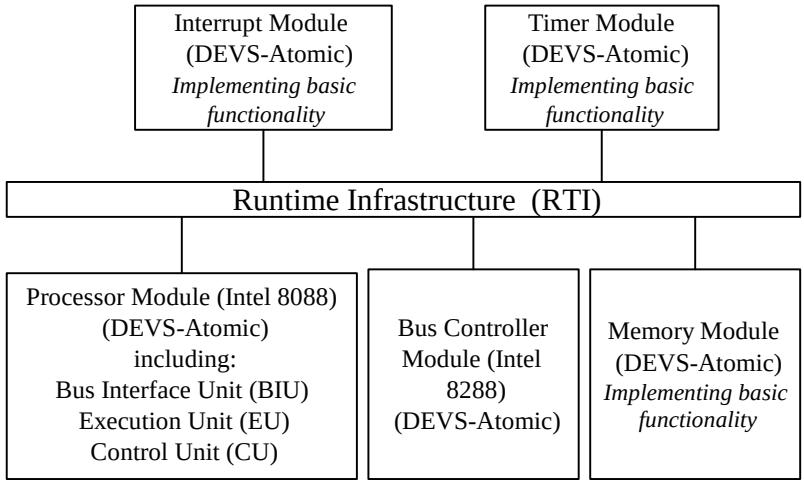


Figure 12: System level block diagram implemented for the case study.

6.2.1.1 Level of abstraction for the simulator's timing

For this case study one bus cycle of the processor is assumed to be four time units of the simulator. Within a bus cycle the timing details are not implemented accurately (scope of the research, chapter 4). For the execution unit of the processor the timing details are implemented with one clock cycle accuracy.

6.2.1.2 Algorithm to start the simulation

All federates in a federation should start the simulation simultaneously, as is the case for components in real hardware platforms. To achieve this objective, the following algorithm with two steps of signaling is defined.

The first step requires that the processor should wait until the other modules join and subscribe to the published attributes of the processor. This is achieved by executing the "enableAttributeRelevanceAdvisorySwitch()" RTI service so that the processor federate could receive the callbacks from the other federates.

The second step requires that all federates should wait for a signal from the processor federate to start the simulation. In this case study the attribute SimEnd_Processor (see Figure 15) is the signal published by the processor module and subscribed by all the other modules. The value SimEnd_Processor=0 indicates the starting of the simulation. The algorithm's second step addresses the scenario in which a federate has joined a federation and has no attributes to publish to a controlling federate (e.g., Processor) so the federate will not get any callbacks from the controlling federate.

6.2.1.3 Stopping the simulation

If the attribute SimEnd_Processor, as discussed in section 6.2.1.2, is set to 1 by the processor module then a signal is provided to the other modules (subscribers of the attribute) to stop the simulation.

6.2.1.4 Communication among Modules/Federates

Attributes are sent, from one federate to the other, in the form of AttributeHandleValuePairSet (AHVPS). AHVPS is a set comprised of attribute handles, values and the size of the values. To create the AHVPS, a method CreateNVPSet(), extracted from the memory module's software code, is defined in Figure 13. In this example the RTI method "AttributeSetFactory::create(1)" creates the AHVPS for only one AHVP, as there is only one attribute to send out i.e., "DataFromMem". The following example also shows that the AHVPS is only created if the attribute "DataFromMem" is changed. Section 6.2.2.2.1 further explains how this AHVPS is received and the attributes are extracted by a federate.

```
// Setting up the data structure required to send this object's state to the RTI.
RTI::AttributeHandleValuePairSet* Memory::CreateNVPSet()
{
  RTI::AttributeHandleValuePairSet* pMemoryAttributes = NULL;
  // Make sure the RTI Ambassador is set.
  if ( ms_rtiAmb )
  {
      //-------------------------------------------------------
      // Set up the data structure required to send this object's state to the RTI.
      //-------------------------------------------------------
      pMemoryAttributes = RTI::AttributeSetFactory::create( 1 );
      if ( hasDataFromMemChanged == RTI::RTI_TRUE )
      {
        pMemoryAttributes -> add( this -> GetDataFromMemRtiId(),(char*) &this -> GetDataFromMem(),
                                                                    (sizeof(int)) );
        out2file << "Data From Memory = " << DataFromMem << endl;
      }
  }
  return pMemoryAttributes;
}
```

Figure 13: Creating an AHVPS for communication among Module/Federates.

6.2.1.5 Time elapse using RTI

The software code, in Figure 14, shows that the module has sent a time elapse request to the RTI according to the time calculated (represented by the “requestTime”) for the next event. The lookahead value (see step 6 Figure 9) is set to 0.5, means that the federate will not send any output until this period is elapsed. The time step value is set to 1.0, means that this is the maximum limit for the time elapse requested to the RTI. The final “requestTime” is calculated by adding the time step value of 1.0 and the “grantTime”, which is the time granted in the previous time elapse request. In other words the “grantTime” is the current simulation time. The RTI method “nextEventRequest(requestTime)” requests for the time elapse until the “requestTime”. In return, the RTI grants the time advance until the next event. The tick() function gives control to the RTI for processing the ongoing events/tasks.

```
//--------------------Setting the lookahead time
        lookahead = RTIfedTime(0.5);
        //--------------------
        RTIfedTime requestTime(1.0); // requestTime = timeStep = 1.0
        requestTime += grantTime; // grantTime is the time granted by the RTI during the last time elapse
        out2file << "\n\nRequest time = " << requestTime << endl;
        timeAdvGrant = RTI::RTI_FALSE;
        rtiAmb.nextEventRequest( requestTime );
  while( timeAdvGrant != RTI::RTI_TRUE )
  {
        //---------------------------------------------------------
        // Tick will turn control over to the RTI so that it can process an event.
        //---------------------------------------------------------
        rtiAmb.tick();
  }
```

Figure 14: Time elapse requested by a Federate using the RTI service methods.

6.2.1.6 Logging simulator activity

The logging capability is developed for all the modules of the simulator. Each module has its own logging file. Whenever a module receives or updates an attribute it sends this information to a file to maintain the logs. The status of the internal registers (e.g., processor module) and the memory dumps are also stored in the file. All this information is logged with the simulator's time stamp. These logs are explained in chapter 7 and the message flow and timing information is extracted and analyzed. The "out2file" statement in Figure 14 is sending data to a log file.

6.2.1.7 FOM/SOM

This section identifies the time management parameters and the interaction attributes used by each federate. This information is a part of a FED file (discussed in section 2.3) that provides an interface between the DEVS model and the RTI layer. Figure 15 shows the names of the attributes used in this case study. These attributes are the input output ports of the DEVS model of each component in the case study. As an example take the memory and the processor modules/federates. The processor's relevant published attributes (e.g., Address_Processor) are included in the subscribed attributes of the memory federate and vice versa.

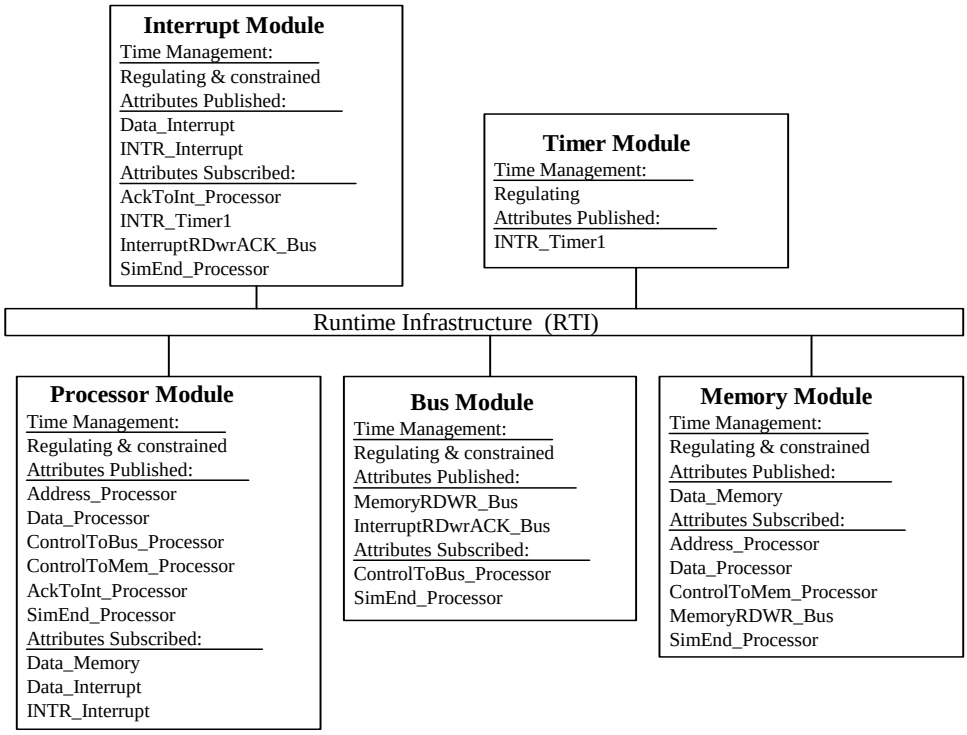


Figure 15: Block diagram containing time management and attribute information.

6.2.2 Memory Module

In this section the assumptions made for the memory module, development of the module using the DEVS formalism along with the excerpts from the software code and the software flow chart are discussed.

6.2.2.1 Assumptions

The Memory size is assumed to be 144 Bytes as it fulfils the requirement of the case study. It is assumed that this memory module does not introduce wait states. The processor has divided the memory module into logical segments as shown in Figure 16. Each number in this figure is a decimal value. An Extra segment is not implemented, as there is no string operation involved in this case study.

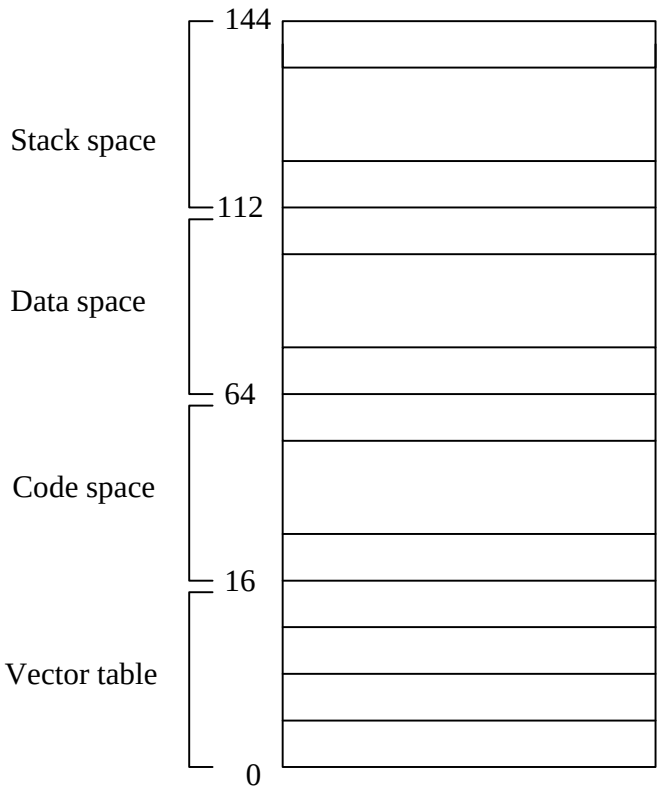


Figure 16: Memory map for 8-bit structure (specific to the case study)

6.2.2.2 Development of Memory module

This section discusses the implementation of the memory module on the basis of the DEVS formalism. It explains the software development of the module's external transition function, duration function, output function, internal transition function and confluent transition function. For further details, the complete software code of this module along with the whole simulator is referred in Appendix A.2.

6.2.2.2.1 External transition function (δ_{ext})

During the δ_{ext} function, the control signal (read/write) from the processor or bus controller module arrives along with the address on the address bus. If the control signal

is for a memory write operation, the data on the data bus is also required from the processor. The software implementation of the δ_{ext} function is implemented as shown in Figure 17. The figure is further divided into sections to facilitate discussion.

As an example, following is a detailed discussion of the memory read operation. To initiate the memory read cycle, the federate ambassador is informed through the RTI that the "ControlSignalFromProToMem" and the "AddressFromPro" attributes have arrived through the input ports of the memory module (as discussed in section 6.2.1.6), each in the form of AHVPS. In order to update the attributes of the memory module, the federate ambassador then calls the "Update(const RTI::AttributeHandleValuePairSet&)" method, shown in Figure 17, section A. Due to the two incoming attributes, the value returned by the "theAttributes.size()" equals to 2 and the "for" loop is executed twice. The actual values of the incoming attributes are extracted and saved using the getHandle() and getValue() methods. The "Out2file" saves these attributes to a log file. The value of the "ControlSignalFromProToMem" attribute tells whether this is a memory read or memory write operation. Assuming this value is for a memory read operation, the "DataFromMem" variable is updated as shown in Figure 17, section B. A flag "hasDataFromMemChanged" is also set so that when UpdateFromMem(FederateTime) method (Figure 17, section D) is called this updated value is sent out to the RTI. Figure 17, section D shows that the state of the memory is updated in terms of time and the new AHVPS are created for the variables that are changed (e.g., "DataFromMem"). CreateNVPSet() method creates the AHVPS as discussed in section 6.2.1.4. These AHVPS are sent out to the RTI during the output function as discussed in section 6.2.2.2.3.

---------- SECTION A ----------

```
void Memory::Update( const RTI::AttributeHandleValuePairSet& theAttributes )
{
  RTI::AttributeHandle attrHandle;
  RTI::Ulong           valueLength;
  // We need to iterate through the AttributeHandleValuePairSet to extract each AttributeHandleValuePair.
  // Based on the type specified ( the value returned by getHandle() ) we need to extract the data from the
  // buffer that is returned by getValue().
  for ( unsigned int i = 0; i < theAttributes.size(); i++ )
  {
     attrHandle = theAttributes.getHandle( i );
     if ( attrHandle == Memory::GetAddressFromProRtiId() )
     {
        int AddressFromPro;
        theAttributes.getValue( i, (char*)&AddressFromPro, valueLength );
       out2file << "Address from Processor = " << AddressFromPro << endl;
     }
    if ( attrHandle == Memory::GetDataFromProRtiId() )
    {
        int DataFromPro;
        theAttributes.getValue( i, (char*)&DataFromPro, valueLength );
       out2file << "Data from Processor = " << DataFromPro << endl;
    }
    if ( attrHandle == Memory::GetControlSignalFromProToMemRtiId() )
    {
        int ControlSignalFromProToMem;
        theAttributes.getValue( i, (char*)&ControlSignalFromProToMem, valueLength );
       out2file << "Control Signal From Pro To Mem = " << ControlSignalFromProToMem << endl;
     }
     if ( attrHandle == Memory::GetSimEndFromProRtiId() )
    {
        int SimEndFromPro;
        theAttributes.getValue( i, (char*)&SimEndFromPro, valueLength );
       out2file << "SimEndFromPro = " << SimEndFromPro << endl;
     }
     if ( attrHandle == Memory::GetMemRDWRSigFromBusRtiId() )
    {
        int MemRDWRSigFromBus;
        theAttributes.getValue( i, (char*)&MemRDWRSigFromBus, valueLength );
        out2file << "RD WR signal fro Bus (RD=1 WR=2) = " << MemRDWRSigFromBus << endl;
     }
  }
}
```

---------- SECTION B ----------

```
// Reading from the Memory
// MemoryMap[ ] is an array of 150 bytes
myMemory -> DataFromMem = MemoryMap[myMemory -> AddressFromPro];
  // Set flag so that when UpdateFromMem( FederateTime ) is called we send this new value to the RTI.
hasDataFromMemChanged = RTI::RTI_TRUE;
```

---------- SECTION C ----------

```
// Writing to the Memory
MemoryMap[myMemory -> AddressFromPro] = myMemory -> DataFromPro;
```

```
                    ---------- SECTION D ----------
// Updating the state of the Memory
void Memory::UpdateFromMem( RTI::FedTime& newTime )
{
  // Set last time to new time
  this -> SetLastTime( newTime );
    //---------------------------------------------------------
    // Updating the state of memory
    //---------------------------------------------------------
      // In order to send the values of our attributes, we must construct an AttributeHandleValuePairSet
      // (AHVPS) which is a set comprised of attribute handles, values, and the size of the values.
      // CreateNVPSet() is a method defined on the Memory class - it is not part of the RTI. Look inside
      // the method to see how to construct an AHVPS
    RTI::AttributeHandleValuePairSet* pNvpSet = this -> CreateNVPSet();
}
```

Figure 17: Implementation of the external transition function in the simulation code.

6.2.2.2.2 Duration function (ta)

The ta function is used to elapse time until the arrival of the next event or the lookahead time which ever comes first. After the time is elapsed, the module executes the output function discussed in section 6.2.2.2.3 and sends out the AHVPS through the RTI. Figure 18 shows that if the module gets the time advance grant from the RTI, it gives control to the RTI by using tick() method. If there is no incoming event, the RTI waits until the elapse of lookahead period (the ta) and carried out the output function.

```
  while( timeAdvGrant != RTI::RTI_TRUE )
  {
        //---------------------------------------------------------
        // Tick will turn control over to the RTI so that it can process an event.
        //---------------------------------------------------------
        rtiAmb.tick();
  }
```

Figure 18: Implementation of the duration function in the simulation code.

6.2.2.2.3 Output function (λ)

The λ function sends out the module's output to the federation. Once the AHVPS are constructed, as discussed in section 6.2.1.4, a method (updateAttributeValues, see Figure 19) is called and the AHVPS along with the module's ID and time information is sent out to the federation. This method is a part of RTI ambassador code.

```
// Send the AHVPS to the federation.
//-----------------------------------------------
// this call sends out NVPSet(), at the current simulation time + loohahead.
//-----------------------------------------------
        (void) ms_rtiAmb->updateAttributeValues( this->GetInstanceId(), *pNvpSet,
                                        this->GetLastTimePlusLookahead(), NULL );
```

Figure 19: Implementation of the output function in the simulation code.

6.2.2.2.4 Internal transition function (δ_{int})

During a memory read operation the δ_{int} function changes the module's state from the read to the wait (see

Figure 10). The module stays there until it receives the δ_{ext} function. During the execution of the tick() method, shown in Figure 18, the δ_{int} function is realized once the λ function is carried out.

6.2.2.2.5 Confluent transition function (δ_{con})

If there is a tie-break between the δ_{int} function and the δ_{ext} function, the module gives priority to the δ_{int} function. For a memory read example, after executing the δ_{int} function the memory module should go to a wait state until the δ_{ext} function arrives. But in this

scenario the δ_{ext} function has already arrived so instead of going to the wait state, the module starts executing the δ_{ext} function.

6.2.2.3 Interaction between Memory module and the RTI

In this section the interaction between the memory module and the RTI is discussed and, as a future research work, this information is useful to develop a tool that could help in modeling hardware platform components. Each input output port of the DEVS memory model has an attribute associated with it and can be defined as the published (i.e., an output port) and/or subscribed (i.e., an input port). This port information for each federate is redefined in the FED file and the RTI uses this information to map the ports of different federates. During the initialization stage of the memory federate, time management is enabled/initialized and the module is registered as a federate to the RTI (see step 2 Figure 9). The δ_{ext} function in the memory module is triggered by the RTI as soon as it receives the updated value(s) of the subscribed attribute(s) on the input port(s). Section 6.2.2.2.1 discusses in detail the use of the RTI built-in methods by the memory module's δ_{ext} function during the attribute(s) extraction from incoming AHVPS and the creation of the new AHVPS to be sent out. For time elapse during the ta and the δ_{int} function, the RTI services are used by the memory module and the definable parameters are the lookahead time and the time step. During the λ function, the memory module calls an RTI method "updateAttributeValues" and sends out the AHVPS values created during the δ_{ext} function.

6.2.2.4 Software Flow diagram

After joining the federation, this module waits for all its attributes to be subscribed by other federations. Figure 20 shows a flow chart of this module.

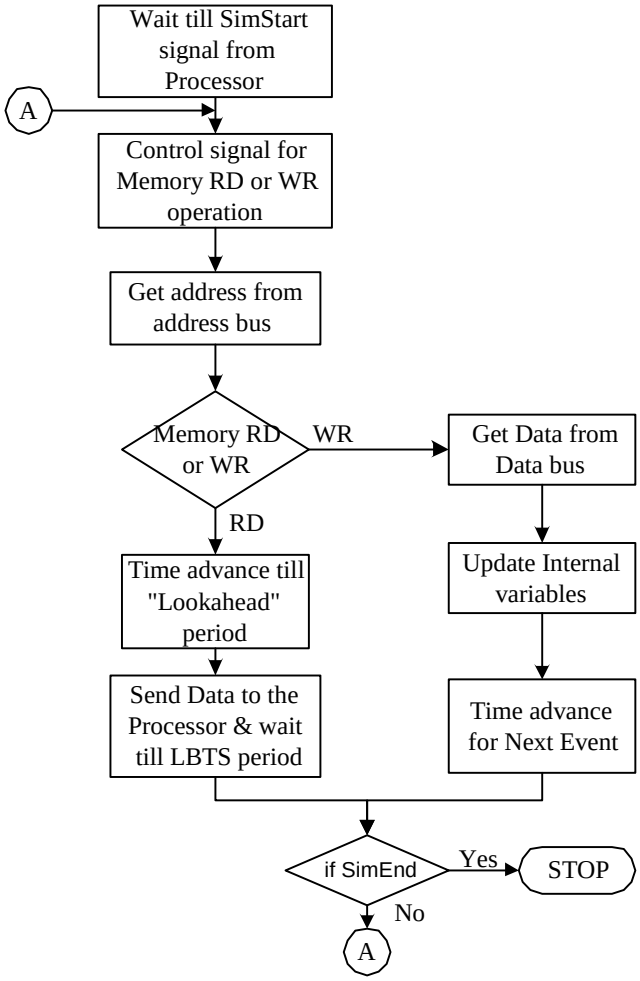


Figure 20: Flow Chart of the memory module

6.2.3 Processor Module

In this section the assumptions made for the processor module, the level of timing detail implemented, the resolution of the interrupt request signal when the interrupt flag is disabled and the software flow chart are discussed. The concept for the development of

the module using the DEVS formalism is the same as discussed for memory module in section 6.2.2.2. For further details, the software code of this module along with the whole simulator is referred in Appendix A.2.

6.2.3.1 Assumption

For this case study, the feature/functionality implemented in the CU, EU and BIU is a subset of the features discussed in section 5.1.2.2. The detailed features/functionality of the CU, EU and BIU is left for a future work. A brief discussion of the CU, EU and BIU implemented is as follows.

The CU checks interrupt signals from Interrupt controller at the end of each op-code execution. The EU reads and decodes the op-code, updates the instruction queue and interacts with the CU and BIU as required during the execution of the op-code. The BIU performs RD/WR operations, sends control signals to the bus and interrupt controller, fetches opcode from memory and maintains the status of the instruction queue. Since the Intel 8088 micro-processor has an 8-bit data bus, the BIU automatically executes two read or write cycles for each 16-bit operand. The least significant byte of the word is stored in the lower valued address location of the Memory and the most significant byte in the next higher address location.

6.2.3.2 Internal Registers of the processor module

The following registers are implemented and initialized in the processor module for this case study. The Code Segment (CS) register, the Data segment (DS) register and the Stack Segment (SS) register are initialized according to Figure 16. The values assigned are CS=1, DS=4 and SS=7. The Instruction Pointer (IP) register and Stack Pointer (SP)

register are also implemented with the initial values of IP=0 and SP=31. The SP register is pointing at the top of the empty stack. The Internal Flag (IF) register defined with the consideration of only the interrupt flag and the zero flag. During initialization the interrupt flag bit is enabled (ready for an interrupt) and the zero flag bit is set to 0. The Data Index (DI), AX, BX and CX registers are also implemented and given an initial value of zero.

The physical address to fetch the opcode is calculated as ((CS×16)+IP). The address calculation for the direct addressing mode (e.g., MOV [BX],CH instruction) is implemented as ((DS×16)+BX). The address calculation for PUSH and POP operations is implemented as ((SS×16)+SP).

6.2.3.3 Resolution of the interrupt request (INTR) signals

If the processor module is already executing an interrupt request, the interrupt flag will be disabled and the processor will not acknowledge any new interrupt requests unless the execution of the current interrupt routine has finished and the interrupt flag is enabled. In the scenario where the interrupt flag is disabled, any interrupt request sent by the interrupt controller module will remain pending in the interrupt controller module. During this period, any new interrupt generated by the timer module will be lost as there is no buffer implemented in the Interrupt controller module. For future work a buffer or queue could be implemented in the interrupt controller module and a priority mechanism can also be implemented in the interrupt controller module.

6.2.3.4 Software Flow diagram

After joining the federation, this module waits for all its attributes to be subscribed by other federations. Figure 21 shows a flow chart of this module.

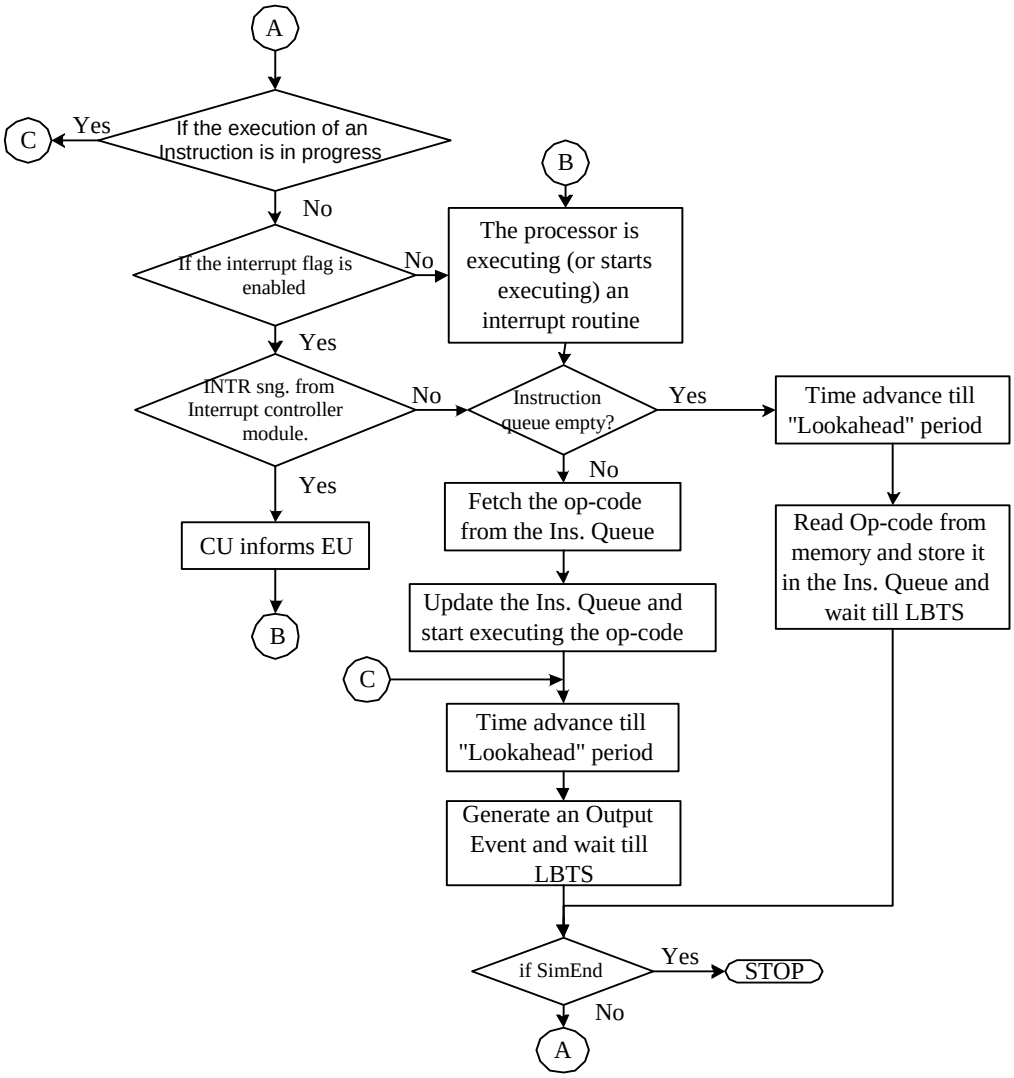


Figure 21: Flow chart of the processor module

6.2.4 Bus Controller Module

In this section the assumptions made for the bus controller module and the software flow chart are discussed. The concept for the development of the module using the DEVS

formalism is the same as discussed for the memory module in section 6.2.2.2. For further details, the software code of this module along with the whole simulator is referred in Appendix A.2.

6.2.4.1 Assumption

The bus controller gets the status signals S_0 S_1 S_2 (as discussed in section 5.1.2.2.3) from the processor module, decodes them and generates appropriate bus cycles/signals to the peripheral devices. The decoding of the status signals is assumed as shown in table 4.

Status Signals S_0 S_1 S_2	**Bus cycles/signals generated by the bus controller module**
1	Memory read
2	Memory write
3	Interrupt acknowledge signal to the interrupt controller module
4	Request to clear interrupt acknowledge signal.

Table 4: Decoding of the status signals (S_0 S_1 S_2) by the bus controller module.

6.2.4.2 Software Flow diagram

After joining the federation, this module waits for all its attributes to be subscribed by other federations. Figure 22 shows a flow chart of this module.

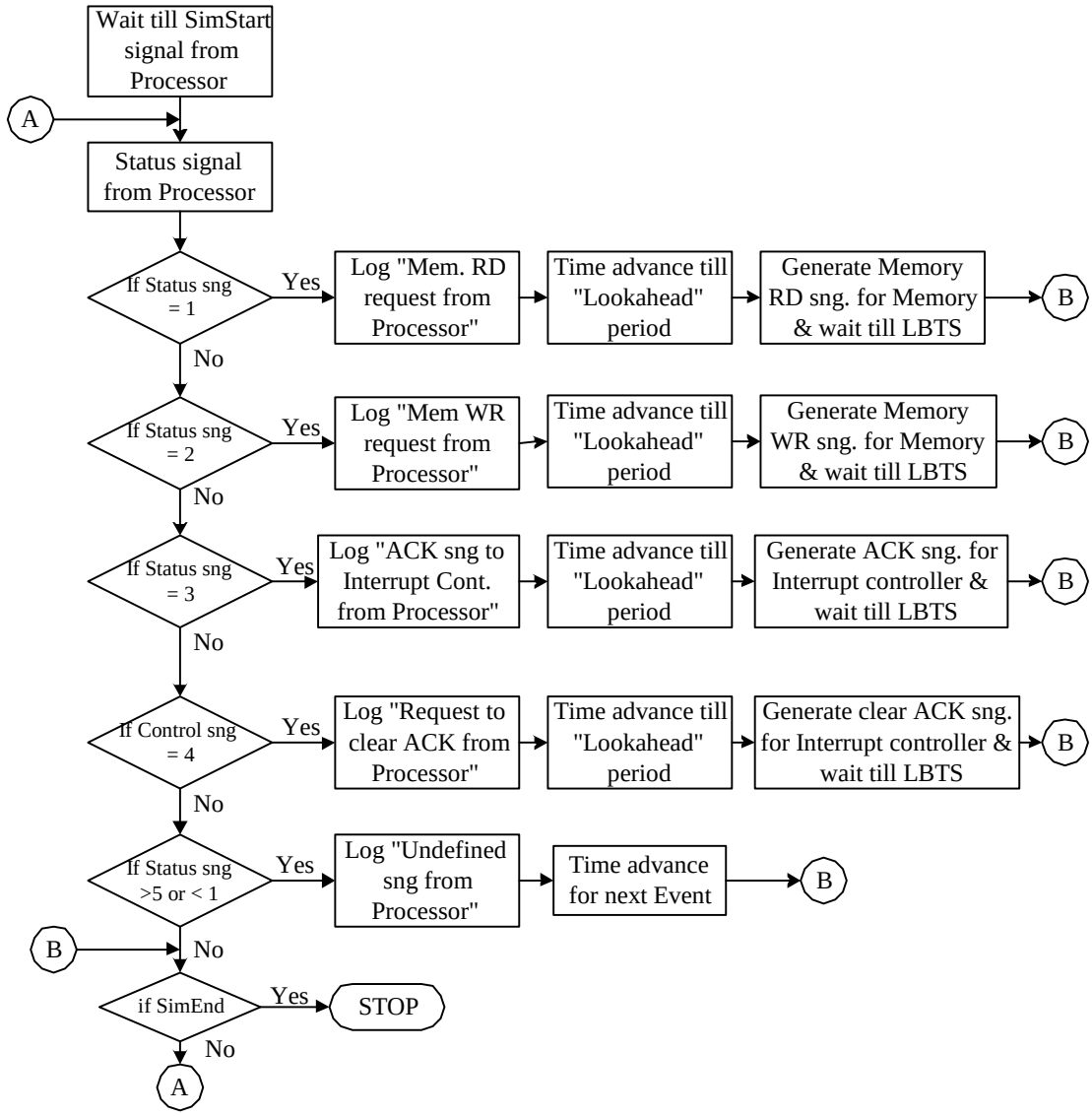


Figure 22: Flow diagram of the bus controller module.

6.2.5 Interrupt Controller Module

In this section the assumptions made for the interrupt controller module and the software flow chart are discussed. The concept for the development of the module using the DEVS formalism is the same as discussed for the memory module in section 6.2.2.2. For further

details, the software code of this module along with the whole simulator is referred in Appendix A.2.

6.2.5.1 Assumption

It is assumed that, being a basic interrupt module, there is no priority-resolving algorithm implemented for the interrupt signals triggered by different peripheral devices. A buffer or queue for the output signal (the interrupt request) is not implemented and the effect of this is discussed in section 6.2.3.3. The interrupt controller module sends out the vector type after getting the interrupt acknowledge signal and it is assumed that the vector type for the timer interrupt is zero. More detailed functionality of the interrupt controller was discussed in section 5.1.2.4 and can be added in future work.

6.2.5.2 Software Flow diagram

After joining the federation, this module waits for all its attributes to be subscribed by other federations. Figure 23 shows a flow chart of this module.

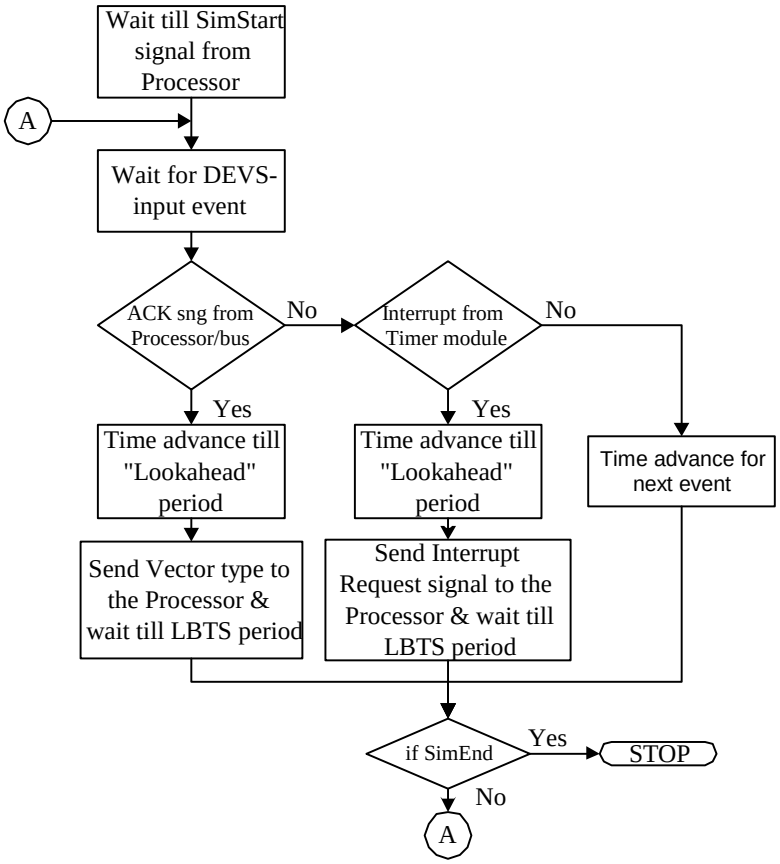


Figure 23: Flow Chart of the interrupt controller module

6.2.6 Timer Module

In this section the assumptions made for the timer module and the software flow chart are discussed. The concept for the development of the module using the DEVS formalism is the same as discussed for the memory module in section 6.2.2.2. For further details, the software code of this module along with the whole simulator is referred in Appendix A.2.

6.2.6.1 Assumption

This module only interacts with the Interrupt controller module. It sends out an INT signal after every 350 time units. The timer module does not wait for the simulation start

signal from the processor. This module starts its function as soon as the interrupt controller joins the federation and subscribes to the interrupt (INT) request signal of the timer module. It generates 3 INT signals and then resigns the federation.

6.2.6.2 Software Flow diagram

Figure 24 shows a flow chart of this module.

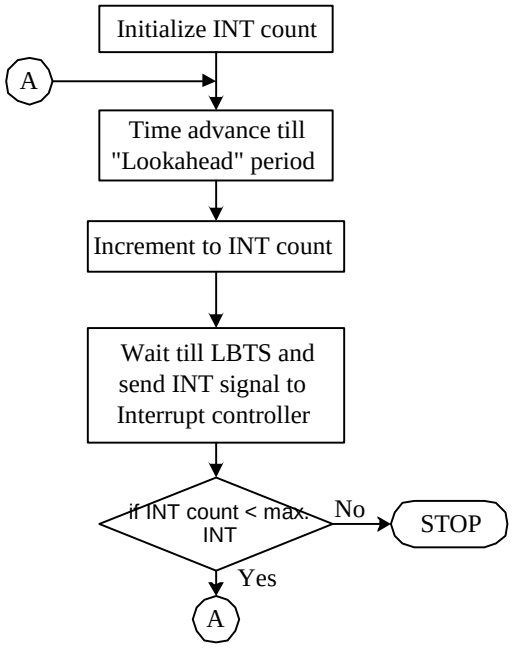


Figure 24: Flow chart of the timer module.

CHAPTER 7

Results

In this chapter the results of the simulator (developed as a case study) are discussed. Each module in the simulator, logs the messages sent out or received from the other modules along with the timing information and the values of appropriate internal state variables. In this chapter, a detailed interpretation of the logs of the processor module and the memory module is shown as a reference and the logs of the other modules can be interpreted in a similar way. Later in the chapter the message event sequences, from each module's logs, are extracted and presented with respect to the simulation time. These message event sequence diagrams are drawn for the minimum and maximum operation modes of the case study processor. At the end of this chapter, an analysis of these results is given, and the acquired timing information for opcode executions is compared with the theoretical values of Table 2 and 3.

This simulator is developed using Visual C++ version 6.0, the DMSO RTI-NG version 1.3, and the Win98se operating system. A standalone computer (Intel Pentium-II processor 400MHz) is used for the simulations and the computer's execution time to run a simulation of 1600 clock cycles is about 4 minutes.

7.1 Module Log Interpretations

The log files for different modules are referred in appendix B. Below is the explanation/interpretation of snap shots taken from the logs of the process and the memory modules. Other modules' logs can be interpreted similarly.

7.1.1 Processor's Log

The following discussion explains each line of Figure 25. Each number in the log is a decimal value.

Line 1 of Figure 25 shows the time advance requested by the processor module to the RTI. In return the RTI calculates the LBTS for the federation and issues a time advance grant to the processor module. This message is shown in line 3. The reason for not granting (line 3) the requested time (line 1) by the RTI is the arrival of "Data From Memory" attribute as shown in line 2. Line 4, 5 and 6 show the contents of the internal registers of the processor and the contents of the instruction queue. Line 6 reflects that "2" has arrived in the processor from the memory module (line 2) and was added to the instruction queue. Line 7 shows the lookahead value for sending out the processor's attribute(s). Line 8 and 9 shows the messages sent out by the processor to other modules at time (29.0 + lookahead) units.

----------**Processor log**----------

```
Line 1 : Request time = 30.0000000000
Line 2 : Data from Memory = 2
Line 3 : FED_HP: Time granted (timeAdvanceGrant) to: 29.0000000000
Line 4 : Zero Flag=0 INT Flag=1 AL=0 DI=0 BX=0 CL=0 CH=0
Line 5 : IP=7 CS=1 SS=7 SP=31 DS=4
Line 6 : Contents of the queue: 177 2
Line 7 : Lookahead = 3.0000000000
Line 8 : ControlSignalFromProToMem = 1
```

```
Line 9 : AddressFromPro = 23
--------------------------------------------------------------------------------------------------------------------------
```

Figure 25: Snap shot of Processor module's log

7.1.2 Memory module's Log

The following discussion explains each line of Figure 26. Line 1 and 4, the request time and time advance grant, are explained in section 7.1.1. Line 2 and 3 show the messages sent to the memory module via the RTI. Time 552.0 is the current time of the federation as shown in line 4. Line 5 shows the lookahead information used by the RTI simulation engine in order to calculate the LBTS for the federation. Lookahead of 1.0 indicates that the memory federate will not send any data until (543.0 + 1.0) units. "Data from memory" as shown on line 5 is actually sent out at (552.0 + 1.0) units. Lines 7 to 15 show the 144 byte contents of memory (16 bytes per line). Section 6.2.2.1 (Figure 16) discusses the logical memory segments along with each segment starting addresses assumed for this case study. Line 7 has 16 memory locations for the vector table of the interrupt vector types. First byte 30 is the least significant byte of the IP register. Next 0 is the most significant byte of the IP and next two locations 1 and 0, show the least and the most significant bytes of the CS. This (IP+(CS×16)) value points to a memory location where the interrupt routine resides. From line 8 and onwards, the code segment starts and it has all the opcodes for the main program, as stated in table 2. The last two bytes of line 9 and line 10 has the opcodes for the interrupt routine, as stated in table 3. From line 11 and onwards, the data segment starts and it has five even numbers generated. The main program actually generates ten even numbers, but due to an interrupt call execution control jumps from the main program to the interrupt routine. Once the interrupt routine

is fully executed, the main program's execution continues until it generates ten even numbers. The first byte on line 13 shows the output of the interrupt routine. Being a simple counter, it increments this memory location on each interrupt call. Line 14 and 15 show the stack segment. Starting from the bottom of the stack, it contains the least and the most significant bytes of the flag register, the least and the most significant bytes of the CS and the least and the most significant bytes of the IP.

----------**Memory log**----------

```
Line 1 : Request time = 555.0000000000
Line 2 : Control Signal From Pro To Mem = 1
Line 3 : Address from Processor = 32
Line 4 : FED_HP: Time granted (timeAdvanceGrant) to: 552.0000000000
Line 5 : LOOKAHEAD = 1.0000000000
Line 6 : Data From Memory = 233
Line 7 : 30 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
Line 8 : 181 0 191 0 0 177 2 187 0 0 3 223 136 47 71 2
Line 9 : 233 131 199 10 117 13 244 0 0 0 0 0 0 0 83 187
Line 10 : 0 32 138 7 254 192 136 7 91 207 0 0 0 0 0 0
Line 11 : 0 2 4 6 8 0 0 0 0 0 0 0 0 0 0 0
Line 12 : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Line 13 : 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Line 14 : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Line 15 : 0 0 0 0 0 0 0 0 0 0 0 10 0 1 2 0
```

Figure 26: Snap shot of Memory module's log

7.2 Message Event Sequence in Minimum Mode

Following results are taken from the log files, referred in appendix B, and presented in the form of message event sequence diagrams. As an example, the message event sequence of memory read (RD), memory write (WR) and interrupt acknowledge cycles are discussed in section 7.2.1 and 7.2.2. The vertical axis of these sequence diagrams shows the time in terms of clock cycles and the horizontal axis shows the interactions between different modules.

7.2.1 Memory RD and Memory WR Operations

Figure 27 shows the level of abstraction in terms of the control signals and the timing information. The implementation of all the control signals and the detailed timing information is outside the scope of this research (chapter 4).

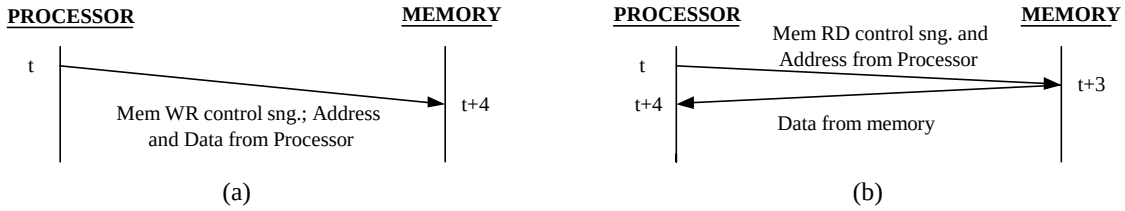


Figure 27: (a) Memory Write (b) Memory Read operation in minimum mode

7.2.2 Interrupt acknowledge Operation

For an interrupt acknowledge (INTA) operation, the processor sends an acknowledge signal and the interrupt controller replies with the vector type. One bus cycle is required for the interrupt acknowledge signal and another for acquiring the vector type. This is shown in Figure 28.

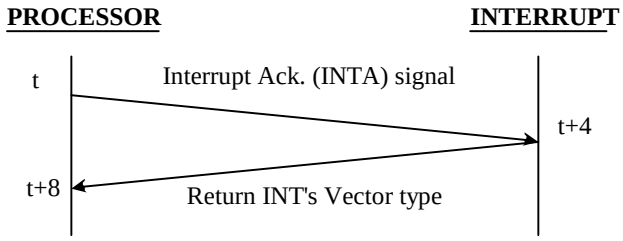


Figure 28: Interrupt acknowledge (INTA) operation in minimum mode

7.3 Message Event Sequence in Maximum Mode

Following results are extracted from the log files, referred in appendix B, and presented in the form of message event sequence diagrams. As an example, the message event sequence of memory read (RD), memory write (WR) and interrupt acknowledge cycles are discussed in section 7.3.1 and 7.3.2.

7.3.1 Memory RD and Memory WR Operations

The processor sends a memory read or memory write request to the bus module, and the bus module generates the appropriate control signals to the memory module. Figure 29 shows the level of abstraction in terms of the control signals and the timing information.

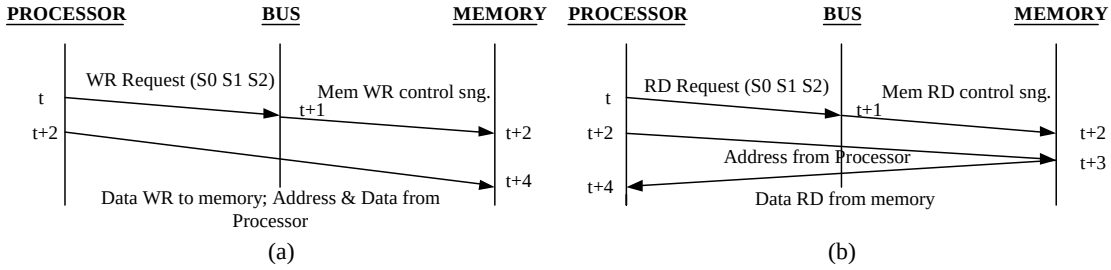


Figure 29: (a) Memory Write (b) Memory Read operation in maximum mode

7.3.2 Interrupt acknowledge Operation

For an interrupt acknowledge (INTA) operation in maximum mode, the processor sends a signal to the bus module to originate the interrupt acknowledge signal to the interrupt controller module. In response, the interrupt controller sends out the vector type to the processor. This operation is shown in Figure 30.

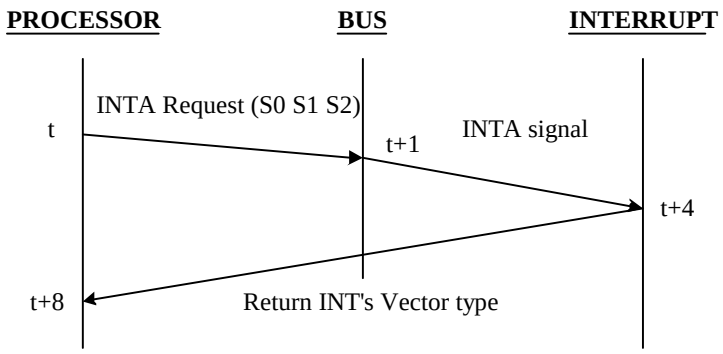


Figure 30: Interrupt acknowledge (INTA) operation in maximum mode

7.4 Sequence diagram for Interrupt Call Operation

The sequence diagram, Figure 31, explains the interactions between different modules in terms of bus cycles during an interrupt call. The time x represents the total time elapsed between the generation of the interrupt signal (at time t) and the execution of the interrupt return (IRET) statement.

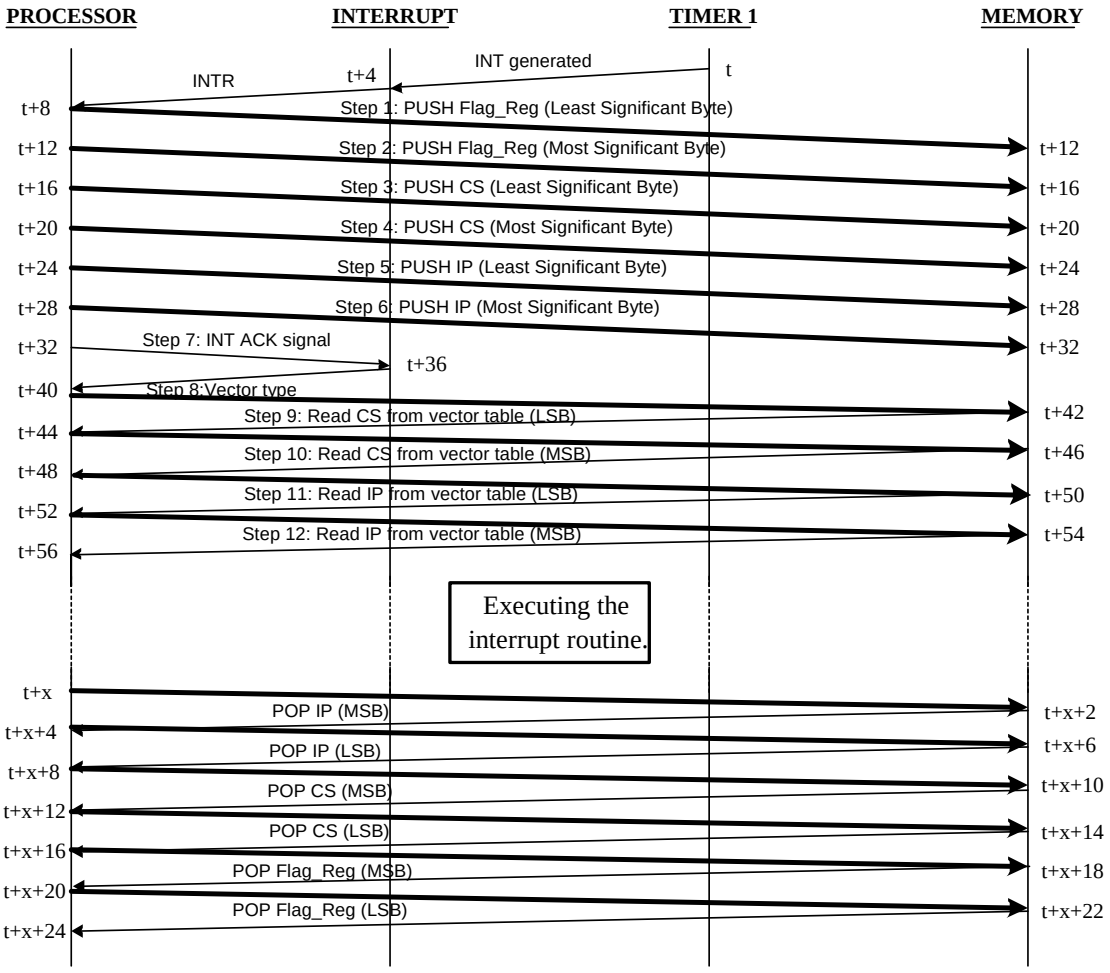


Figure 31: Interaction flow between different modules during Interrupt operation.

7.5 Analysis of the results

As mentioned in chapter 4, the detailed bus level signals are not the scope of this research. But for implementing the interactions between different federates, the time breakdowns within a bus cycle (represented as 4 time units) are required. These time breakdowns (figure 27, 29 & 30) should not be compared with the real timing diagrams.

To analyze the case study's results the timing information gathered from the logs of different modules can be verified against the theoretical timing values given in Table 2

and 3. The level of accuracy for the execution unit of the processor is up to one clock cycle. Hence the simulation results for the execution timing for each program instruction should match with the theoretical timing values of Table 2 and 3. Here two program instructions (opcodes) from the case study example are analyzed.

MOV [BX],CH

Table 3 shows that it requires 14 clock cycles for the EU and the BIU to execute this opcode. Figure 32 shows a snap shot of the processor module's log, when the MOV [BX],CH opcode is at the top of the Instruction queue. This opcode moves the value of CH register to the memory location calculated as ((DS×16)+BX). Lines 4 and 5 show the values of the BX, CH and DS registers. Hence as an execution result of this opcode the value 2 should be moved to a memory location of 4×16+1=65. Line 3 shows that 138.0 is the execution start time for this opcode.

```
---------------------------Processor log-----------------------------
Request time = 139.0000000000
Data from Memory = 47
FED_HP: Time granted (timeAdvanceGrant) to: 138.0000000000
Zero Flag=0 INT Flag=1 AL=0 DI=1 BX=1 CL=2 CH=2
IP=14 CS=1 SS=7 SP=31 DS=4
Contents of the queue: 136 47
Lookahead = 3.0000000000
ControlSignalFromProToMem = 1
AddressFromPro = 30
---------------------------------------------------------------------
```

Figure 32: Snap shot of the Processor's log for the analysis of the MOV [BX],CH instruction.

Figure 33 shows a snap shot of the memory module's log when the content of the CH register has been moved to the memory location of 65. This is shown in the second byte of line 10. Line 5 shows that 152.0 is the execution end time for this opcode.

```
-----------------------------Memory log------------------------------
Request time = 155.0000000000
```

```
Control Signal From Pro To Mem = 2
Address from Processor = 65
Data from Processor = 2
FED_HP: Time granted (timeAdvanceGrant) to: 152.0000000000
30 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
181 0 191 0 0 177 2 187 0 0 3 223 136 47 71 2
233 131 199 10 117 13 244 0 0 0 0 0 0 0 83 187
0 32 138 7 254 192 136 7 91 207 0 0 0 0 0 0
0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
--------------------------------------------------------------------------
```

Figure 33: Snap shot of the Memory's log for the analysis of the MOV [BX],CH instruction.

The difference between the execution start and end time is 14 clock cycles. This result matches the theoretical clock cycle value (table 2) required for the execution of MOV [BX],CH instruction.

POP BX

Table 3 shows that it requires 12 clock cycles for the EU and the BIU to execute the POP BX opcode. Figure 34 shows a snap shot of the processor module's log, which can be interpreted as discussed in section 7.1.1. Line 5 shows that 91, the opcode of the POP BX, is at the top of the instruction queue and line 2 shows that the simulation time is 515.0 units.

```
---------------------------Processor log-----------------------------
Request time = 515.0000000000
FED_HP: Time granted (timeAdvanceGrant) to: 515.0000000000
Zero Flag=0 INT Flag=0 AL=1 DI=4 BX=32 CL=2 CH=8
IP=42 CS=1 SS=7 SP=23 DS=4
Contents of the queue: 91 207
Lookahead = 3.0000000000
ControlSignalFromProToMem = 1
AddressFromPro = 58
--------------------------------------------------------------------------
```

Figure 34: Snap shot of the Processor's log, showing the start of the execution of the POP BX instruction.

Figure 35 shows a snap shot of the processor log when the contents of the BX register have been retrieved (pulled) from the memory location ((SS×16)+SP). BX is a 16 bit register, so the Intel 8088 processor requires two bus cycles to retrieve the register value. The first bus cycle retrieves the most significant byte and the second bus cycle retrieves the least significant byte. Figure 35 and line 2 shows the transfer of the least significant byte (4), line 4 and 7 show that the value of the BX register has been updated from 32 to 4. Line 3 shows the execution end time (527.0 units) for the POP BX instruction. Line 9 shows that the opcode of POP BX (91) has been removed from the instruction queue after the completion of this instruction.

```
---------------------------Processor log-----------------------------
Request time = 527.0000000000
Data from Memory = 4
FED_HP: Time granted (timeAdvanceGrant) to: 527.0000000000
Zero Flag=0 INT Flag=0 AL=1 DI=4 BX=32 CL=2 CH=8
IP=42 CS=1 SS=7 SP=25 DS=4
Contents of the queue: 91 207
Zero Flag=0 INT Flag=0 AL=1 DI=4 BX=4 CL=2 CH=8
IP=42 CS=1 SS=7 SP=25 DS=4
Contents of the queue: 207
Lookahead = 3.0000000000
ControlSignalFromProToMem = 1
AddressFromPro = 138
-----------------------------------------------------------------------
```

Figure 35: Snap shot of the Processor's log, showing the end of the execution of the POP BX instruction.

The difference of the instruction fetch time and the end execution time is 12 clock cycles. This result matches with the theoretical value (Table 3) and hence the simulator's execution has been verified against the theoretical timing values.

CHAPTER 8

Conclusions

8.1 Conclusions

The following conclusions are drawn from this research.

- Hardware platform components have been modeled by using the DEVS atomic formalism and simulated under the HLA framework.
- Conservative time advance approach has been used in the DEVS atomic model, as this is simple and HLA compliant.
- The level of details shown for each hardware component in chapter 5, addresses the requirements of the hardware designers to simulate their designs before physical implementation. The case study has proved that by implementing these models, accurate results can be gathered within a clock cycle (in section 7.5). For this case study, the implementation of all the features suggested (in chapter 5) for the hardware components were not required, as this study is focused for a specific hardware platform (Intel 8088 based).
- The HLA-specific details that should be visible in a component's model of a hardware platform are discussed in section 6.2.2.3 and concluded as follows.

- The attributes for input and output ports should be published or subscribed to the RTI and defined in the HLA compliant FOM/SOM.
- In the start of the simulation, a component should be able to register itself with the RTI so that it becomes a part of the federation. At the end it should be able to resign.
- A component's model should be able to react on the RTI's call for the δ_{ext} function. It should be able to schedule the ta function and any wait states through the RTI. For the λ function, the component's model should be able to create and send out the AHVPS using RTI methods.

8.2 Summary of Contributions

The list of contributions is as follows.

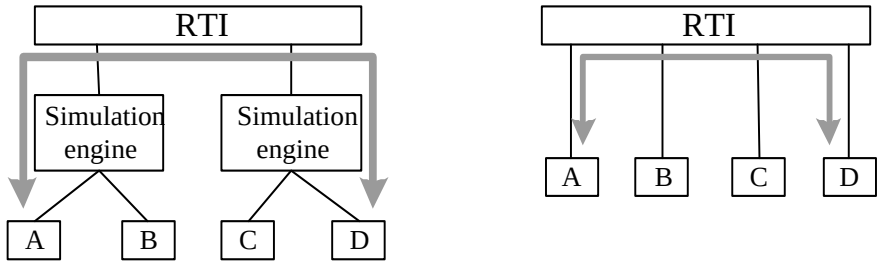
- The DEVS/HLA approach is developed to model and simulate computer hardware platforms.
- An approach is developed towards flattening the hierarchy when the systems are modeled in the DEVS formalism and worked under the HLA framework (section 5.1.2).
- An HLA framework is defined for the computer systems to be simulated (section 5.1.1).
- As a case study, a simple hardware platform (Intel 8088 based) is modeled and a simulator is developed. These hardware components' models can be further developed and generalized (as suggested in chapter 5) to build this simulator as a simulation tool for computer systems designers.

- With the help of this case study, the mapping of the hardware platforms with the HLA has been demonstrated.
- Recommendations for future work have been made as shown in section 8.3.

8.3 Future Research

Following are the recommendations for future work along with some suggestion.

- A software tool (as discussed in section 6.2.2.3), based on the DEVS formalism and Atomic models, can be developed. This tool should be able to send/receive the transition functions directly to/from the RTI and the input and output events should be defined in the HLA compliant FOM/SOM.
- The simulator developed in this research and used for the case study can be further developed for general hardware platform models and the level of details in terms of timing and bus level signals can also be implemented. This development could be a step in the direction of abstraction to reality.
- Analyzing the performance for hierarchical models (the DEVS-Coupled) versus flat models (the DEVS-Atomic), using the HLA framework. This analysis will be useful for future simulator development. The processing delay for the interaction of components A and D (as shown in Figure 36) could be one of the parameters of interest.



(a) (b)

Figure 36: (a) Hierarchical models using the HLA framework (b) Flat models using the HLA framework

- A case study could be done for the combination of systems developed using flat (the DEVS-atomic) approach and hierarchy (the DEVS-coupled) approach, using the HLA framework. Identification of all the issues and implementation requirements could be a useful contribution for future simulator developments. A block diagram is shown in Figure 37.

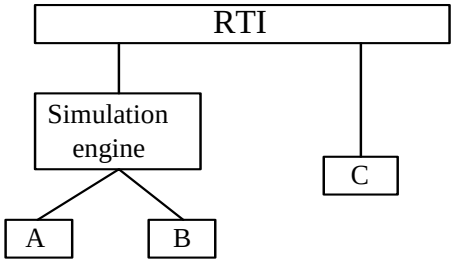


Figure 37: Combination of Flat and Hierarchical models using the HLA framework

References

[1] Dahmann, J.S. “High Level Architecture for simulation” Distributed Interactive Simulation and Real Time Applications, 1997., First International Workshop on , 1997. Page(s): 9 -14

[2] Zeigler, B.P., et al. “Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions”, Simulation Interoperability Workshop, March 1999. Orlando,FL.

[3] Banks, Jerry, Carson, John S., Nelson, Barry L. “Discrete-Event System Simulation”. Second Edition, Prentice-Hall International Series, 1996 ISBN 0-13-217449-9

[4] Wainer, G “CD++: A toolkit to develop DEVS models” Online report, Systems and computer engineering department, Carleton University. URL: http://www.sce.carleton.ca/faculty/wainer/wbgraf/index.html

[5] IEEE standard for modeling and simulation (M&S;) high level architecture (HLA) - Framework and Rules IEEE Std. 1516-2000 , Sep. 2000; Page(s): i -22

[6] IEEE standard for modeling and simulation (M&S;) high level architecture (HLA) - Federate Interface Specification IEEE Std 1516.1-2000 , 2001; Page(s): i – 467

[7] IEEE standard for modeling and simulation (M&S;) high level architecture (HLA)-object model template (OMT) specification IEEE Std 1516.2-2000 , 2001; Page(s): i -130

[8] High-level Architecture / Runtime Infrastructure
RTI 1.3-Next Generation, Programmer's Guide Version 3.2; 7 September 2000
www.dmso.mil

[9] Zeigler, B., Cho, H.; Lee, J.; Sarjoughian, H. "The DEVS/HLA Distributed Simulation Environment And Its Support for Predictive Filtering". DARPA Contract N6133997K-0007: ECE Dept., UA, Tucson, AZ. 1998.

[10] Zeigler, B. P., Kim, T. G., and Praehofer, H. "Theory of Modelling and Simulation", Academic Press, 2nd Edition, 2000.

[11] Rodriguez, D.; Wainer, G. "New Extensions to the CD++ tool". In Proceedings of SCS Summer Multiconference on Computer Simulation. 1999.

[12] Kim, Tag Gon "DEVS Research at KAIST CORE LAB: From Theory to Practice" Computer engineering (CORE) Lab. Department of Electrical Engineering, KAIST, Korea. July 3, 1995
Online report URL: http://sim.kaist.ac.kr/~tkim/devsim.html

[13] Intel iAPX 88 Book July 1981.

[14] Triebel A. Walter and Singh, Avtar "16-bit Micro-Processors. Architecture, Software and Interface Techniques."; Prentice-Hall, Inc, Englewood Cliffs, NJ.
ISBN 0-13-811407-2 01

APPENDIX A

Simulator's Directory Structure & Software Code in C++

This section describes the procedure to configure the simulator (developed for the case study) on the computer and provides the software code, of different modules, required to run the simulator. The preliminary step during the configuration process is to get registered and download the HLA/RTI software from the DMSO Internet site (http://www.dmso.mil). Then run the Helloworld example on the computer to verify the computer's configuration and the correct installation of the HLA software. Section A.1 describes the directory structure for the simulator and section A.2 gives a list of files attached with this document along with some basic definitions.

For this research work, the RTI version 1.3 (Operating System: Win98se; Compiler: VC++6.0) is used.

A.1 Directory Structure

The directory structure of different modules is shown in Figure 38. This is one of many possible structures/configurations. All the modules i.e., Processor, Bus, Interrupt and Timer1 have the same directory structure as Memory module.

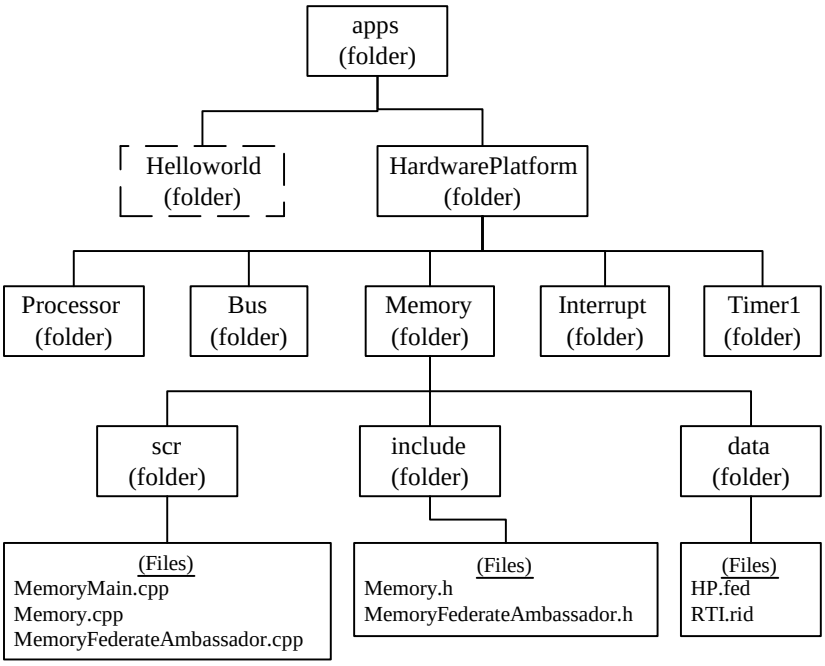


Figure 38: Simulator's directory structure. (This is one of many possible configurations).

A.2 Software Code in C++

Following is the list of files, provided in a floppy disk, attached with this document.

- ProcessorMain.cpp: It contains the main flow of Processor module.
- Processor.cpp: It contains the body of Processor class. Code of all the functions, defined in Processor.hh, is present here.
- ProcessorFederateAmbassador.cpp: Functions of ProcessorFederateAmabassador.hh are defined here. It contains a lot of error/exception handling and appropriate code for every function
- Processor.hh: It defines the functions, variables and constants of Processor class.
- ProcessorFederateAmbassador.hh: It defines ProcessorFederateAmbassador as a derived class of abstract FederateAmbassador class to implement methods so that RTI can call functions in the federate.

- MemoryMain.cpp
- Memory.cpp
- MemoryFederateAmbassador.cpp
- Memory.hh
- MemoryFederateAmbassador.hh
- BusMain.cpp
- Bus.cpp
- BusFederateAmbassador.cpp
- Bus.hh
- BusFederateAmbassador.hh
- InterruptMain.cpp
- Interrupt.cpp
- InterruptFederateAmbassador.cpp
- Interrupt.hh
- InterruptFederateAmbassador.hh
- Timer1Main.cpp
- Timer1.cpp
- Timer1FederateAmbassador.cpp
- Timer1.hh
- Timer1FederateAmbassador.hh
- HP.fed: HP (Hardware Platform) provides an interface between the DEVS model and RTI software. This HP.fed file is developed by using OMDT tool. This tool and its manual can be obtained, along with the HLA/RTI software, from www.dmso.mil.

APPENDIX B

Log Files for each Federate

This section contains a list of log files, provided in a floppy disk, attached with this document. The list contains minimum mode and maximum mode log files.

- Processorlog_min.txt
- Memorylog_min.txt
- Interruptlog_min.txt
- Timer1log_min.txt
- Processorlog_max.txt
- Buslog_max.txt
- Memorylog_max.txt
- Interruptlog_max.txt
- Timer1log_max.txt