USING THE ALFA-1 SIMULATED PROCESSOR WITH EDUCATIONAL PURPOSES

Abstract

Alfa-1 is a simulated computer designed to be used in Computer Architecture and Organization courses. The DEVS formalism was used to attack the complexity of the design, allowing the definition of individual components that were later integrated into a modelling hierarchy. The goal of the toolkit is to allow the students acquiring some practice building hardware components. Here, we present the design and implementation of the tools, focusing in how to use them and how to extend the existing components. We also explain how to assemble and link applications, how to execute them, and how to test new extensions based on a set of testing tools.

1. INTRODUCTION

Computer Organization courses are usually based on the analysis of the behavior of a computer, as described in the bibliography of the area (for instance, [HP94, Pat95, Sta96, HJ97, Tan99]). In these books, Computer Architecture concepts are usually covered theoretically, leaving the students with incomplete and sometime erroneous views of how a computer works. The complexity of these systems is so high that they are organized in a layered structure, using different formalisms to describe different abstraction levels. These layered descriptions allow providing higher insight when analyzing a given subsystem, including assembly language, instruction sets, microprogramming and digital logic. This organization contributes to loose comprehension of system operation as a whole. Likewise, the detailed behavior of the subsystems and their interaction are complex to be attacked. The introduction of higher levels (Programming Languages, Operating Systems) makes the task even more complex.

As a result, advanced courses in the area (Operating Systems, Embedded Systems, Computer Architectures, etc.) face problems derived from the learning process. The main issue making the underlying complexity not to be fully understood is the lack of practical experience. In most cases, some practice is acquired at the level of the instruction set programming, but the inter-level interactions are not fully understood. In addition, it is impossible to cover the behavior for each of the subsystems in detail. The existence of hardware or software tools with educational purposes in this area is reduced, making difficult to make the concepts clear through practice.

We decided to attack these problems based on the use of simulation tools. At present there are several simulation tools devoted to analyze architectural properties (for instance, [SFL92, BAB96, RBDH97, ZB97, ER98, BGW00]). Most of them are mainly devoted to study performance, defining the main building blocks of architecture and their interaction. None is devoted to meet

educational purposes; therefore, they are complex to be used in early stages of Informatics careers, where Computer Organization courses are usually taught. These and other issues were presented in [Self1], where the pros and cons for each set of tools were discussed. Besides these problems, many of these tools are not public domain, making difficult their use in massive computer organization courses. In many cases, they even require to know specialized programming tools, which introduces extra complexity to the pedagogical goals we face.

Therefore, we decided to build from scratch a set of simulation tools to be applied in the area. The idea is to allow anyone with basic knowledge of programming techniques to be able to use them. They are public domain, and have been built using public domain software. They were built to be fully understood by people who have taken an introductory course in computer programming.

The hierarchical and discrete event nature of the problem to study made DEVS [ZPK00] a good choice to ease the development task. DEVS provides a systems theoretic framework to describe discrete event systems as a composite of submodels that can be simulated by abstract entities. Every model can be integrated into a hierarchy, allowing reusing tested models. The modelers have to focus in the problems to solve, as the abstract simulators will be in charge of executing them. In this way, the security of the simulations is enhanced and productivity is improved. Each model can be behavioral (atomic) or structural (coupled). A DEVS atomic model is described by:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >.$$

X: input events set;

S: state set;

Y: output events set;

δ_{int}: $S \rightarrow S$, internal transition function;

δ_{ext}: $Q \times X \rightarrow S$, external transition function, with $Q = \{ (s, e) / s \in S, \text{ and } e \in [0, D(s)]\}$;

λ: $S \rightarrow Y$, output function; and

D: $S \rightarrow \boldsymbol{R}_0^+$, elapsed time function.

The basic idea is that each model uses input/output ports in the interface (defined by the X and Y sets) to communicate with other models. The input external events are received in input ports, activating an external transition function. Every state has an associated lifetime, after which, the internal transition function is activated. These internal events also produce state changes, whose results are transmitted through the output ports. The output function is in charge of doing so, and it is activated before the execution of the internal transition function.

DEVS atomic models can be used to build coupled models. It was proven that DEVS coupled models are semantically equivalent to atomic models, therefore, different modelling hierarchies can be built maintaining the original semantics. Atomic or coupled models can be reused by including them in a modelling hierarchy, as the models are closed under coupling. A DEVS coupled model is defined by:

$$CM = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\} >$$

X is the set of input events;

Y is the set of output events;

D is an index of components, and $\forall i \in D$,

$\mathbf{M_i}$ is a basic DEVS model;

$\mathbf{I_i}$ are the influencees of model i, and $\forall j \in I_i$

$\mathbf{Z_{ij}}$: $Y_i \rightarrow X_j$ is the i to j translation function.

Each coupled model consists of a set of basic models (atomic or coupled) connected through the input/output ports of the interfaces. Each component is identified by an index number. The other models where output values must be sent are called the **influencees** of each model . The translation function uses an index of influencees created for each model (I_i). This function defines which outputs of model M_i are connected to inputs in model M_j. When two submodels have simultaneous events, the *select* function prescribes which of them should be activated first.

There are different tools implementing the theoretical concepts defined by the DEVS formalism (DEVSJava, DevsSim++, DEVS-C++, JDEVS, DEVS-Scheme, etc.). We decided to use the CD++ tool [WT01], which is public domain and fits our needs. Atomic models can be programmed in C++, entitling users with basic programming skills to be able to develop new models. Coupled models are defined using a built-in specification language.

The results of this project were presented in [Self1]. Here, we explain how the tool can be used and modified. The project was successful, as undergraduate students developed every component. The definition of the formal models was done by 3rd year students of a Discrete Event Simulation course, and these formal specifications were used by students in the 2nd year Computer Organization course to build all the models composing the simulated computer. The students have taken a previous course on computer programming in C++, and have background in Mathematics. Undergraduate Teaching Assistants were in charge to develop the final integration models.

In the following sections we present some of the results obtained. We first include a brief description of the underlying architecture and comment how it was defined as a set of DEVS models. Then, we show how to use it, including examples of extensions. After, we present a set of tools devoted to improve the testing, and we finally present an ongoing effort devoted to build a GUI for the toolkit.

2. DESCRIBING THE PROCESSOR ARCHITECTURE.

The Alpha-1 simulated computer is built as a set of interacting submodels representing the behavior of the architectural components. The processor **organization** is mainly based in the specification of the Integer Unit of the SPARC processor [Sun01]. The Figure 1 shows a sketch of this architecture. This RISC processor is provided with 520 general purpose integer registers; eight of them global (*RegGlob*, shared by every procedure), and 512 organized in windows of 24 registers each (*RegBlock*). Each window includes 8 input, 8 output and 8 local registers for each executing procedure.

The processor includes several special purpose registers:

- **PCs**: there are two program counters. PC contains the address of the next instruction, and nPC stores the address of the next PC. If the instruction is a conditional branch, nPC is assigned to PC, and nPC is updated with the jump address.

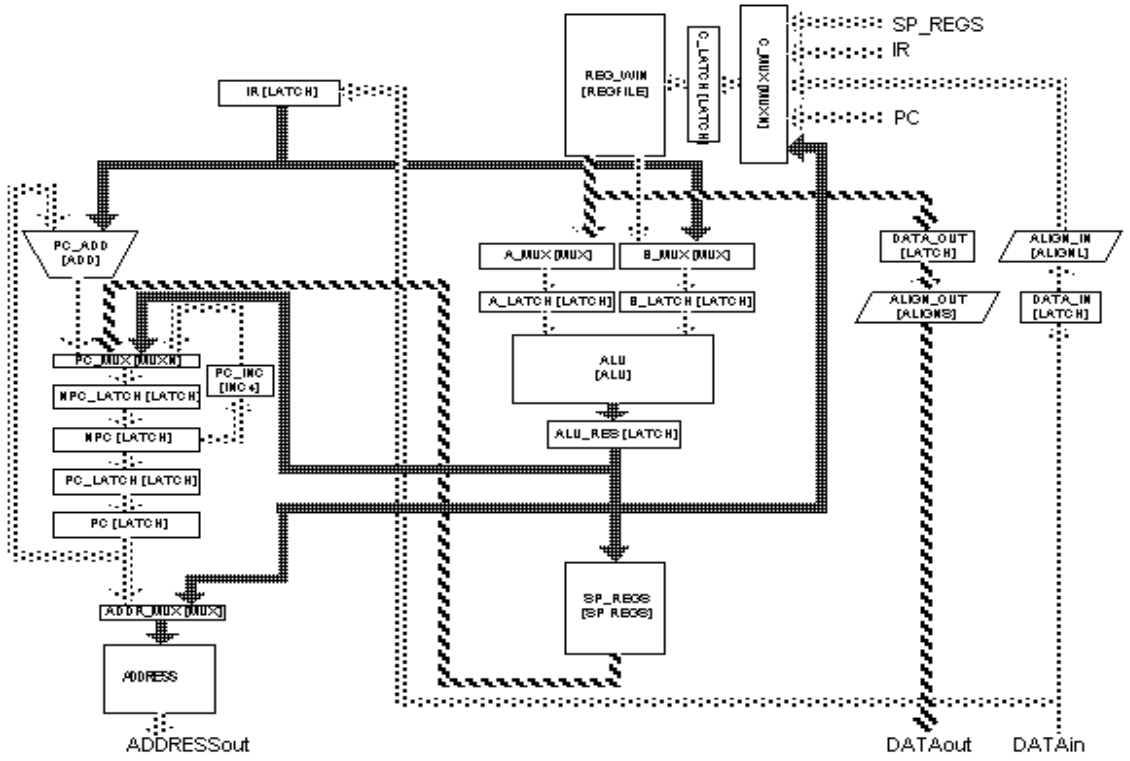


Figure 1. Organization of the Integer Unit.

- **Y**: is used by the product and division operations;
- **BASE** and **SIZE:** we used a flat memory addressing scheme. The BASE register points to the lowest address a program can access, and SIZE stores the program size, which represents the maximum addressable memory unit.
- **PSR** (Processor Status Register): stores the program status. It is interpreted as follows:

Bits	Content	Description
31..24	Reserved	
23	N – Negative	1 when the result of the last operation is negative
22	Z – Zero	1 when the last operation is zero
21	V – Overflow	1 when the last operation is overflow
20	C – Carry	1 when the last operation carried one bit
19..12	Reserved	
11..8	PIL – Processor Interrupt Level	Lowest interrupt number to be serviced.
7	S – State	1= Kernel mode; 0=User mode.
6	PS – Previous State	Last mode.
5	ET – Enable Trap	1=Traps enabled; 0=Traps disabled.
4..0	CWP – Current Window Pointer	Points to the current register window.

- **CWP** (Circular Window Pointer): it is a specialized 5-bit register that marks the active window. Every time a new procedure starts, *CWP* is decremented.
- **WIM** (Window Invalid Mask): this 32-bit register is used to avoid overwriting a register window. When *CWP* is decremented (because a procedure was called), the *WIM* bit is verified for the new window. If it is active, an interrupt is raised and the service routine must store the window contents in memory. *WIM* usually marks the oldest window.
- **TBR** (Trap Base Register): it points to the memory address storing the location of trap routines. The first 20 bits (Trap Base Address) store the base address of the trap table. When an interrupt request is received, the number of trap to be serviced is stored in the bits 11..4. Therefore, the TBR points to the table position containing the address of the service routine. The last 4 bits in 0 guarantees at least 16 bytes to store each routine.

Bits	Content	Description
31..12	Trap base address	Base address of the Trap table
11..4	Trap Type	Trap to be serviced
3..0	Constant (0000)	

The computer emulates a reduced version of the **instruction set** level of the SPARC architecture [Sun01]. This processor uses instructions with a fixed size of 32 bits, with 8, 16 or 32 bit operands. There are two basic *Load*/*Store* operations, classified according to the size and sign of their operands. Arithmetic and Boolean operations include *add*, *and*, *or*, *div*, *mul*, *xor*, *xnor*, and *shift*. Several *jump* instructions are available, including *relative* jumps, *absolute* jumps, *traps*, *calls*, and *return* from traps. *Multiplication* uses 32-bit operands, producing 64-bit results. The 32 most significant bits are stored in the Y register, and the remaining in the ALU-RES register. Integer *division* operation takes a 64-bit dividend and a 32-bit divisor, producing a 32-bit result. The Y register stores the 32 most significant bits of the dividend. One ALU input register stores the least significant bits of this number, and the other, the divisor. The integer result is stored in the ALU-RES register, and the remainder in the Y register. Other instructions include changing the movement of the register window, NOPs, and read/write operations on the PSR. There are two execution modes: *User* and *Kernel*. Certain instructions can only be executed in Kernel mode. Also, the Base and Size registers are used only when the program is running in User mode.

The CPU executes under the supervision of the **Control Unit**. It receives signals from the rest of the processor using 64 input bits (organized in 5 groups: the Instruction Register, the PSR, BUS_BUSY_IN, BUS_DACK_IN, and BUS_ERR). Its outputs are sent using 70 lines organized in 59 groups. Some of them include reading/writing internal registers, activating lines for the ALU or

multiplexers. Also, connections with the PC, nPC, Trap controller and PSR registers are included. Finally, the Data, Address and Control buses can be accessed.

The **memory** is organized using byte addressing and the Little-Endian standard to store integers. The processor issues a memory access operation by writing an address (and data, if needed) in the bus. Then, it turns on the AS (Address Strobe) signal, interpreted by the memory as an order to start the operation. The memory uses the address available and analyzes the RD/WR line to see which operation was asked. If a *read* was issued, one word (4 bytes) is taken from the specified address and sent through the data lines. In a *write* operation the address stored in the Byte Select register (lines BSEL0..3) defines the byte to be accessed in the word pointed by the Address register. If an address is wrong, the ERR line is turned on. A Data Acknowledge (DTACK) is sent when the operation finished.

This architecture was implemented using the CD++ tool. First, the behavior of each component was specified, analyzing inputs and outputs of the original circuits. This specification allowed to provide test cases derived from the specification (details of this phase can be found in [Self2]). Then, each subcomponent of the computer was defined as a DEVS model following the specification. After, each model was implemented in CD++, including and experimental framework following the test cases defined in the specification (details of this phase can be found in [Self3]). Finally, the main model was built as a coupled model connecting all the submodels previously defined.

Some of the components were implemented using two different approaches: first, the dynamics of the circuit were coded in transition functions, building behavioral models. Then, Boolean logic was used to build structural models by combining Boolean gates. Two different abstraction levels are provided to be used according to the interest. Besides this, a third level of abstraction is provided by the microcoded operations in the Control Unit. Finally, a fourth abstraction level can be studied by coding programs in assembly language. A higher level of abstraction would consist in coding programs in high-level languages to be translated into assembly or executable code, or building an operating system kernel in assembly language, making available a wide variety of tools in each of the levels.

Individual tests were developed for each of the components. Finally, integration tests were executed. The execution flow of the computer is carried out by the Control Unit model. This model is built using several input/output ports representing the CU lines.

According with the input received, it issues different outputs, activating the different circuits that were defined previously (details can be found in [Self3]). Finally, a thorough integration test was executed. The results of this phase will be presented in section 5.

3. USING THE COMPUTER

The Appendix is devoted to explain how to install the tools needed to work with the simulated computer. Once installed, the first step is to write the source code of the program intended to be executed using SPARC Assembly language [Sun01]. Then, the assembler source code should be assembled and linked using the standard software available for SPARC processors. The result is an executable file, which must be loaded as a memory image in the simulated memory (the procedures to do this are explained in the Appendix). Then, the simulation tool should be activated. Once the program has ended, you can dump the Memory at the Processor Halt, and check the memory contents. While the simulation is being executed, a detailed log file shows the components being activated, allowing to analyze changes in any state variables contained in memory, processor, or bus components. The following figure shows a sketch of these procedures (details about how to execute the toolkit can be found in the Appendix 1).

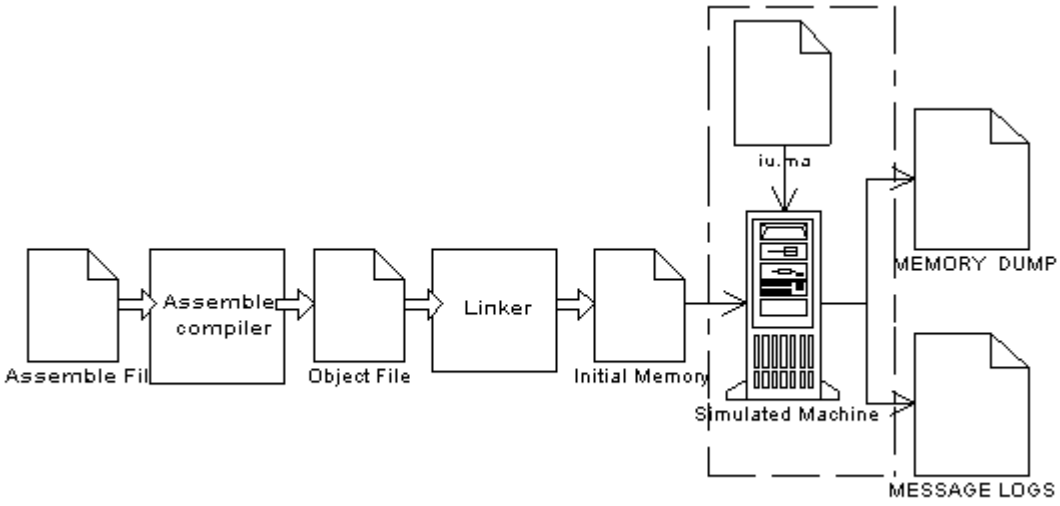


Figure 2. Process of simulation from the assembler file to the memory dump

We will show this procedure using an example. The first lines in the following figure show parts of a program written in assembly language. The second part presents the binary code generated when assembled, together with the absolute addresses for each instruction or data (one word each). The idea of this simple program is to place a 1 in a given address, and then shift this value to the left, storing the result in the following address. The cycle is repeated 12 times. We show the translation of the binary code based

on the specification of the instruction set of the SPARC processor. Finally, we show the memory image after the program execution, where the values stored in memory have followed the instructions defined in the executable code.

```
        set 1, %r1                              ! Load the register 1 with the value 1
cycle:  sll %r1, %r2, %r3                       ! Shift the value the number of times in r2
        stb %r3, [%r2+dest]                     ! Store the result in the variable dest + r2

        subcc %r2, 12, %r0                      ! Repeat the cycle 12 times
        bne cycle
        inc 1, %r2 !Delay slot

        unimp

dest:   .ascii "                        "
        .ascii "                        "
        .ascii "                        "
        .ascii "                        "
        .ascii "                        "

Initial Image
Addr.             Memory Image              Interpretation
...
032     10000010 00010000 00100000 00000001  set <1>, 1
036     10000111 00101000 01000000 00000010  Take the register 1, shift and store in R3
040     11000110 00101000 10100000 00111100  Store in address 60 1 byte
044     10000000 10100000 10100000 00001100  substract 12 to R0
048     00010010 10111111 11111111 11111101  Relative jump to address –2 words (40)
052     10000100 00000000 10100000 00000001  increment 1
056     00000000 00000000 00000000 00000000  unimp
060     00100000 00100000 00100000 00100000  Destination variable
064     00100000 00100000 00100000 00100000
068     00100000 00100000 00100000 00100000
072     00100000 00100000 00100000 00100000
...
Final image
...
060     00000001 00000010 00000100 00001000  A value 0x01 shifted 12 times
064     00010000 00100000 01000000 10000000
068     00000000 00000000 00000000 00000000
...
```

Figure 3. Execution of a simple routine.

Besides the memory maps, the execution logs allow seeing the control flow in the processor data path. The following figure shows parts of a the log file generated when the previous program was executed. It includes the messages interchanged between executing models. There are four types of messages: ***** (used to signal a state change due to an internal event), **X** (used when an external event arrives), **Y** (the model's output) and **done** (indicating that a model has finished with its task). The **I** messages initialize the corresponding models. For each message we show its type, timestamp, value, origin/destination, and the port used for the transmission.

```
Message I / 00:00:00:000 / Root(00) to top(01)      // Initialize the higher level
Message I / 00:00:00:000 / top(01) to mem(02)       // components: memory, bus, CS, etc.
Message I / 00:00:00:000 / top(01) to bus(03)
Message I / 00:00:00:000 / top(01) to csmem(04)
Message I / 00:00:00:000 / top(01) to cpu(05)
Message I / 00:00:00:000 / top(01) to c1(64)
Message I / 00:00:00:000 / top(01) to dpc(65)
Message D / 00:00:00:000 / mem(02) / ... to top(01) // The models reply the next
Message D / 00:00:00:000 / bus(03) / ... to top(01) // scheduled event
Message D / 00:00:00:000 / csmem(04) / ... to top(01)
Message I / 00:00:00:000 / cpu(05) to ir(06)        // The CPU initializes the components
Message I / 00:00:00:000 / cpu(05) to pc_add(07)
```

```
Message I / 00:00:00:000 / cpu(05) to pc_mux(08)
...
Message * / 00:00:00:000 / Root(00) to top(01)
Message * / 00:00:00:000 / top(01) to cpu(05)
Message * / 00:00:00:000 / cpu(05) to npc(10)       // Take the nPC
Message Y / 00:00:00:000 / npc(10) / out2 / 1.000 to cpu(05)
Message Y / 00:00:00:000 / npc(10) / out5 / 1.000 to cpu(05)
Message D / 00:00:00:000 / npc(10) / ... to cpu(05)
Message X / 00:00:00:000 / cpu(05) / in2 / 1.000 to pc_latch(11)   // Send to pc-inc
Message X / 00:00:00:000 / cpu(05) / op2 / 1.000 to pc_inc(13)     // to increment
Message X / 00:00:00:000 / cpu(05) / in5 / 1.000 to pc_latch(11)   // the value
Message X / 00:00:00:000 / cpu(05) / op5 / 1.000 to pc_inc(13)
Message D / 00:00:00:000 / pc_latch(11) / 00:00:10:000 to cpu(05)         // Schedule the
Message D / 00:00:00:000 / pc_inc(13) / 00:00:10:000 to cpu(05)   // activation of the
Message D / 00:00:00:000 / pc_latch(11) / 00:00:10:000 to cpu(05) // pc-inc model
Message D / 00:00:00:000 / pc_inc(13) / 00:00:10:000 to cpu(05)
Message D / 00:00:00:000 / cpu(05) / 00:00:00:000 to top(01)
Message D / 00:00:00:000 / top(01) / 00:00:00:000 to Root(00)
Message * / 00:00:00:000 / Root(00) to top(01)
Message * / 00:00:00:000 / top(01) to cpu(05)
Message * / 00:00:00:000 / cpu(05) to pc(12)
Message Y / 00:00:00:000 / pc(12) / out5 / 1.000 to cpu(05)              // Initial address
Message D / 00:00:00:000 / pc(12) / ... to cpu(05)                // 010000 = 32
...
Message * / 00:00:00:000 / Root(00) to top(01)
Message * / 00:00:00:000 / top(01) to cpu(05)
Message * / 00:00:00:000 / cpu(05) to clock(45)                   // Clock tick
Message Y / 00:00:00:000 / clock(45) / clck / 1.000 to cpu(05)
Message D / 00:00:00:000 / clock(45) / 00:01:00:000 to cpu(05)
Message X / 00:00:00:000 / cpu(05) / clck / 1.000 to cu(43)
Message D / 00:00:00:000 / cu(43) / 00:00:00:000 to cpu(05)
Message D / 00:00:00:000 / cpu(05) / 00:00:00:000 to top(01)
Message D / 00:00:00:000 / top(01) / 00:00:00:000 to Root(00)
Message * / 00:00:00:000 / Root(00) to top(01)
Message * / 00:00:00:000 / top(01) to cpu(05)       // Arrival to the CU
Message * / 00:00:00:000 / cpu(05) to cu(43)        // And activation of the components
Message Y / 00:00:00:000 / cu(43) / a_mux_reg / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / b_mux_reg / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / enable_alu / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / addr_mux / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / ir_latch_en / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / as / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / rd_wr / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / busy / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / c_mux2 / 1.000 to cpu(05)
Message Y / 00:00:00:000 / cu(43) / pc_mux0 / 1.000 to cpu(05)
Message D / 00:00:00:000 / cu(43) / ... to cpu(05)
...
Message * / 00:00:10:000 / Root(00) to top(01)
Message * / 00:00:10:000 / top(01) to cpu(05)
Message * / 00:00:10:000 / cpu(05) to pc_latch(11)
Message D / 00:00:10:000 / pc_latch(11) / ... to cpu(05)
Message D / 00:00:10:000 / cpu(05) / 00:00:00:000 to top(01)
Message D / 00:00:10:000 / top(01) / 00:00:00:000 to Root(00)
Message * / 00:00:10:000 / Root(00) to top(01)
Message * / 00:00:10:000 / top(01) to cpu(05)
Message * / 00:00:10:000 / cpu(05) to pc_inc(13)           // Update the nPC
Message Y / 00:00:10:000 / pc_inc(13) / res3 / 1.000 to cpu(05)
Message Y / 00:00:10:000 / pc_inc(13) / res5 / 1.000 to cpu(05)
Message D / 00:00:10:000 / pc_inc(13) / ... to cpu(05)
...
Message * / 00:00:20:001 / Root(00) to top(01)
Message * / 00:00:20:001 / top(01) to mem(02)       // Memory returns the first instr.
Message Y / 00:00:20:001 / mem(02) / dtack / 1.000 to top(01)
Message Y / 00:00:20:001 / mem(02) / out_data0 / 1.000 to top(01)
Message Y / 00:00:20:001 / mem(02) / out_data13 / 1.000 to top(01)
Message Y / 00:00:20:001 / mem(02) / out_data20 / 1.000 to top(01)
Message Y / 00:00:20:001 / mem(02) / out_data25 / 1.000 to top(01)
Message Y / 00:00:20:001 / mem(02) / out_data31 / 1.000 to top(01)
Message D / 00:00:20:001 / mem(02) / ... to top(01)
Message X / 00:00:20:001 / top(01) / in_dtack / 1.000 to bus(03)
Message X / 00:00:20:001 / top(01) / in_data0 / 1.000 to cpu(05)
Message X / 00:00:20:001 / top(01) / in_data13 / 1.000 to cpu(05)
Message X / 00:00:20:001 / top(01) / in_data20 / 1.000 to cpu(05)
Message X / 00:00:20:001 / top(01) / in_data25 / 1.000 to cpu(05)
Message X / 00:00:20:001 / top(01) / in_data31 / 1.000 to cpu(05)
```

```
Message D / 00:00:20:001 / bus(03) / 00:00:00:001 to top(01)
...
```

Figure 4. Log file of a simple routine.

The execution cycle starts by initializing the higher level models (memory, CPU, etc.). The message arrived to the CPU model is sent to its lower level components: Instruction Register, PC Adder, PC multiplexer, Control Unit, etc. When the initialization cycle has finished, the imminent model is executed. In this case, the *nPC* model is activated, transmitting the address of the next instruction. As we can see, the 2nd and 5th bits are returned with a 1 value. That means that the nPC value is 100100 = 36 (as we see in figure 4, the program starts in the address 32). The value is sent to the *pc-inc* model, in charge of adding 4 to this register. The update is finished in 10:000, as the activation time of this model was scheduled using the circuit delay. At that moment, a 4 value is added to the nPC, and we obtain the 3rd and 5th bits in 1 (*res3* and *res5*), that is, 101000 = 40, the next PC.

After, the PC is activated, and the value 010000 (that is, 32) is obtained. This is the initial address of the program. The following event is the arrival of a clock tick, sent to the processor. The CPU schedules the next tick (in 1:00:000 time units) and transmits the signal arrives to the Control Unit, which activates several components: *a-mux*, *ALU*, *Addr-mux*, *IR*, etc.

We finally see, in the simulated time 20:000, that the memory has returned the first instruction (compare the results with the bit configuration in the address 32). The instruction is sent to the CPU to be stored in the Instruction Register and to follow with the execution. The rest of the instruction cycle is completed in the same way. Following the log file, or connecting output models to the output lines in the processor, we are able to follow the execution flow for any program.

4. EXTENDING THE ALFA-1 SIMULATOR

After installing and running the simulator, the user can modify or extend the existing components. To do so, atomic models may need to be modified or added. After, they can integrate any existing coupled model. In this section, we present a sketch showing how to code new models, following the guidelines presented in [WT01].

4.1. New Atomic models

In CD++, a new atomic model is created as a new class that inherits from *Atomic* base class. To tell CD++ that a new atomic definition has been added, the model must be registered in the MainSimulator.registerNewAtomics() function. The state of a model

is made of all those variables that can change during a simulation cycle. The basic state variables required by an atomic model are defined in the *AtomicState* class. A user can create a new class to define the state variables required by his model.

```
class AtomicState : public ModelState {
public:
      enum State {
            active,
            passive
      } ;

      State st;
      AtomicState(){};
      virtual ~AtomicState(){};

      AtomicState& operator=(AtomicState& thisState); //Assignment
      void copyState(BasicState *);
      int  getSize() const;
};
```

Figure 5. The AtomicState class.

To access the current state the function **`ModelState* getCurrentState()`**should be used. The pointer that is returned can be casted to the proper type. When creating a new atomic model, a new class derived from atomic has to be created. *Atomic* is an abstract class that declares a model's API and defines some service functions the user can use to write the model. The *Atomic* class provides a set of services and requires a set of functions to be redefined:

- holdIn(state, Time) : tells the simulator that the model will remain in the state *state* for a period of Time *time.* It corresponds to the D(s) function of the DEVS formalism

- passivate(): sets the next internal transition time to infinity. The model will only be activated again if an external event is received.

- sendOutput(Time, port, BasicMsgValue*): sends an output message through the port. The time should be set to the current time.

- nextChange(): Returns the remaining time for the next internal transition.

- lastChange(): Returns the time since the last state change

- state(): Returns the current model's phase.

```
class Atomic : public Model  {
public:
   virtual ~Atomic(); // Destructor

protected:

   //User defined functions.
   virtual Model &initFunction() = 0;
   virtual Model &externalFunction( const ExternalMessage & );
   virtual Model &internalFunction( const InternalMessage & ) = 0 ;
   virtual Model &outputFunction( const CollectMessage & ) = 0 ;
   virtual string className() const

   //Kernel services
   Time nextChange();
```

```
   Time lastChange();

   Model &holdIn( const AtomicState::State &, const Time & ) ;
   Model &sendOutput(const Time &time, const Port & port , Value value)
   Model &passivate();

   //State functions
   virtual ModelState* getCurrentState() const;
   virtual ModelState* getCurrentState() ;

   //State shortcuts
   Model &state( const AtomicState::State &s )
   { ((AtomicState *)getCurrentState())->st = s; return *this; }

   const AtomicState::State &state() const
   {return ((AtomicState *)getCurrentState())->st;}
}; // class Atomic
```

Figure 6. The Atomic Class

The new class should override the following functions:

- virtual Model &initFunction(): this method is invoked by the simulator at the beginning the simulation and after the model state has been initialized.

- virtual Model &externalFunction(const ExternalMessage &): this method is invoked when one external event arrives to a port. It corresponds to the δ_{ext} function of the DEVS formalism.

- virtual Model &internalFunction(const InternalMessage &): this method corresponds to the δ_{int} function of the DEVS formalism.

- virtual Model &outputFunction(const CollectMessage &): it is in charge of sending all the output events of the model.

4.2. New Coupled Models

After each atomic model is defined, they can be combined into a multicomponent model. Coupled models are defined using a specification language specially defined with this purpose. The language was built following the formal definitions for DEVS coupled models. Optionally, configuration values for the atomic models can be included.

The **`[top]`** model always defines the coupled model at the top level. As showed in formal specifications presented in section 1, four properties must be configured: components, output ports, input ports and links between models. The following syntax is used:

`Components`: `model_name1[@atomicclass1] [model_name2[@atomicclass2] ...`

Lists the component models that integrate the coupled model. A coupled model might have atomic models or other coupled model as components. For atomic components, an instance name and a class name must be specified. This allows a coupled model to use more than one instance of an atomic class. For coupled models, only the model name must be given. This model name must be defined as another group in the same file.

Out: `portname1 portname2 ...`

Enumerates the model's output ports. This clause is optional because a model may not have output ports.

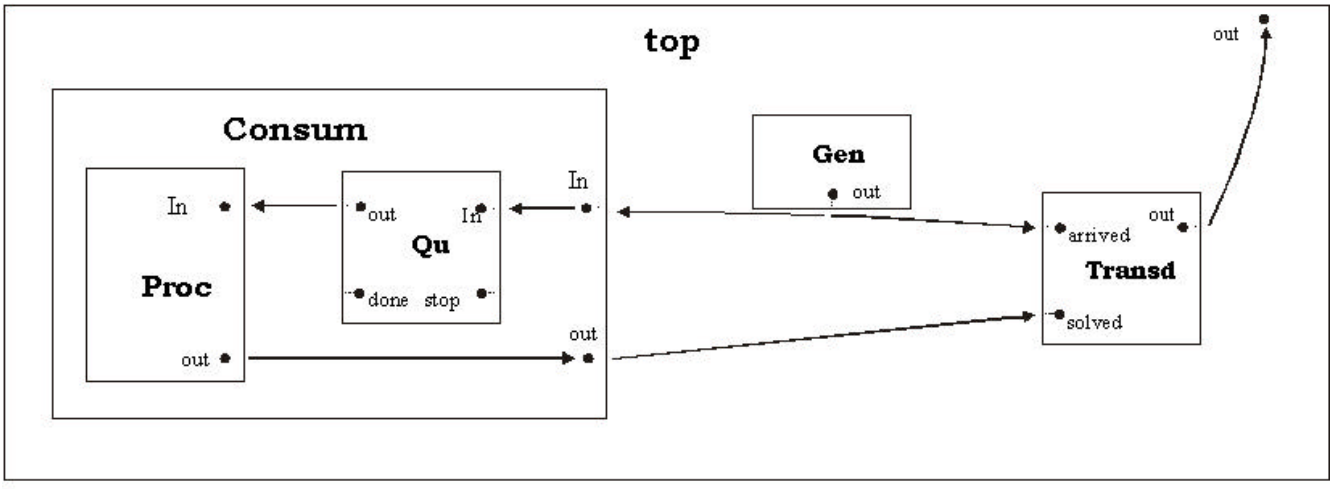
In: `portname1 portname2 ...`

Enumerates the input ports. This clause is also optional because a coupled model is not required to have input ports.

Link: `source_port[@model] destination_port[@model]`.

It describes the internal and external coupling scheme. The syntax is: The name of the model is optional. The model that will be used by default is the coupled model currently being defined.

The following figure shows a sample coupled model and its specification in CD++. It consists of three models: a generator, in charge of creating data to be consumed, a consumer, and a transducer, in charge of measuring the consumer speed. The consumer is also a coupled model, composed by a processor and a queue to keep waiting jobs.




```
[top]
components : Transd@Transducer Gen@Generator Consum
Out : out
```

```
Link : out@generator arrived@transducer
Link : out@generator in@Consumer
Link : out@Consumer solved@transducer
Link : out@transducer out

[Consum]
components : Qu@Queue Proc@CPU
in : in
out : out
Link : in in@qu
Link : out@qu in@Proc
Link : out@Proc done@qu
Link : out@Proc out
```

Figure 7. Example for the definition of a DEVS coupled model

In the top level of this example, the *Generator* influences the *Transducer* and the *Consumer*. The *Consumer* also influences the *Transducer*. The *Consumer* influences the *Queue* and the Processor also influences it. Then, the *Queue* influences the *Processor*. Finally, the *Transducer* influences the top model. These influences define the influencee's sets for each of the components, which is used to define the translation functions. The figure shows the influences for this example, which are carried out by transmitting information through the input/output ports in the models.

4.3. A sample look-aside component

Using the previous descriptions, the user can modify or change existing components. In this section we will focus on extending the simulator by adding components external to the processor. A look-aside component (usually an input/output device) is attached to one of the bus slots. Therefore, to add any kind of component we must first get a good grip on how the bus protocol works on the ALFA-1. Following the architecture description in [Self2], we can see that the bus uses the following lines:

Pin Name	***In***	***Out***	***Description***
Data0-31	X	X	Data
A0-31	X	X	Address
Clock	X	X	Clock
AS	X	X	Address Strobe
RD/WR	X	X	Read or Write
DTACK	X	X	Data Acknowledge
Err	X	X	Error
RESET	X	X	Reset
IRQ1-15	X	X	Interrupt Request
Busy	X	X	Bus Busy

When information transfers are carried out over the bus, two participating components can be identified: an active and a passive

one. The first thing that an active component should do is to take over the bus. This can be done in two steps. First it must assert that it will be the next one to use the bus. All the active components are chained by a bus grant signal (BGRANT). Every component will receive a BGRANT line from the component with next higher priority and it will send a BGRANT signal to the component with next lower priority (Figure 8).

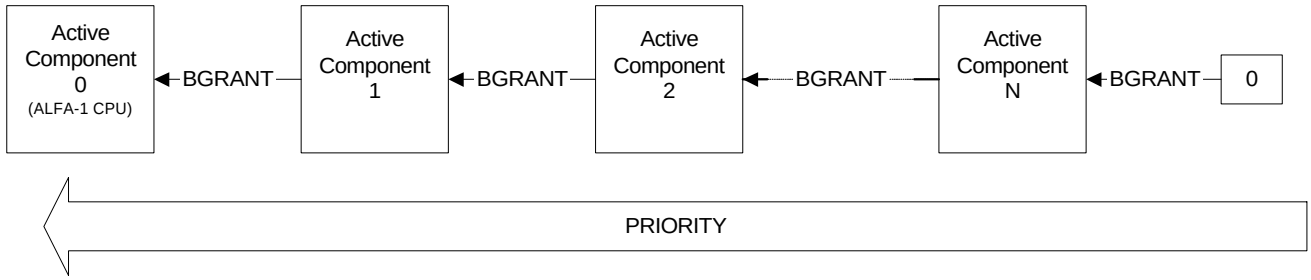


Figure 8. Active components BGRANT chain

If the component receives a 1 over the BGRANT input line, this means that none of the components with higher priority will need the bus. Then, it sends a 0 over the BGRANT output line to advice the components with lower priority that it is intending to be the next at the bus. Then it must wait until a 0 is received over the BUSY line meaning that the bus is no longer being used, finally it will send a 1 over the BUSY signal letting the other components know that the bus is being used.

The next thing to do after taking over the bus is to set the address to be accessed on the address bus (A0-31). The address to be used can be split in two levels. On a first level, the active component will choose which 32-bit word will be used by setting its address on the address bus. On the second level the component can also split the 32 bit word into four parts (D0-7, D8-15, D16-23, D24-31) and choose any combination of them by using the BSEL signal. If the active component has set a 1 into BSEL0 then D0-7 will be addressed, BSEL1 corresponds to D8..15, etc. If the component has requested to write by sending a 0 over RD/WR, it will also send the data to be written over the selected bytes of the data bus (D0-31). Subsequently the active component will send a 1 over the AS line to let the passive component know that everything's been set for the request and it will wait until a 1 is received over the DTACK line meaning that the request has been fulfilled. If the component was performing a read operation, gather the data on the D31-0 signals. The active component should now send a 0 over the AS line to finish the operation. Finally the active component can now perform another operation-he is still in control of the bus-or release the bus by outputting BGRANT with 1 and BUSY with 0.

The passive component (for instance the main memory) will use a different protocol. It will first receive a request for being read or written, meaning that the active component it's trying to get/write data from/to. The request will be issued by first sending a 1 over the AS signal (meaning that the request is on the bus). Then, it will receive the address (A0-31), the bytes requested in the byte mask (BSEL0-3). It must verify the memory address (it must be the specified address space for this component), and if it is a write request, the data to be written should be available over the selected bytes of the data bus (D0-31). Then, it will output or receive data from Data0-31 according to the action specified by the RD/WR signal. When the request has been fulfilled (the required data is on the data bus (D0-31) for a read, the data was taken from written into the component), the passive component will set a 1 over the DTACK. Finally the AS signal will be turned to 0 again meaning the communication has finished, the passive component should release its used lines by sending a 0 over the DTACK line.

Using this basic information about the bus lines, we will show the specification and implementation of a look-aside component. This model returns 1s in all the data bus D31-0 bits selected by the BSEL3-0 mask when a read operation is issued. It returns an error if it was asked to be written. A look-aside component is attached to the bus like the memory so it must be connected to the bus in the same way. The minimal signals used by of the component will be the following:

Signal	***In***	***Out***
RD/WR	X	
AS	X	
DTACK		X
ERR		X
$D_{32..0}$	X	X
$A_{32..0}$	X	
$Bsel_{3..0}$	X	

This atomic model can be defined as:

$$\textbf{SAMPLELA} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >.$$

X =AS, RD/WR,A $\in$ {0,4,…,2^{32}-1}, BSEL {0,..,2^4-1};

S = responseTime $\in$ $\boldsymbol{R}_0^+$;

Y=D $\in$ {0,...,2^{32}-1}, DTACK, ERR;

and the transition functions can be informally described as follows:

```
δext() {
   When (1 is received in the port AS)
        if the value on received over A32..2 is in our address space then
                if (RD/WR=1) then
                        if (BSEL0 = 1) then D7..0 := FFh endif
                        if (BSEL1 = 1) then D15..8 := FFh endif
                        if (BSEL2 = 1) then D23..16 := FFh endif
                        if (BSEL3 = 1) then D31..24 := FFh endif
                        DTACK:=1
                else
                        ERR:=1
                endif
        endif
   endwhen;
   When (0 is received in the port AS) and (1 is being sent in the port DTACK)
        DTACK:=0;
   endwhen;
   hold_in(responseTime)
}

δint() { passivate; }

λ() {
   IF (D31..0 is different to the last output) send D31..0 over the D31..0 ports
   IF (DTACK is different to the last output) send DTACK over the DTACK port
   IF (ERR is different to the last output) send ERR over the ERR port
   continue;
}
```

Figure 9. Behavior of the samplela look-aside component

As we can see, this component uses the algorithm previously presented for passive components. This behavior is implemented on **δext()**, when it senses a 1 over the AS line. Every passive component attached to the bus must have a designated address space since that's the only way a passive component has to know if that request was for it. Hence, the component first checks the address space. If the component finds out that the request was for it, it will answer, filling the selected data bus (D0-31) bytes with 1s. Finally the component will wait for the active component to set a 0 over the AS signal. If this happens, and it was the one answering, the *samplela* component will send a 0 over the DTACK line to end the communication. If we want to simulate a delay on the response of the component we must tell the atomic model to wait for some time before jumping to the next step. Thus, we call the `hold_in` procedure with the respective delay as a parameter.

On the other hand **δint()** performs just one operation, `passivate`. This means that the component will remain inactive until a

signal arrives on any input port and the external function **δint()** just sends the signals over the output ports when needed. The following figure shows the implementation of these functions using the CD++ tool .

```
Model & sampleLA::initFunction() {
   //Establishing the initial output and waiting time
   m_ctlOut[dtackout].val=0;
   holdIn( active, Time::Zero );
   return *this;
}

Model & sampleLA::externalFunction( const ExternalMessage & msg) {
   string portName;
unsigned long portNum;
   unsigned i;

   //Reading over what port the message arrived and which was its value
   nameNum(msg.port().name(), portName, portNum );

   for( i=0; i<NUM_CTL_IN_PORT; i++)
        if ( portName == ctlInPortNames[i] )
                m_ctlIn[i]= bit (msg.value());

   if ( portName == string(addrInPortNames[0]) ){
        m_addrIn[portNum]= bit (msg.value());
   }

   if (portName == bselInPortNames[0] )
        m_bselIn[portNum]= bit (msg.value());

   if (portName == dataInPortNames[0] )
        m_dataIn[portNum]= bit (msg.value());

// This component will only output all Data bits in 1 when someone tries to read from its address space

   if (m_ctlIn[asin] && (m_state==waiting)) { //If someone has requested a read/write operation to the bus

        unsigned long long addrRequested = fromBits(m_addrIn,m_addrWidth);

        if ((m_minAddrSpaceLimit <= addrRequested) && (addrRequested < m_maxAddrSpaceLimit)) {
                if (m_ctlIn[rwin] == 0) //if its trying to write to the model
                        m_ctlOut[errout].val  = 1; //we send an error
                else  {
                        for (unsigned long long i = 0 ; i< m_bselWidth; i++)
                                if (m_bselIn[i])
                                        for (unsigned long long j=0 ; j<m_byteSize ; j++)
                                                m_dataOut[i*m_byteSize+j].val = 1;
                        m_ctlOut[dtackout].val = 1;
                }
        }

        m_state = sending;
   } else if (m_state == waiting) {
        //If we don't receive an AS signal we output 0 on dtack
                m_ctlOut[dtackout].val = 0;
                m_ctlOut[errout].val = 0;
                m_state = waiting;
   }
   this->holdIn( active, m_responseTime );
   return *this;
}

Model & sampleLA::internalFunction( const InternalMessage & msg) {
   this->passivate();
   return *this;
}

Model & sampleLA::outputFunction( const InternalMessage & msg) {
   //Outputing data over the ports
   for(unsigned i=0; i<NUM_CTL_OUT_PORT; i++)
        if (needSend(m_ctlOut[i]))
```

```
                this->sendOutput( msg.time(), *m_ctlOutPorts[i], m_ctlOut[i].val);
   for(unsigned i=0; i<m_dataWidth; i++){
        if (needSend(m_dataOut[i]))
                this->sendOutput( msg.time(), *m_dataOutPorts[i], m_dataOut[i].val );
   }
   return *this;
}
```

Figure 10. CD++ samplela look-aside component implementation

The initialization function (`initFunction`) is executed as soon as the simulation starts. The first lines of the `externalFunction` (δ**`ext()`**) are devoted to identify the signal that has recently arrived. Since this function is activated every time a signal arrives to an input port, we identify only one signal at a time. We will distinguish if we're sending a response or waiting for a request using the `m_state` variable. There are two arrays named `m_ctlIn` and `m_ctlOut`, containing the input and output control ports respectively. These ports are referenced within the arrays by enumerated constants which represent each one of the ports, for instance the `rwin` constant within the `m_ctlIn` array represents the RD/WR port, hence, `m_ctlIn[rwin]` will contain a 1 when the last input received on that port was 1. We will send data over the output ports if they need to change the value they are sending. Therefore output ports (`m_ctlOut`, `m_dataOut`) are represented by arrays of a data type that will allow to distinguish between the ones that need to be sent and the ones that don't. The data type used for output arrays can be queried about its need to be sent by the `needSend` function. Its value can be changed at the `.val` internal variable for instance `m_ctlOut[dtackout].val=1` means that, if needed a 1 will be sent over the DTACK output port. Finally the `internalFunction` (δ**`int()`**) and `outputFuntction`(λ) code its just the implementation of that function's on figure 9.

This model behaves following the bus specification, and it is suitable to attach to the bus and work as a memory component. Assuming you have compiled the simulator with the new component and that it's name is *samplela* this is done by simply editing the coupled model file of the ALFA-1 simulator (*iu.ma*) and adding the following lines in [top]:

```
components: samplela

Link : OUT_A2@bus A2@samplela
. . .
Link : OUT_A31@bus A31@samplela

Link : OUT_DATA0@samplela IN_DATA0@cpu
. . .
Link : OUT_DATA31@samplela IN_DATA31@cpu
```

```
Link : OUT_DATA0@cpu IN_DATA0@samplela
. . .
Link : OUT_DATA31@cpu IN_DATA31@samplela

Link : DTACK@samplela IN_DTACK@bus
Link : OUT_RD_WR@bus RW@samplela

Link : OUT_BSEL0@bus BSEL0@samplela
. .
Link : OUT_BSEL3@bus BSEL3@samplela
```

Figure 11. The iu.ma definition of the ALFA-1

These lines are just connecting the *samplela* model to the bus. More precisely, following the original *iu.ma* connections, the address bus (A2-31), the BSEL signals and the RD/WR are connected from the *bus* component onto the *samplela* component. The DTACK signal is connected the other way round, from *samplela* to *bus*; the data bus (D0-31) is connected directly from and to the *cpu* component, the connection ought to be in both ways due to the bi-directional characteristic of this bus.

On the other hand, the *samplela* model will have a response delay that can be configured using the `responseTime` atomic model parameter. This argument, as well as the address space boundaries `minAddr` and `maxAddr` (lower and higher bounds respectively) will be passed to the model by adding the following lines at the end of the *iu.ma* file.

```
[samplela]

responseTime = 0:0:0:0
minAddr =  65536
maxAddr = 131072
```

Figure 12. Adding the atomic component samplela to the ALFA1

4.4. A sample look-through component

A look-through component (usually a cache memory) is inserted between the microprocessor and the bus and it should be transparent to the other components attached to the bus. We will now include the specification of a look-through component, returning the data bus D31-0 bits selected by the BSEL3-0 bit mask filled with 1s if it was asked to be read, and an error if it was asked to be written. Although it may seems to be practically equal to the look-aside component, we must remember that this component will be attached between the CPU and the BUS. Therefore, it must be transparent to any other components when its address space its not being referenced.

If we look at the signals of a look-through component, we can easily detect that it can be divided in two sides: the one connected to

the CPU and the side connected to the Bus. The side connected to the CPU must act as if it were a standard Bus, and the side connected to the bus as if it where a CPU thus we can expose the component's signals divided into those two sides:

	CPU			***Bus***		
Signal	In	Out	In/Out	In	Out	In/Out
Err		X		X		
Dtack		X		X		
AS	X				X	
R/$\underline{W}$	X				X	
Busy			X			X
Bgrant		X		X		
$A_{31..2}$	X				X	
$BSEL_{3..0}$	X				X	
$D_{31..0}$			X			X

This atomic model can be defined as:

$$\textbf{SAMPLELT} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >.$$

X = AS(CPU), $Busy_{in}$(CPU), $A_{in} \in \{0,...,2^5\text{-}1\}$,$BSEL_{in} \in \{0,...,2^5\text{-}1\}$

, $R/\underline{W}_{in}$, D_{in}(CPU) $\in \{0,...,2^5\text{-}1\}$,$Err_{in}$,$DTACK_{in}$,$BGrant_{in}$,$Busy_{in}$(Bus),$BGrant_{in}$,$D_{in}$(Bus) $\in \{0,..,2^5\text{-}1\}$;

S = responseTime $\in \boldsymbol{R}_0^+$

Y = Err_{out}, $Dtack_{out}$, $Busy_{out}$(CPU), $Busy_{out}$(Bus), $Bgrant_{out}$, D_{out}(CPU) $\in \{0,...,2^5\text{-}1\}$

, $R/\underline{W}_{out}$, AS(BUS),$A_{out} \in \{0,...,2^5\text{-}1\}$,$BSEL_{out} \in \{0,...,2^5\text{-}1\}$,$D_{out}$(BUS) $\in \{0,...,2^5\text{-}1\}$;

And the transition functions can be informally described as follows:

```
δext() {
   When (1 is received in the port AS(CPU))
        if the value on received over A_in32..2 is in our address space then
               if (R/W_in=1) then
                      if (BSEL_in0 = 1) then D_out(CPU)7..0 := FFh endif
                      if (BSEL_in1 = 1) then D_out(CPU)15..8 := FFh endif
                      if (BSEL_in2 = 1) then D_out(CPU)23..16 := FFh endif
                      if (BSEL_in3 = 1) then D_out(CPU)31..24 := FFh endif
                      DTACK_out:=1
               else
                      ERR_out:=1
               endif
               When (0 is received in the port AS(CPU)) and (1 is being sent in the port DTACK_out)
               DTACK_out:=0;
        else
               wait for BGRANT(BUS)=0
               wait for BUSY_in(BUS)=0
               BUSY_out(BUS):=1
               start forwarding
               wait for DTACK_in(BUS)=1
```

```
                DTACK_out(CPU):=1
                wait for AS(CPU)=0
                AS(BUS):=0
                BUSY_out(BUS):=0
                stop forwarding
                DTACK_out(CPU):=0
        endif
    endwhen
    hold_in(responseTime)
}

δint() { passivate; }

λ() {
    IF we're forwarding
        D_out(CPU)31..0:=D_in(Bus)31..0
        D_out(Bus)31..0:=D_in(CPU)31..0
        A_out31..2:=A_in31..2
        Bsel_out3..0:=Bsel_in3..0
        Err_out:=Err_in
        Dtack_out:=Dtack_in
        As_out:=As_in
        R/W_out:=R/W_in
    ENDIF
    IF (D_out(CPU)31..0 is different to the last output) send D_out(CPU)31..0  over the D_out(CPU)31..0 ports
    IF (D_out(Bus)31..0 is different to the last output) send D_out(Bus)31..0  over the D_out(Bus)31..0 ports
    IF (DTACK_out is different to the last output) send DTACK_out over the DTACK_out port
    IF (ERR_out is different to the last output) send ERR_out over the ERR_out port
    continue;
}
```

Figure 13. Behavior of the samplelt look-through component

We can see that these transition functions are not that different from the ones in the *samplela* component. The δ**ext()** function, as long as the address of the request is in the address space of the component, is the same. When the address is referencing another component, this one must act transparently, due to that in the latter case the component will transition three states, first it will take over the bus. It will start forwarding every data, every address line and the control lines RD/WR, AS, DTACK, ERR from the CPU end to the BUS end. When the communication has ended, it will release the bus and stop forwarding the signals.

The implementation of these functions using CD++ can be found in the distribution files of the Alfa-1 computer. We have used similar implementation techniques than those used in *samplela*. A variable named `m_forwardingState` is used to distinguish among the three forwarding states previously identified. In this case, the code that copies the input ports on to the output ports that take place when the component's forwarding signals between the Bus and the CPU has been moved into the external function because that way it will be executed only once, when its needed.

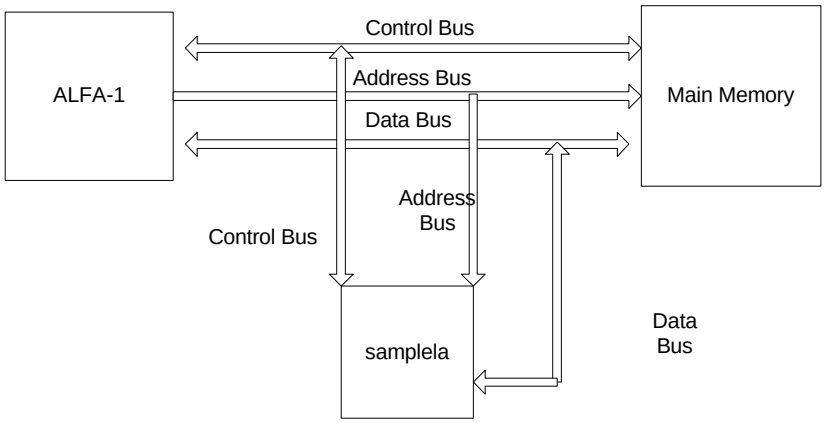


Figure 14. Implementation of the internal function samplela

This look-through component has to be attached between the CPU and the Bus. To do so, we must first detach the CPU component from the bus (described in figure 14), and we must connect every signal at both sides of the *samplelt* component (as described in the figure 15). Assuming the new component has been compiled into the simulator, and that its name is *samplelt*, we can change the coupling scheme by editing the coupled model file of the ALFA-1 (*iu.ma*). In this case, the lines

```
Link : OUT_DATA0@cpu IN_DATA0@mem
...
Link : OUT_DATA31@cpu IN_DATA31@mem

Link : AS@cpu IN_AS@bus

Link : RD_WR@cpu IN_RD_WR@bus

Link : A0@cpu IN_A0@bus
...
Link : A31@cpu IN_A31@bus

Link : BSEL0@cpu IN_BSEL0@bus
.
.
Link : BSEL3@cpu IN_BSEL3@bus
```

Figure 15. Removing connections to define a new coupling scheme

in the *top* model should be removed, and they should be replaced by the following ones:

```
Link : OUT_DATA0@cpu IN_DATA_CPU0@samplelt
...
Link : OUT_DATA31@cpu IN_DATA_CPU31@samplelt

Link : OUT_DATA0@samplelt IN_DATA_CPU0@cpu
...
```

```
Link : OUT_DATA31@samplelt IN_DATA_CPU31@cpu

Link : AS@cpu IN_AS@samplelt

Link : RD_WR@cpu IN_RW@samplelt

Link : A0@cpu IN_A0@samplelt
...
Link : A31@cpu IN_A31@samplelt

Link : BSEL0@cpu IN_BSEL0@samplelt
..
Link : BSEL3@cpu IN_BSEL3@samplelt


Link : OUT_DATA_BUS0@samplelt IN_DATA0@mem
...
Link : OUT_DATA_BUS31@samplelt IN_DATA31@mem

Link : AS@samplelt IN_AS@bus

Link : OUT_RW@samplelt IN_RD_WR@bus
Link : OUT_A0@samplelt IN_A0@bus
...
Link : OUT_A31@samplelt IN_A31@bus

Link : BSEL0@samplelt IN_BSEL0@bus
..
Link : BSEL3@samplelt IN_BSEL3@bus
```

Figure 16. New links to connect a new component

5. TESTING ALFA-1

Every time the simulated computer is modified, it is necessary to test the resulting changes. In order to improve the testing scheme, we have included a testbench that can be applied to any future modification.

In our case, we have used a Functional Testing approach [BEI90]. As it was explained, we have developed individual tests for each of the components, using a black box testing approach. The idea is to test the real execution of each of the models without knowing of their internal behavior. Black box testing implies that the selection of test data as well as the interpretation of test results are performed on the basis of the functional properties of software; the primary objective is to assess whether the simulated computer does what it is supposed to do. We have built an Experimental Framework, consisting on a data Generator connected to the model to be tested, and an Acceptor that contains information about the desired output value for each one of the generated inputs. These tests were carried out by different groups of students that only were provided with the basic Experimental Framework, the model specification, and the object code for the models (not having the chance to know the internal behavior of the models). The results of this testing phase were returned to the original developing teams, who changed the models having problems.

Once the individual testing was finished, the integration tests were carried out. The first tests were carried out by the developers of the Control Unit model, running sample programs developed in assembly language, and using them as initial feed for the simulation. After, a complete set of tests was built, in order to provide future developers of the tool with a way of testing the external behavior. In this case, the simulated computer model operates on a finite number of inputs that can be interpreted as a binary bit stream. A complete functional test would consist of subjecting the program to all possible input streams, defining the desired behavior to be obtained in each case. For each input the model would either accept the stream and produces a correct outcome; accepts the stream and produces a incorrect outcome; or reject the stream. Because the rejection message is itself an outcome, the problem is reduced to verifying that the correct outcome is produced for every input. This approach was employed for many of the individual components, which have a bounded number of inputs and outputs. Nonetheless, even a short executable program of 10 bytes length has 2^{80} possible input streams and corresponding outcomes. So complete functional testing in this sense is impractical. For this reason we have divided the inputs in classes, and selected some representations of each class to test it. If for a given class we find a representation that fails to pass the test, we found a bug over this class of inputs.

We have chosen to use the instruction set as the source of information to divide the inputs. We have built a set of programs executing only one instruction. For example, a program using the ***add*** instruction taking two registers and storing the result into a third one. There are different subclasses for each class, for example we can execute this instruction using the registers *%r1*, *%r2* and move the result to the *%r3* register. Another kind of subclasses would involve the use of the carry flag. Once the individual classes were tested, combination examples were executed, in order to check interaction influences.

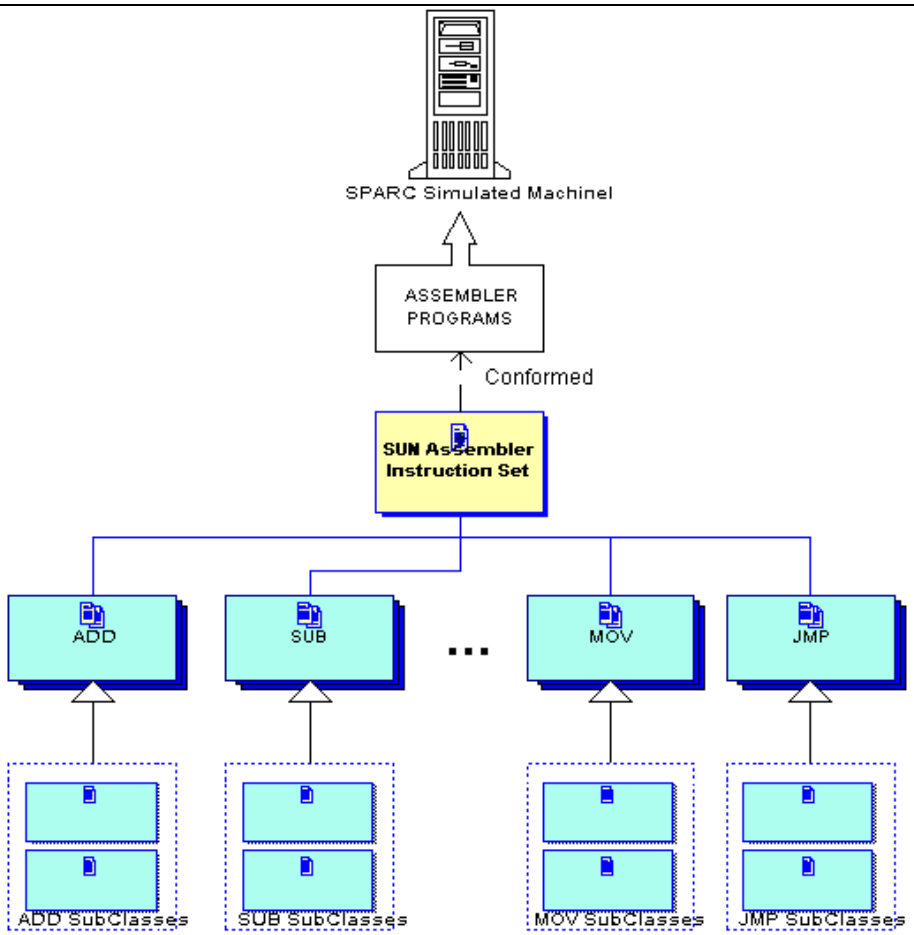


Figure 17. Classes and SubClases in a Simulated Computer test

Considering the SPARC instruction set, we have defined define the following test classes (we show some examples of each class test).

Store
ST (STORE)
ST REG, [ADRRESS] ∀ REG

MOV (MOVE)
MOV REG_a, REG_b ∀ REG_a, REG_b (REGISTER TO REGISTER)
MOV CONST, REG ∀ REG (MOVE A CONSTANT TO A REGISTER)
SET
SET LABEL, REG ∀ REG

Increment, decrement, add, subtract and jumps
INC, DEC, SUB, ADD, SUBCC, ADDCC, Bxx
INC CONST, REG ∀ REG, CONST={0,FFFFFFFF,..}
DEC CONST, REG ∀ REG, CONST={0,FFFFFFFF,..}
ADD REG, CONST, REG ∀ REG, CONST={0,FFFFFFFF,..}

ADD	REG, REG, REG	∀ REG, CONST={0,FFFFFFFF,..}
ADDCC	REG, CONST, REG	∀ REG, CONST={0,FFFFFFFF,..}
ADDCC	REG, REG, REG	∀ REG, CONST={0,FFFFFFFF,..}
SUB	REG, CONST, REG	∀ REG, CONST={0,FFFFFFFF,..}
SUB	REG, REG, REG	∀ REG, CONST={0,FFFFFFFF,..}
SUBCC	REG, CONST, REG	∀ REG, CONST={0,FFFFFFFF,..}
SUBCC	REG, REG, REG	∀ REG, CONST={0,FFFFFFFF,..}
Bxx		∀ Bxx, CONST1 {<,=,>} CONST2

Example for each Bxx:

```
                MOV    CONST1, REG1
                MOV    CONST2, REG2
                CMP    REG1, REG2
                Bxx    LABEL_YES
LABEL_NO:       MOV 0, REG3
                BA LABEL_END
LABEL_YES:      MOV FFFFFFFF, REG3
LABEL_END:      ST REG3, [RES]
```

Load and Store

LD, ST		
LD	REG, [LABEL]	∀ REG
LD, ST		∀ REG1, ∀ REG2, CONST en [0..n]

Example for LD and ST:

```
                LD     REG1, [LABEL]
                MOV    CONST, REG2
BEGIN:          ST     REG1, [LABEL], REG2
                LD     REG1, [LABEL], REG2
                DEC    REG2
                CMP    REG2, %G0
                BNE    BEGIN
```

EXCLUSIVE OR and AND Operations

XOR, AND, OR, ANDN, XORcc, ANDcc, ORcc, ANDNcc	
XOR, AND, OR, ANDN, XORcc, ANDcc, ORcc, ANDNcc	∀ Bxx, ∀ REG1, ∀ REG2 CONST1 {=, <>, <, >, = not } CONST2

Example for each OP:

```
                MOV    REG1, CONST1
                MOV    REG2, CONST2
                OP     REG1, REG2
                Bxx    LABEL_YES
LABEL_NO:       MOV REG3, 0
                BA LABEL_END
LABEL_YES:      MOV REG3, FFFFFFFF
LABEL_END:      ST REG3, [RES]
```

Shifts

SLL, SRL, SRA	∀ REG1, ∀ REG2, CONST2 en [0..31]

Example:

```
MOV             CONST1, REG1
OP              REG1, CONST2, REG2
ST              REG1, [ORIGIN]
```

```
ST          REG2, [DESTINATION]
```

MUL, MULScc, UMUL, UMULScc, DIV, DIVScc, UDIV, UDIVScc " CONST1/2, REG1/2/3

Example:

```
MOV         CONST1, REG1
MOV         CONST2, REG2
OP          REG1, REG2, REG3
ST          REG1, [A]
ST          REG2, [B]
ST          REG3, [RESULT]
```

For each of the classes, we used a black box testing approach, based on the construction of an oracle [HOW81]. An oracle is any program, process, or body of data that specifies that the expected outcome of a set of tests as applied to a tested object. In our case we used an input/outcome oracle approach, that is, an oracle that specifies the expected outcome for a specified input has occurred. The oracle is based on the execution of existing programs generated using assembly language. For each of these programs, we know the results. For example if we have the following source code, and we assemble it and execute in a SPARC processor or in the simulated computer, the result should be 5h in the register %r3.

```
[sth]
! The sum between registers r1 and r2  and we keep the result in memory

set 2, %r1            !Set the register r1 with 10
set 3,   %r2          !Set the register r2 with 5
add %r1, %r2, %r3     ! Add register r1 and r2 and store the result in r3
st %r3, [dest]        ! Move the result to the memory

unimp
.align 4
dest: .word FFFFFFFF
```

Figure 18. Add two registers and store the result in memory

When each of the programs belonging to the different classes of operations were created, we generated their source code, computed the results, and stored them. Then, we executed the assembler in the simulated machine and compared these results with the expected ones. We generated about 100.000 assembly language source programs, with the corresponding expected results. Each of them includes combinations of operations is a combination of all of the classes proposed.

For instance, if we want to test the instruction *BEQ* (JUMP EQUAL), we must define examples that represent the *BEQ* class of tests. In this case we used 279 different cases. The assembler code sets two registers with random values and then compares them.

In this way, we can predict that the result is the jump to a label, in that label an instruction sets on memory a result, and we check in the dump to find that result.

```
        set 82,  %r26                !Set the register 26 with 82
        set 543854, %r1              !Set the register 1 with 543854
        cmp %r26, %r1                !Compare the registers
        beq LABELYES    (1)          !If they are equal save the result in memory
        NOP
        LABELNO:        mov 0, %r10   (2)     !if the numbers are not equal save the result in memory
                        ba LABELEND   (3)
                        NOP
        LABELYES:       mov 4294967295, %r10  !Equal set FFFF FFFFh
        LABELEND:       st %r10, [dest] (4)        !Save the result in memory

        unimp

        .align 4
        value:   .ascii "VALUE:"     !The tester will look for the label VALUE at the memory dump
        dest:    .word  FFFFFFFF     !Result of the test
```

Figure19. Assembler File beq1.s

In this example the registers are not equal, therefore, we expect that the result is 0 at the `[dest]` (memory address location) in the memory dump. In (1), the result of comparing both numbers does not result in a jump, as the numbers are not equal. In the following operation (2) we set the register 10 with 0, and then we execute (3) jump to **LABELEND** and the result is stored in memory at (4).

At the same time we create a test file, in which we include `VALUE:int32,0`, meaning that we expect an int32 number with value 0 at the memory dump. The expected result file includes register with the following format: `VALUE:`*`data type, integer number.`* After executing the tester we obtain the result: `File:beq1. TEST OK` at the tester log file. After running all the tests, a table is generated with the following contents:

Test Number	Test Class	Test Subclass (if any)	Number of tests	Number of successful tests	Bug file
1	ADD	Add with carry	100.000	100%	
		Add two negatives	100.000	100%	
		Add two positives	100.000	100%	
		Add neg and pos	100.000	100%	
		Add to zero	100.000	100%	
2	BEQ	Numbers are equal	200.000	100%	
		Numbers are not equal	200.000	100%	
3	DIV	Negative numbers	100.000	99%	Divr1-r2.s
…	…	…	…	…	…
N	INC	Force carry	100.000	100%	
		Normal inc	100.000	100%	

This procedure allowed us to find some errors derived of the coupled model. For instance, we found that the division instruction and two conditional jumps were not working properly. By tracing the execution flow of the programs, we found the source of the errors, which were fixed in less than 8 man/hours.

6. VISUALIZING THE EXECUTION OF THE SIMULATED COMPUTER

At present we have started developing a GUI in order to let the students to interact with the simulated computer. The idea is to let the users to check the system state at every moment. The interface must be adaptive, as if the simulated computer is changed, we sill will be able to conform the new components. To do so, we made use of the basic design of Alfa-1, which includes a coupled model of the Integer Unit (***iu.ma***). This file defines all the components and subcomponents used to simulate the machine. If a new component is added, it should be included in the ***iu.ma*** file. Therefore, the GUI can adapt to the new simulated machine and display the new internal components.

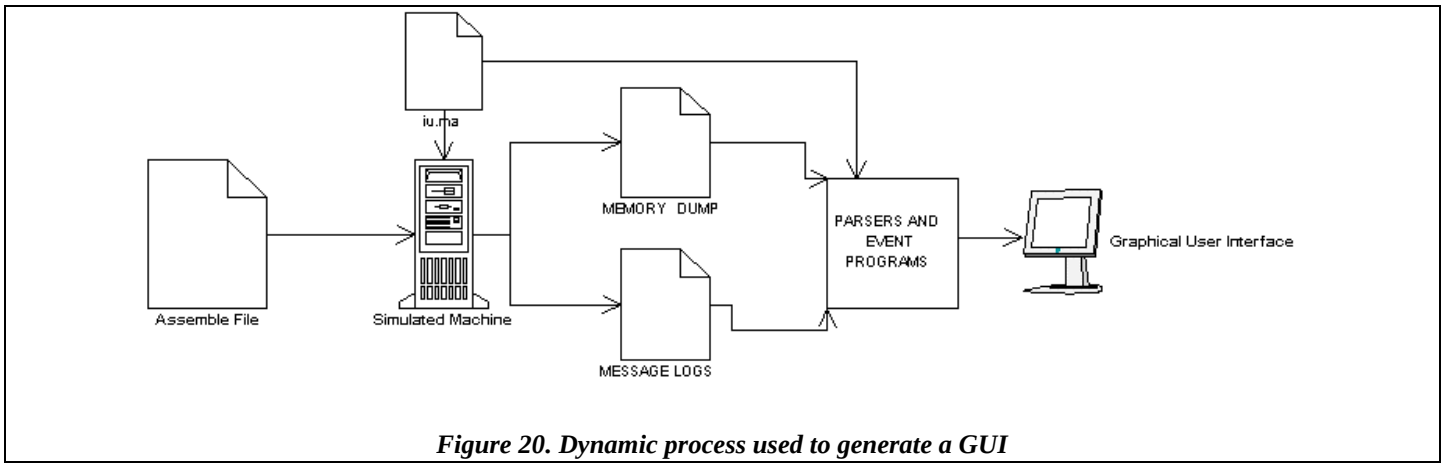


Figure 20. Dynamic process used to generate a GUI

As shown in section 3, the simulator generates a message log. The messages included in the log show how a program is executed. Each message has a source and destination and the change over the component, information that is used to analyze the present state of a component. The GUI makes two passes over the log files. In the first pass, it creates the models definitions with its ports and links. At the second pass the GUI resolves the components that we define previously and in that way a component non atomic reference another component.

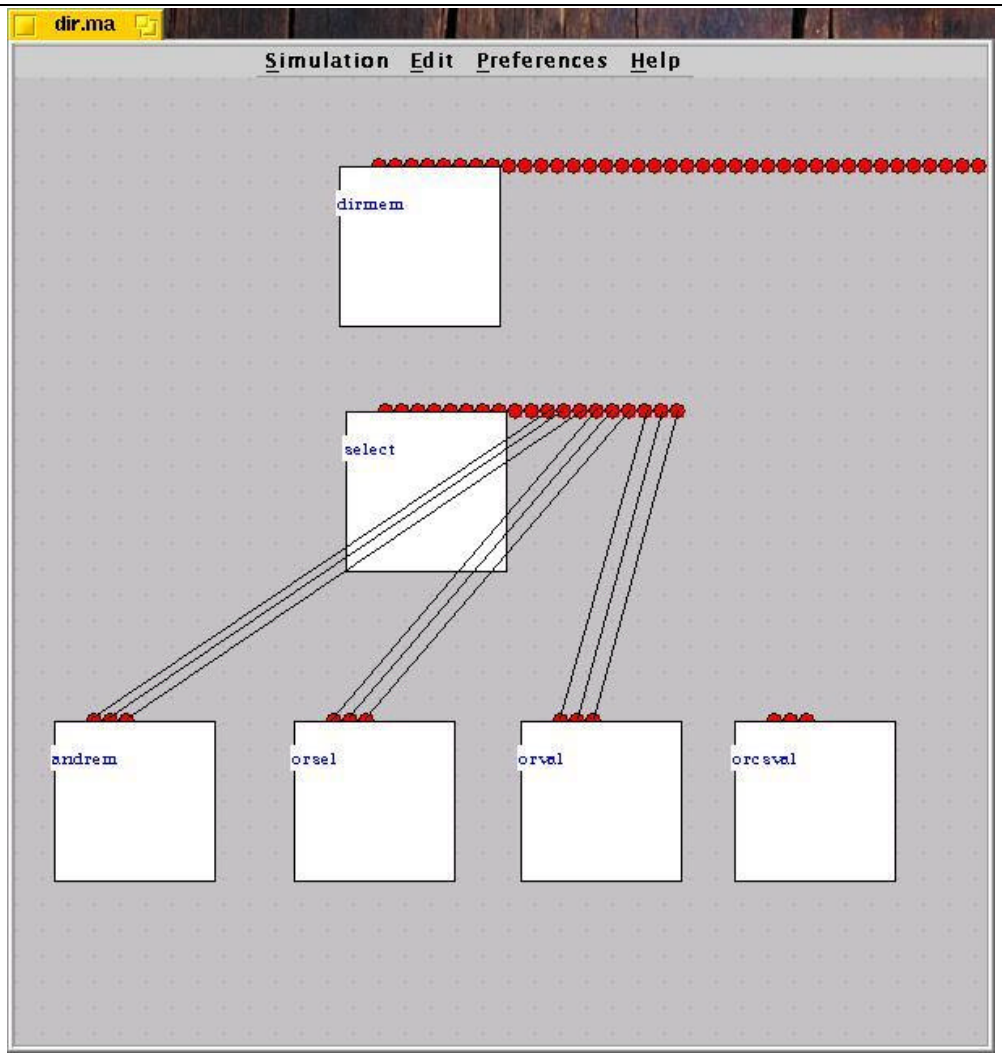


Figure 21. Snapshot of the GUI.

At present, the GUI is being finished by providing a debugger-like interface, letting the students to interact with the results in a friendly way.

7. CONCLUSION

We have presented the design of a simulated computer that can be used in Computer Organization courses to analyze and understand the basic behavior of the different levels of a computer system. The use of DEVS allowed us to have reusable models. We could test the feasibility of the approach, and several models were reused: the ALU, boolean gates, comparators, multiplexers, latches, etc. DEVS also allows us to provide reusable code for different configurations. We provided different levels of abstraction: the digital logic, the microcode and the instruction set levels, with different performance in each case, depending on the educational

needs. The interaction between levels can be studied, and experimental evaluation of the system can be done. The students can analyze in detail each of the layers by studying the behavior of the components belonging to the layer. As the simulator uses all the existing layers, the system behavior as a whole can be also studied. Starting with the instruction level, we can see how the instruction is carried out analyzing the activation flow in the datapath and the reaction of individual components.

The whole project was designed as an assignment in a 3rd year Discrete Event Simulation course, and the specifications were used by students in a Computer Organization course to build the final version of the architecture. These students had only taken previous prerequisite courses in programming. Final integration was planned by a group of undergraduate Teaching Assistants (which also developed the Control Unit and a coupled model representing the whole architecture showed in the Figure 1). Individual and integration testing was also done by 2nd year students. Several of the modifications showed here were developed as course assignments. These facts show the feasibility of the approach from a pedagogical point of view.

The use of this set of tools allowed the students to obtain a complete understanding of the computer organization. None of the problems existing in other tools occurred, and our educational goals could be achieved. Upper level courses reported higher success rates and detailed knowledge of the subjects after using Alfa-1. The code generated by GNU assembler/linker executed straightforward in our architecture. Nevertheless, the implementation of an assembler and linker are interesting assignments that can be faced to complete the layered view applied in these courses.

At present, the set of tools is being completed by including an input/output subsystem. The main input/output devices, the input/output interfaces, DMA controllers and channels will be simulated. Different transference techniques (polling, interrupts, DMA) will be considered to implement the input/output electronics. Other tasks faced at present include finishing the definition of a graphical interface, allowing to improve our educational goals, and the implementation of different cache management algorithms.

REFERENCES

[BAB96] BURGER, D.; AUSTIN, T.; BENNET, S. "Evaluating future microprocessors: the Simplescalar tool set". Technical Report CS-TR-96-1308. University of Wisconsin-Madison, July 1996.

[BEI90] BEIZER, B. Software Testing Techniques, 2nd Edition. New York. Van Nostrand Reinhold. 1990.

[BGW00] BURNS, M.; GEORGE, A.; WALLACE, B. "Modeling and Simulative Performance Analysis of SMP and Clustered Computer Architectures". *Simulation.* February 2000.

[ER98] EDMONSON, J.; REILLY, M. "Performance simulation of an ALPHA microprocessor". *IEEE Computer*, May 1998.

[HJ97] HEURING, V.; JORDAN, H. Computer Systems Design and Architecture. *Addison-Wesley*. 1997.

[HOW81] HOWDEN, W. E. A survey of dynamic analysis methods. In Software Testing and Validation Techniques (Second Edition.(E. Miller and W. E. Howden, editors). New York: IEEE Computer Society Press, 1981.

[HP94] HENNESY, J.; PATTERSON, D. *Computer Architecture: a quantitative approach.* Prentice Hall International. 1994.

[Pat95] PATTERSON, D. *Computer Organization and Design: the Hardware/Software Interface*. 2nd. Edition. University of California, Berkeley. 1995.

[RBDH97] ROSEMBLUM, M.; BUGNION, E.; DEVINE, S., HERROD, S. "Using the SimOS machine simulator to study complex computer systems". *ACM Transactions on Modelling and Computer Simulation*. January 1997.

[Self1] _____. "Experiences in modeling and simulation of computer architectures in DEVS". ___. 2001.

[Self2] _____. "Architectural definition of the ALFA-1 simulated processor". ___. 1998.

[Self3] ____, “Definition of components for the ALFA-1 simulated processor”. ___. 1998.

[SFL92] SHANMUGAN, K.; FROST, V.; LA RUE, W. "A Block-Oriented Network Simulator (BONeS)." *Simulation*. February 1992.

[Sta96] STALLINGS, W. *Computer Organization and Architecture*. Macmillan, New York. 4th. Edition. 1996.

[Sun01] SUN MICROSYSTEMS. SPARC Assembly Language Reference Manual. http://docs.sun.com/. 2001.

[Tan99] TANENBAUM, A. S., *Structured Computer Organization*, 4th edition, Prentice Hall, New Jersey, 1999.

[WT01] WAINER, G.; TROCCOLI, A. "CD++ User Manual". http://www.sce.carleton.ca/faculty/wainer/celldevs. 2001.

[ZB97] ZAGAR, M.; BASCH, D. "Microprocessor Architecture Design with ATLAS." *IEEE Design and Test of Computers*. July 1997.

[ZPK00] ZEIGLER, B.; PRAEHOFER, H.; KIM, T. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press. 2000.

APPENDIX 1 - INSTALLATION IN UNIX SYSTEMS

Requirements

For the installation in systems UNIX, it should be necessary approximately 75 Mb in disk. The first step is to install the set of BinUtils developed by GNU. This is a collection of binary tools. The main ones are:

as - the GNU assembler. With this binary you can ensemble an assembler source file to an object file

ld - the GNU linker. With this file you can link the object file to generate a machine executable file.

Another set of utilities include:

addr2line - Converts addresses into filenames and line numbers.

ar - A utility for creating, modifying and extracting from archives.

c++filt - Filter for encoded C++ symbols.

gprof - Displays profiling information.

nlmconv - Converts object code into an NLM.

nm - Lists symbols from object files.

objcopy - Copies and translates object files.

objdump - Displays information from object files.

ranlib - Generates an index to the contents of an archive.

readelf - Displays information from any ELF format object file.

size - Lists the section sizes of an object or archive file.

strings - Lists printable strings from files.

strip - Discards symbols.

windres - A compiler for Windows resource files.

The BinUtils can be obtained at:

BinUtils GNU http://www.gnu.org/directory/binutils.html
BinUtils Manual http://www.gnu.org/manual/binutils/html_mono/binutils.html
BinUtils Download ftp://ftp.gnu.org/binutils

Installing the tools

A directory is needed to install the source files for the simulated computer. Another special directory is needed to store the BinUtils, assembler and linker. In this document we sill be referring to these using two directories with the following notation:

[alfa1-dir]: Directory for the Alfa-1 source code (i.e. /home/alfa1)

[binutils-dir]: Directory for the BinUtils (Assembler, compiler, etc) (i.e. /usr/local/bin/sun4)

The file alfa1-dist.tar.gz contains the sources. The **[alfa1-dir]** directory is used to unpack the source code of the simulated computer, by issuing the following command:

```
[user@machine [alfa1-dir]]$gunzip -c alfa1-dist.tar.gz | tar xvf -
```

Once this command has been executed a directory *alfa1* will be created on **[alfa1-dir]**. To compile the source code, we use the directory *./alfa1/source*, where a *makefile* is included. This file contains the information required to build the binaries. We have used the tools provided by GNU (version egcs 2.91.66 19990314). After building the executable file with the `make` command, the file **alfa1-simu** will be created in the directory **[alfa1-dir]**/alfa1/source.

In order to use the simulated machine we should feed the simulated machine with the programs. To generate executable files from the assembler source code, we use the GNU assembler and linker. In our case we use the BinUtils version 2-10. We must unpack the file in the directory **[binutils-dir]** by executing:

```
[user@machine [binutils-dir]]$gunzip -c binutils-2.10.tar.gz | tar xvf –
```

This will create the directory *binutils-X*. Once it is unpacked, the package must be configured to operate in our platform (in particular, it should generate code SUN4). In the directory **[binutils-dir]**/binutils-X, we must execute:

```
[user@machine [binutils-dir]/binutils-X.Y$./configure--target=sun4
```

This will generate code sun4, creating the appropriate *makefile* to build the binaries of the BinUtils. Once we have been able to configure it, we should compile the source code. We have to run the `make` command at the *./configure* directory (do not use *make install*, because it would change the system standard assembler and linker settle found in */usr/local/bin* and */usr/local/lib*). We recommend to create a symbolic link to the assembler and the linker from the original directories */as* and */ld*. For instance,

```
[user@machine [binutils-dir]/binutils-X.Y]$ln -s /usr/local/bin/alfa1-as [binutils-dir]/binutils-X.Y/[gas]/as
[user@machine [binutils-dir]/binutils-X.Y]$ln -s /usr/local/bin/alfa1-ld [binutils-dir]/binutils-X.Y/[gas]/ld
```

As a result, if we check the directory */usr/local/bin*, we will find

```
lrwxrwxrwx 1 [user] [user] 48 alfa1-as  - > [binutils-dir]/binutils-X.Y/[gas]/as
lrwxrwxrwx 1 [user] [user] 47 alfa1-ld  - > [binutils-dir]/binutils-X.Y/[gas]/ld
```

Assembling source code

As explained in section <>, a testing workbench is included. The set of testing programs are located in the directory **[alfa1-dir]**/alfa1/tests. Let us consider one of those examples, called *store.s*

```
[sth]
Table 1: I File store.s
Sep 0x12345678, %r1
st %r1, [dest]
sth %r1, [dest+4]
sth %r1, [dest+10]
stb %r1, [dest+12]
stb %r1, [dest+17]
stb %r1, [dest+22]
stb %r1, [dest+27]
unimp dest: .ascii""
```

Figure22. Simple example to run on the simulated machine

The first pass is to compile the assembler file, we use the *alfa1-as* to compile the source file *store.s* and generate the object file:

```
[user@machine [alfa1-dir]/alfa1/tests]$alfa1-as store.s -o store.ens
```

Then the linker *alfa1-ld*:

```
[user@machine [alfa1-dir]/alfa1/tests]$alfa1-ld store.ens -o store.map -Ttext 0x20
```

Once we have created the *.map* file, we should configure the machine to use memory of size *x* bytes, loading this file (*store.map*) as input. For that we must edit the file *iu.ma* that it is inside the directory **[alfa1-dir]**/alfa1/source/coupled/alfa1. In this file we defined:

```
…
[mem]
preparation: 0:0:0:50
memsize: 256
```

```
memfile: call.map
dumpfile: memory.dmp
…
```

Figure 23. Editing the memory source in the iu.ma file

where *memsize* is the size of our memory in the simulation (256 bytes by default), the input file (in this case *store.map*) and a *dumpfile* file to store the results of the simulation after exit, for defect memory.dmp.

Once we have changed these values, the simulation can be executed. The executable file *alfa1-simu* is located in the directory **[alfa1-dir]**/alfa1/source/.

```
[user@machine [alfa1-dir]/alfa1/source]$alfa1-simu -m[alfa1-dir]/coupled/alfa1/iu.ma –o
```

-m**[alfa1-dir]**/coupled/alfa1/iu.ma refers to the location of the file *iu.ma* where we define the coupling of the simulated machine

–o generates an standard output of the PC (program counter) to see the progress of the simulation.

As a result of the simulation we obtain the file *memory.dmp* with the state from the memory when concluding the execution, this file will have the size in bytes that we had used at [MEM] definitions.

APPENDIX 2 – COUPLED MODEL SPECIFICATION PREPROCESSOR

As coupled models grow in compexity we can observe that the specification file described on section 4.2 becomes less and less readable, furthermore the impossibility to split one coupled model specification file into several smaller files lends to maintainability pronlems too.

In order to find a solution for the previously presented problems we've developed a preprocessor for the coupled model specifications. The tool was thought focusing on the following problems:

- Massive indexed repetition of terms: Suppose that we need to declare 31 links between two components whose names only differ on the link number as we can see on figure 11, in the standard specification file we will be forced to write each link.

- Coupled model inclusion: Suppose that we have two complex coupled models, the only way in the standard specification file to develop a third coupled model by joininig the first two will be by writing a third file which will include the complete specification for the first two models, changing the [top] model on each one and then write the third model.

- Comments: The possibility to include comments on the coupled model specifications.

The preprocessor is a command line tool which receives as parameter the name of a source extended syntax coupled model specification file and the name of the destination standard syntax coupled model specification file described on section 4.2. for instance

```
[user@machine [coupled-model-dir]]$ writema source_file destination_file
```

The simple syntax extension is the comment inclusion, any line preceded by a `//` will be considered a comment line

```
//THIS IS A COMMENT
components: samplela

Link: OUT_A2@bus A2@samplela
...
Link: OUT_A31@bus A31@samplela
```

Figure 24. The first 31 lines of figure 11 including a comment

For massive indexed repetition the preprocessor provides two constructs, the first one is

`#simple term # index_start index_end`

where `term` is the term to be repeated and `index_start index_end` denote the start and the end of the index., thus taking the first three lines of the figure 11:

```
components: samplela

Link: OUT_A2@bus A2@samplela
...
Link: OUT_A31@bus A31@samplela
```

Figure 25. The first 31 lines of figure 11

the equivalent using the extended syntax would be:

```
components: samplela

#simple Link: OUT_A%d@bus A%d@samplela # 2 31
```

Figure 26. Equivalent for the first 31 lines of figure 11 with the preprocessor syntax

Where the `%d` will be replaced for each number between 2 and 31 inclusive on the destination file thus the destination file will be exactly as seen on figure 25.

As we can see the previous construct solves well the problem on the first 31 lines but it would be useless if we changed the specification as follows:

```
components: samplela

Link: OUT_A2@bus A0@samplela
...
Link: OUT_A31@bus A29@samplela
```

Figure 27. Modified specification for the first 31 lines of figure 11.

In order to perform these repetition of terms the preprocessor provides the following construct

```
#double term # index1_start index1_end index2_start index2_end
```

thus the extended syntax that once preprocessed would produce the same specification would be:

```
components: samplela

#double Link: OUT_A%d@bus A%d@samplela # 2 31 0 29
```

Figure 28. Equivalent using the extended syntax for the specification on figure 26.

Finally the extended syntax also includes a construct to join several coupled model specifications.

```
{ model_name file }
```

The `[top]` model on the extended syntax coupled model specification file `file` will be inserted as a [model_name] model on the destination in the destination coupled model specification file. If we split the two models specified on figure 7 into two files `consumer.cpe` for the consumer and `main.cpe` for the main file:

```
[top]
components : Qu@Queue Proc@CPU
in : in
out : out
Link : in in@qu
Link : out@qu in@Proc
Link : out@Proc done@qu
Link : out@Proc out
```

Figure 29. Contents of the file consumer.cpe

```
[top]
components : Transd@Transducer Gen@Generator Consum
Out : out
Link : out@generator arriver@transducer
Link : out@generator in@Consumer
Link : out@Consumer solved@transducer
Link : out@transducer out

{ Consum consumer.cpe }
```

Figure 30. Contents of the file main.cpe

Then the command

```
[user@machine [coupled-model-dir]]$ writema main.cpe main.cp
```

will produce a file `main.cp` with the same specification seen in figure 7.

INSTALLING THE PREPROCESSOR

After performing the installation described on APPENDIX-1 the source code for the preprocessor will be located in the directory **[alfa1-dir]**/alfa1/tools. Then the only steps required for the installation are:

```
[user@machine [alfa1-dir]]$ cd alfa1/tools
[user@machine [alfa1-dir]]$ make install
```