Top-Down Modeling of RISC Processors in VHDL

Hsiao-Ping Juan
Nancy D. Holmes
Smita Bakshi
Daniel D. Gajski

Technical Report #92-96
October 5, 1992

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

hjuan@ics.uci.edu
nholmes@ics.uci.edu
sbakshi@ics.uci.edu
gajski@ics.uci.edu

Abstract

In this report, we present a top-down VHDL modeling technique which consists of two main modeling levels: specification level and functional level. We modeled a RISC Processor (RP) in order to demonstrate the feasibility and effectiveness of this methodology. All models have been simulated on a SPARC 1 workstation using the ZYCAD VHDL simulator, version 1.0a. Experimental results show feasibility of the modeling strategy and provide performance measures of RP design features.

Contents

List of Figures

List of Tables

1 Introduction

For years, CAD tools have been developed to support the design process. Most of these tools are intended for use at low levels, to eliminate some of the difficulty in tasks such as schematic capture, circuit layout, and logic design [1, 6]. While low-level CAD tools have certainly revolutionized IC design, the need for some organized, higher-level design methodology remains unsatisfied.

Development of a modeling technique for the high-level design process provides several benefits. First, system specifications can be documented using a high-level model so that all participants in the design process will have access to them at all times. Secondly, the specification level model can be simulated within its intended environment to ensure that it is feasible to meet performance and functionality requirements. Thirdly, several target architectures for the design can be tested by means of modeling and simulation, and the "best" one in terms of predefined quality metrics is selected. Finally, high-level design models may be used to balance the design so that all critical paths have nearly the same delay. After choosing an architecture and verifying timing, existing low-level design tools are used to implement the design.

In this report, we describe a modeling methodology for the top-down design process. It is hoped that this methodology will form the basis for a CAD environment, capable of supporting both system-level and chip-level designs. The methodology consists of four basic levels: the specification level, the functional level, the register-transfer level, and the layout level. As design modeling and synthesis at the register-transfer and layout levels are well-studied problems, we focus on specification and functional levels. At each of these levels, models of the unit under design (UUD) are developed and simulated to verify cost and performance constraints. We demonstrate this modeling methodology on a RISC Processor (RP) design.

Section 2 provides a brief description of the RP example, and Section 3 contains an overview of the modeling methodology. Sections 4 and 5 discuss the specification and structural level modeling steps. Section 6 describes simulation results for the design models (at various stages in the design process) on the Livermore Loop benchmarks, and Section 7 contains concluding remarks.

2 RISC Processor Example

This section gives a brief description of our example. We shall first present an overview of the architecture and then discuss the execution flow in the processor.

2.1 Architecture

RP is a 32-bit processor which contains a data memory, a program memory, an ALU, registers, and a controller, as shown in Figure 1. Its features include a Harvard architecture which permits simultaneous data and program memory accesses in each instruction cycle, a large register address space which allows fast register-to-register access for more operands, and special purpose registers which enhance array data transfer both to and from the data memory. The RP also provides a simple hardware mechanism for automatic loopback at the end of an inner loop based on loop length and loop count.

The register space (RS) of RP contains 256 32-bit registers, divided into two blocks of 128 registers each. The first block, called data registers, consists of vector access registers (VARs), accumulators, and registers for temporary operand storage. The second RS block, called the interface registers, consists of less frequently used control and I/O registers. The memory space (MS) of RP contains internal and external memory. Internal memory includes program memory and data memory. In our model, external memory is restricted to data RAM, though it can be extended to include program memory as well. The internal data memory and the external data memory share the same address space, and they are accessed via the same address and data buses.

The ALU performs arithmetic, logic, and bit-manipulation operations. It is supplied with two operands from RS and, depending on the instruction, the operands are directed to the adder-shifter or multiply-accumulate unit. The result of the operation is returned to RS.

The controller generates signals to initiate transfer of data via the buses, control the operation of ALU, and store data into registers or memories, according to the instruction currently stored in the instruction register (IR) as well as the status word.

The I/O features of RP include a serial receiver/transmitter port (SRT interface), and an 8-bit parallel port (host interface). The host interface consists of a 32-word$\times$8-bit circular buffer. The circular buffer can be simultaneously accessed by RP and the host device. Typically, while one device is writing to the circular buffer the other could be reading from it. The SRT interface consists of a controller, a transmitter and a receiver. The transmitter and the receiver both contain a 16-bit buffer. The transmitter can be transmitting data to the peripheral device while the receiver is receiving data.

In addition, clock divider and timer units are provided. The clock divider generates pulses by dividing the frequency of the original RP clock according to a user-specified parameter. The timer counts clock pulses and sends an interrupt signal to the controller when a pre-specified number is reached.

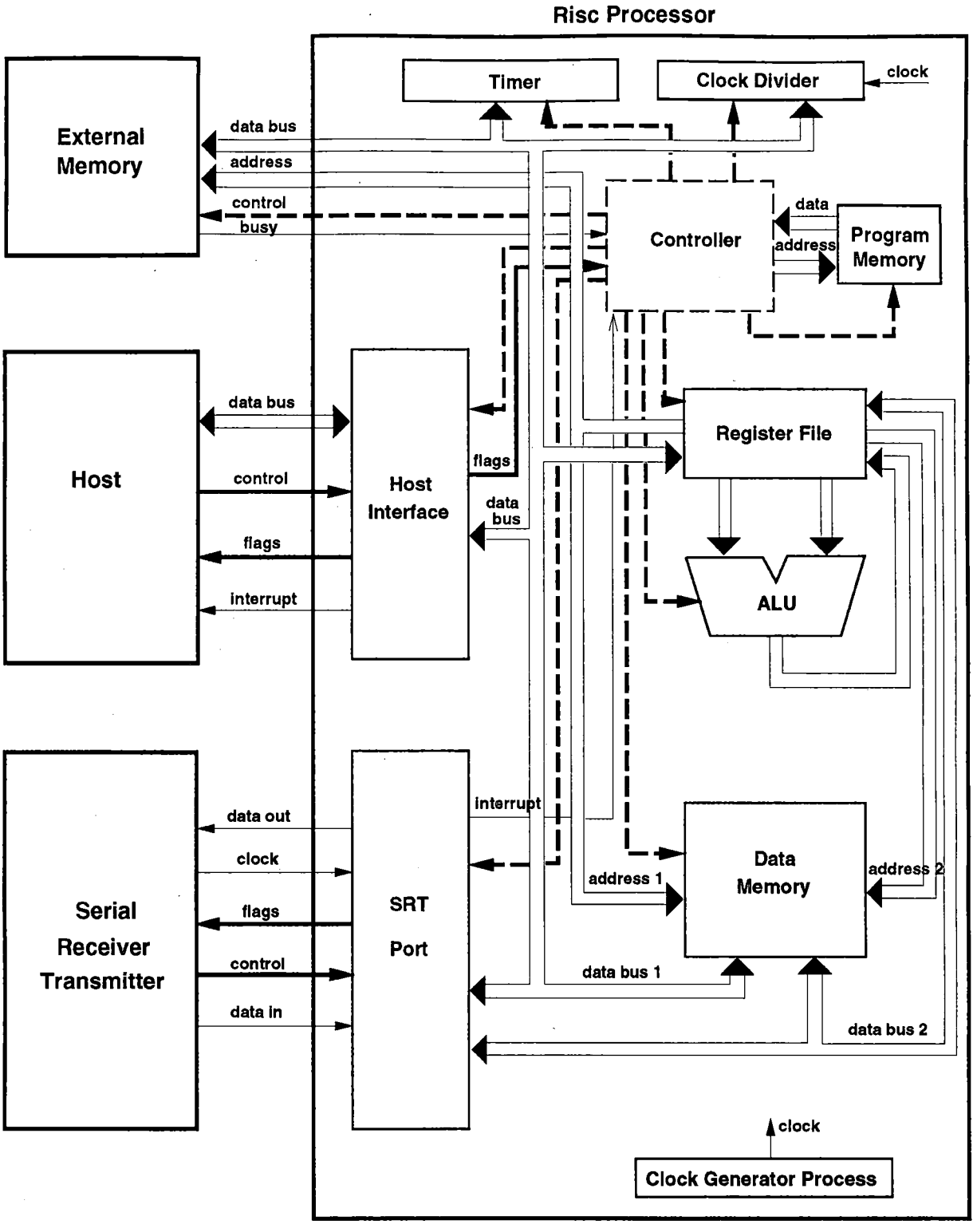

Figure 1: Block diagram of RISC Processor.

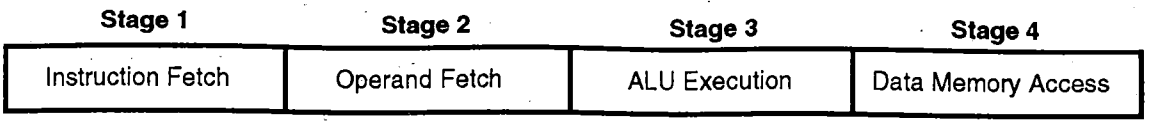

Figure 2: RP instruction execution.

2.2 Instruction Set

The instruction set of RP consists of arithmetic, logical, program flow, interface, and other specialized instructions. Three addressing modes, *immediate, direct* and *indirect*, are provided. Examples of the various instruction types are as follows:

1. **ALU instructions:** add/subtract, shift, multiply, and logical operations.
2. **Data transfer instructions:** load from register to memory, store from memory to register, move from register to register.
3. **Program control instructions:** branch, jump, call, return.
4. **Interface instructions:** set/reset/check interface signals, read/write interface registers.
5. **Special purpose instructions:** VAR setup, automatic loopback.

The RP instruction word consists of an opcode, three sources, a destination, immediate operands, and various options. The number and types of the sources, destinations, immediate values, and options may vary from instruction-to-instruction since the RP has a variable instruction word format.

2.3 Program Execution

The RP initiates program execution by initializing the program counter (PC) to the address of the first instruction. The instructions are then fetched from program memory and executed sequentially. Instruction execution consists of four stages:

1. stage1: instruction fetch,
2. stage2: operand fetch from registers,
3. stage3: ALU execution, and
4. stage4: data memory access (load/store/VAR operations),

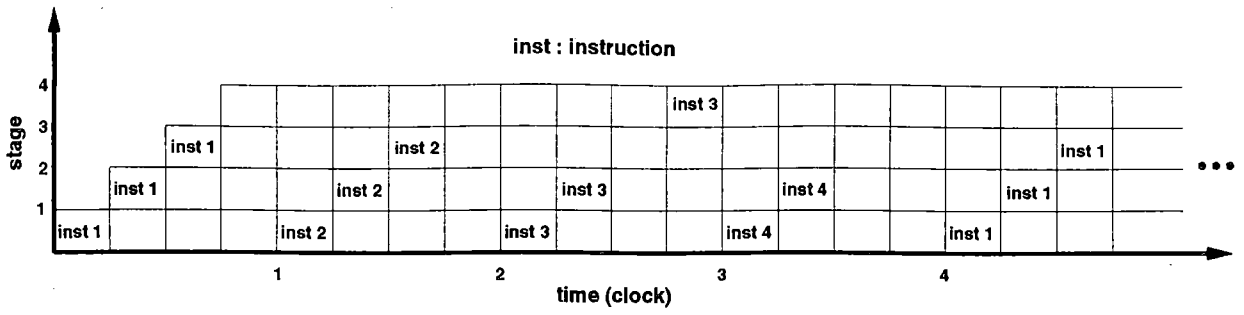

Figure 3: Example of RP program execution (if branch is taken).

as shown in Figure 2.

For an example, consider the execution of the ALU instruction shown below.

$$\text{ADD\_SUB\_IMMED}, \quad R5, 50, R5; \qquad (R5 \leftarrow R5 + 50)$$

The instruction is fetched from program memory in Stage 1, and the source operand is fetched from register $R5$ in Stage 2. During Stage 3, the addition is performed and the result is returned to $R5$. Since this instruction does not involve memory operations, Stage 4 is not used.

Execution of a memory access instruction such as

$$\text{STORE\_D}, \quad R5, 1; \qquad (data\_memory[1] \leftarrow R5)$$

is described as follows. The instruction is fetched in Stage 1, while the operand is fetched from $R5$ in Stage 2 and stored to data memory location 1 in Stage 4. Stage 3 is not used since no ALU operation is needed.

In general, instructions are executed sequentially at the rate of one instruction per clock. However, in the case of program control instructions, such as jumps, branches, calls, and returns the execution sequence may be altered. For example, if the instruction fetched is conditional (branch), the branch decision is performed during the second stage of the instruction execution, and the PC is updated with the value indicated in the instruction word.

To illustrate the execution of a small program in the RP, part of the Livermore Loop benchmark is taken as an example.

instruction 1: ADD_SUB_IMMED, $R5, 50, R5;$	$(R5 \leftarrow R5 + 50)$
instruction 2: MULT_REG_WORD, $R1, R2, R3;$	$(R3 \leftarrow R1 \times R2)$
instruction 3: STORE_I, $R3, R4;$	$(data\_memory[R4] \leftarrow R3)$
instruction 4: BRANCH_IMM, $R5, 151, -3;$	$(if R5 = 151, execute$ **instruction 5**; $otherwise, execute$ **instruction 1**$)$
instruction 5: LOAD_D, $R1, 3;$	$(R1 \leftarrow data\_memory[3])$

Figure 3 gives the graphical illustration of program execution in the case where $R5 \neq 151$.

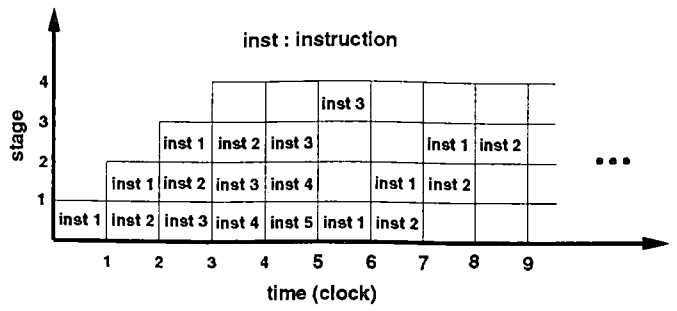

Figure 4: Pipelined RP instruction execution (if branch is taken).

We also introduce pipelining into RP. In the pipelined design, each instruction is executed in the four stages described above. One instruction is launched at each clock, and it progresses through the pipe at the rate of one stage per clock. The first clock is used for instruction fetch, the second for operand fetch and branch decisions, the third for ALU operations, and the fourth for memory accesses. The frequency of the clock in the pipelined design is four times greater than the clock in the non-pipelined design.

Note that a cycle may be wasted if an instruction requires the result of a previous instruction as a source operand. The RP provides special bypass interconnections which make the result available to the source operand before it reaches its destination. In the example above, instruction 2 uses $R3$ as the destination register, while instruction 3 uses it as the source register (Figure 4). The result of the ALU operation in instruction 2 is stored to $R3$ at the end of Stage 3, and instruction 3 fetches the operand from $R3$ in Stage 2. Due to the pipelined architecture, Stage 3 of instruction 2 and Stage 2 of instruction 3 are performed at the same time. The bypass hardware mechanism allows the ALU output to be selected as the ALU source operand for instruction 3 rather the contents of $R3$.

It should also be noted that since the branch decision is made during Stage 2, instruction 5 in Figure 4 is fetched into the instruction register at the time the decision is made. Therefore, if the branch is taken, this instruction is converted to a null operation.

3 Modeling Methodology and VHDL Implementation

In this section, we describe the basic modeling methodology used to design the 32-bit RISC processor introduced in Section 2. Modeling strategies in VHDL are also discussed.

3.1 Basic Modeling Methodology

The basic idea of our modeling technique is to iteratively refine a high-level design description down to the layout level by systematically developing a hierarchy of VHDL models. The modeling hierarchy is illustrated in Figure 5, and the RP example is used to explain

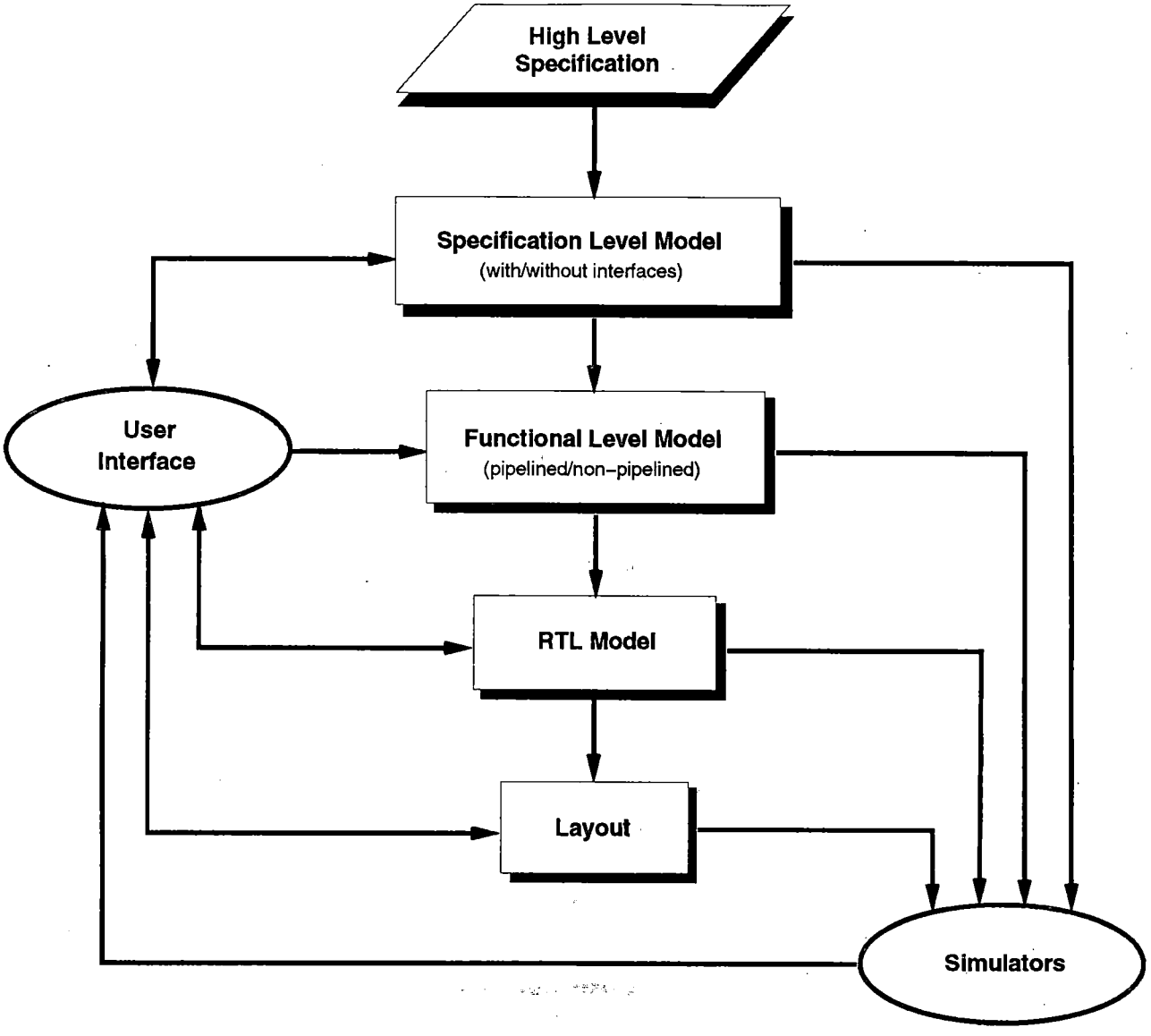

Figure 5: Modeling hierarchy.

each level.

Initially, the system specification is captured in a high-level description called the "specification level model". The designer then refines this model by adding architectural details until the design consists of functional blocks and interconnections. The resulting model is called the "functional level model". Functional blocks can be further refined to produce a register-transfer level (RTL) model, from which the layout may be synthesized.

As RTL modeling and synthesis tools are gaining acceptance, we focus on specification and functional level modeling only [2, 3, 4]. These models are discussed in greater detail in the following subsections, using RP as an example.

3.1.1 Modeling at the Specification Level

The specification level model (SLM) is a "coarse-grain" model of the system. At this stage of the design process, the modeler may have very little information about architectural details such as the number and width of internal buses, the number of ports on the memory, the number of pipeline stages etc. Since these details are unavailable, it is not possible to incorporate timing (such as functional unit delays, propagation delays on buses, memory access times etc.) in the model. It should be noted that the SLM may be rewritten several times if simulation reveals improper functionality. However, modeling time for SLM is typically a small percentage of the entire design cycle.

The SLM test environment must be developed together with test vectors to provide a mechanism for simulating and monitoring the unit under design. The same test vectors can be used to verify models at successive levels of refinement in the design process.

In the RP example, the initial specification consists of the instruction set and a set of protocols that define communication between the processor and its peripherals: host processor, external memory, and the serial receiver and transmitter (SRT).

Since specifics of the architecture are not available at this point, RP is modeled with a single VHDL process. In order to model concurrent behaviors, the process may contain several procedures that, although written sequentially, execute in zero time according to the simulation semantics of VHDL. Since the timer and clock divider units work concurrently they are modeled as separate procedures (Figure 6.)

The performance of the system is tested by executing benchmarks on the SLM. In the testing environment, we model three peripheral devices, also as processes without specified timing. The communication between RP and the three peripherals is assumed to be under the control of the processor and is, therefore, implemented using RP instructions. Based on simulation results, the designer may detect that communication is too slow. He can then introduce separate hardware interfaces between the RP and the peripherals, thus trading

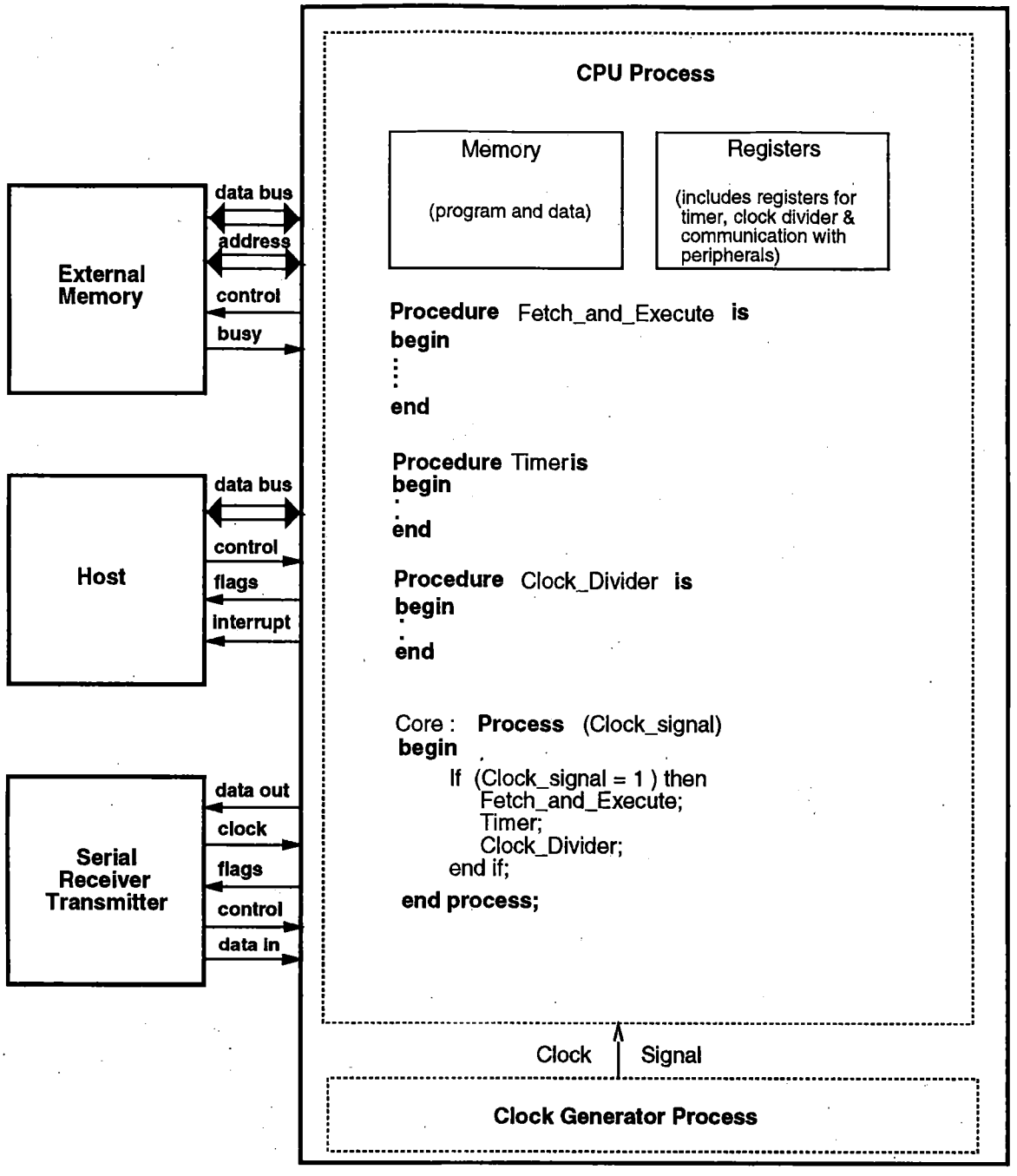

Figure 6: Conceptual View of the Specification Level Model

off software for hardware to improve performance.

In our example, the designer, for instance, can introduce two hardware interfaces: host interface and SRT interface. The communication is controlled by the hardware, but the RP has to set up the interface registers to start the communication. The instruction set is modified to accommodate for the setup. Two separate processes are used to model the hardware interfaces and the term "CPU Process" in Figure 7 identifies the remaining part of the original process. The internal logic of the interfaces is not shown in the block diagram as all interfaces are modeled at the behavioral level within the corresponding processes. Several storage elements, however, are depicted.

After this model is completed, the instruction set, interfaces, and the communication protocols are considered fixed, and ready for further refinement. Since the instruction set is steady at this time, this model can be given to the compiler design group for compiler validation. Timing of the communication interfaces can be incorporated, and the model can be used by customers embedding the design into their own systems.

Section 4 gives the VHDL details of the SLM model for our example.

3.1.2 Modeling at the Functional Level

In a functional level model (FLM), the design is modeled as a set of functional blocks, such as ALU, register files, or memory, that communicate via signals and buses. The main purpose of FLM is to verify timing and balance delay paths in the design. This cannot be done effectively with the SLM, which models the design as an abstract process with no architectural details. It is to be noted that a detailed timing description, including propagation delays for functional units and setup and hold time for storage elements, can be incorporated into FLM.

The FLM requires specification of architectural parameters such as number and size of buses, number of ports on memories/register files, number and type of functional blocks, number of pipeline stages (if any) in the design, etc. This model encourages the designer to vary these parameters and analyze the cost and performance of the system; therefore, the process of writing the FLM is also an iterative one. Furthermore, due to the increased level of modeling detail, writing an FLM is more time-consuming than writing the SLM.

After the FLM is completed, the basic structure of functional blocks and interconnections is known. Hence, different design groups may work concurrently on the functional blocks, thereby reducing the design time. The completed functional blocks may be easily integrated into the design since interfaces are also well-defined.

In our example, we decomposed RP into seven functional blocks: controller, register file, ALU, program memory (PMEM), data memory (DMEM), timer and clock divider, as

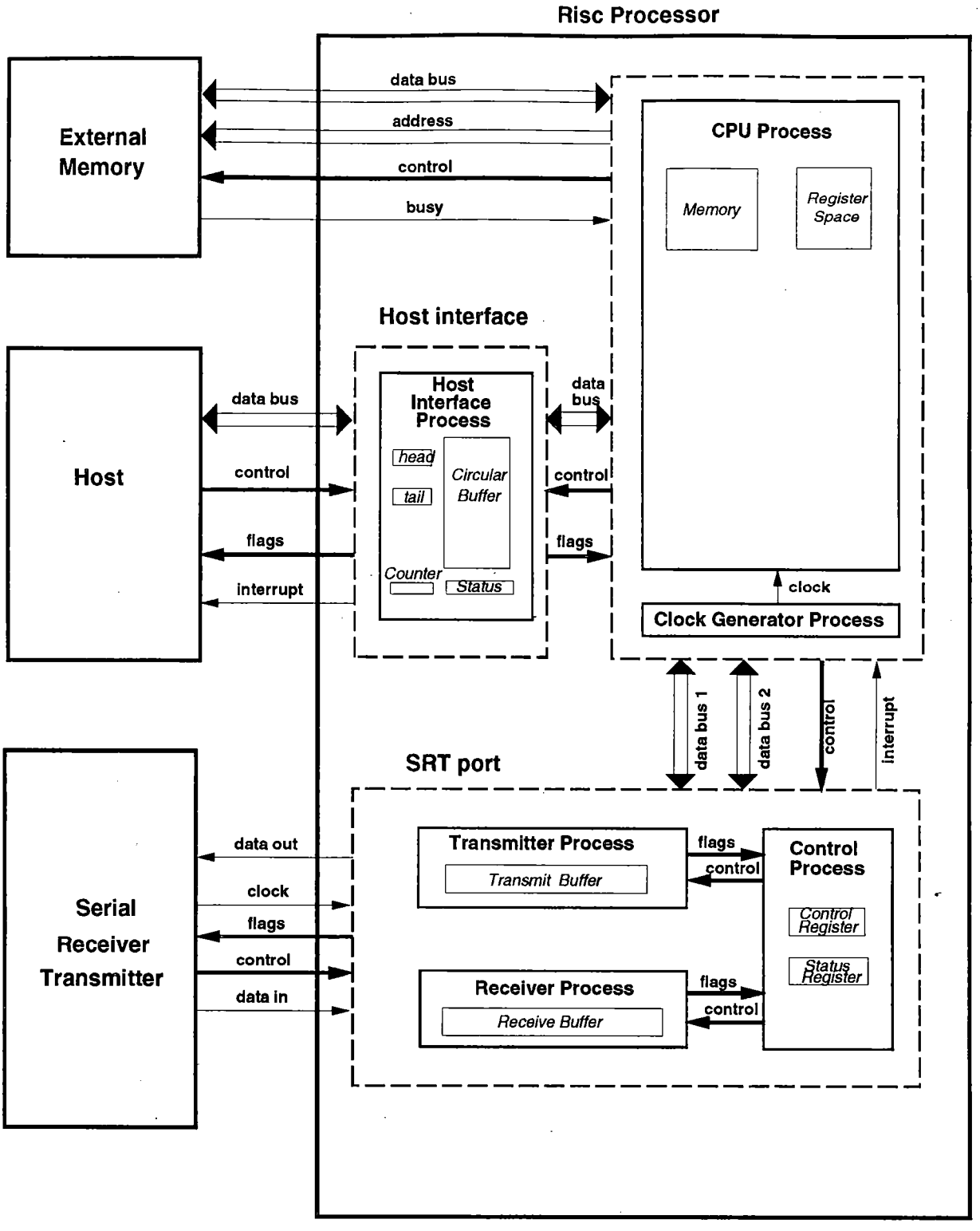

Figure 7: Refined Specification Level Model

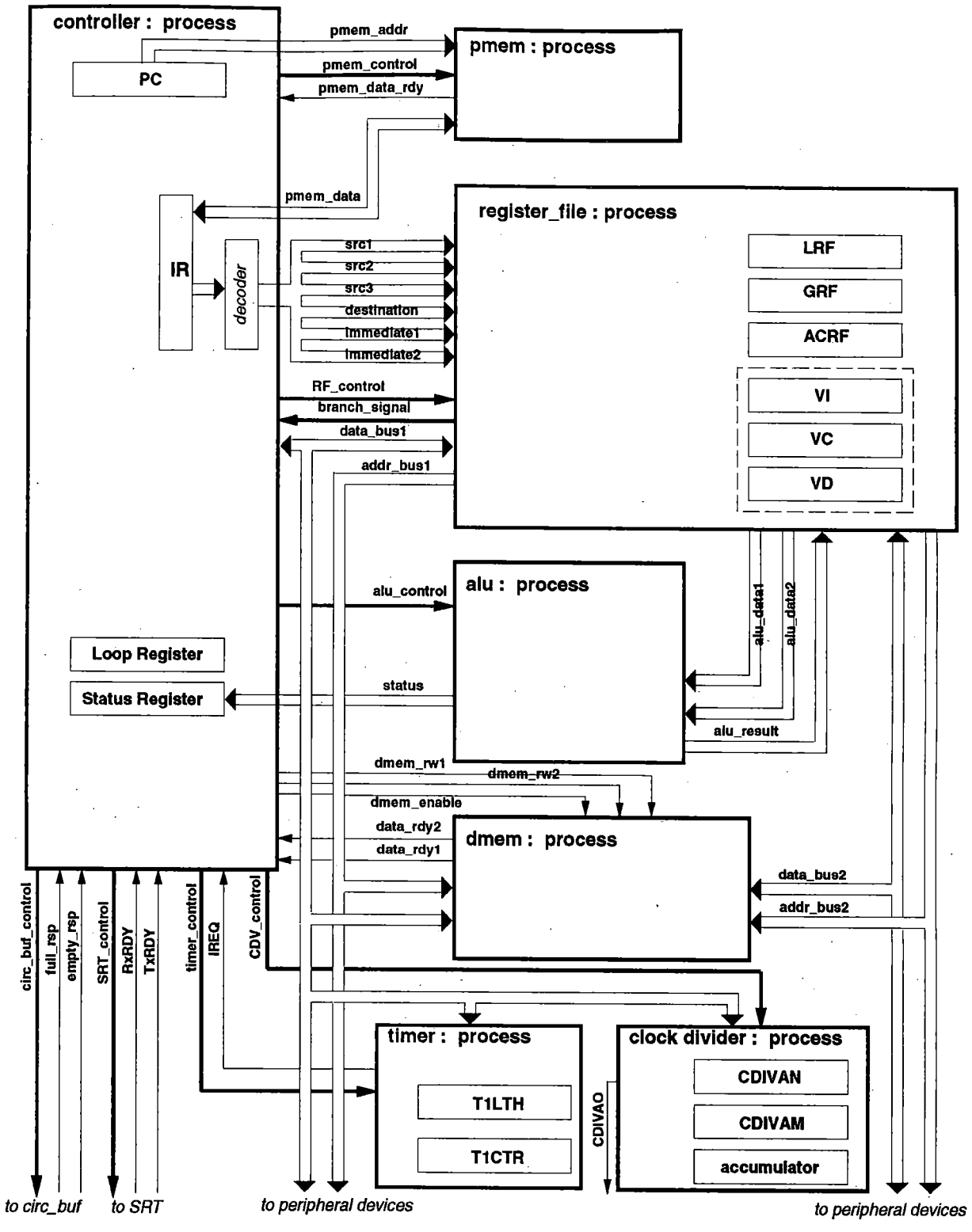

Figure 8: Block diagram for non-pipelined model.

shown in Figure 8. Note that architectural details such as busing structure and number of ports on memories and register files are also specified. For instance, the program memory has one port, while the data memory has two ports.

The functional blocks are modeled as processes, and delay values may be inserted to model timing behavior. Propagation delays for signals and buses may also be inserted, if desired. Simulation with the component delays can be used to detect critical paths in the design. In our case, this resulted in a 200 ns clock for the model. As this clock period is long, and the utilization of the functional blocks is low, we pipelined the design to improve the performance and component utilization.

The original non-pipelined FLM was converted into a four-stage pipelined model (Figure 9). The primary difficulty in constructing this new model, referred to as PM (Pipelined functional level Model, to be distinguished from NPM: Non-Pipelined functional level Model), is obviously how to describe the pipeline in VHDL. The VHDL details of both non-pipelined and pipelined structural level modeling are discussed in section 5.

3.2 Modeling in VHDL

In this section, we discuss the basic modeling strategy for specification level and functional level models in VHDL. VHDL was selected as our modeling language because it allows simulation over many different levels of abstraction and is widely accepted as a standard for modeling hardware [5, 7, 8]. In all our models, we follow the VHDL modeling conventions listed below.

1. High-level behaviors are modeled as *processes* or *sets of communicating processes.*
2. All the registers or buffers are modeled as *variables* within processes.
3. Processes communicate with each other using *global signals.*
4. Separate *processes* must be used to model different concurrent behaviors.

For example, the SRT port in RP consists of the transmitter, the receiver, and a controller. These components operate concurrently. For instance, the transmitter and the receiver can transmit or receive data at the same time without interfering with each other. Therefore, the SRT port is modeled as a block which contains three concurrent processes: control process, transmitter process, and receiver process, and these processes communicate with each other via VHDL *global signals.*

The VHDL details of the RP models are introduced in subsequent sections, and some pseudo codes are given as examples to explain these models.

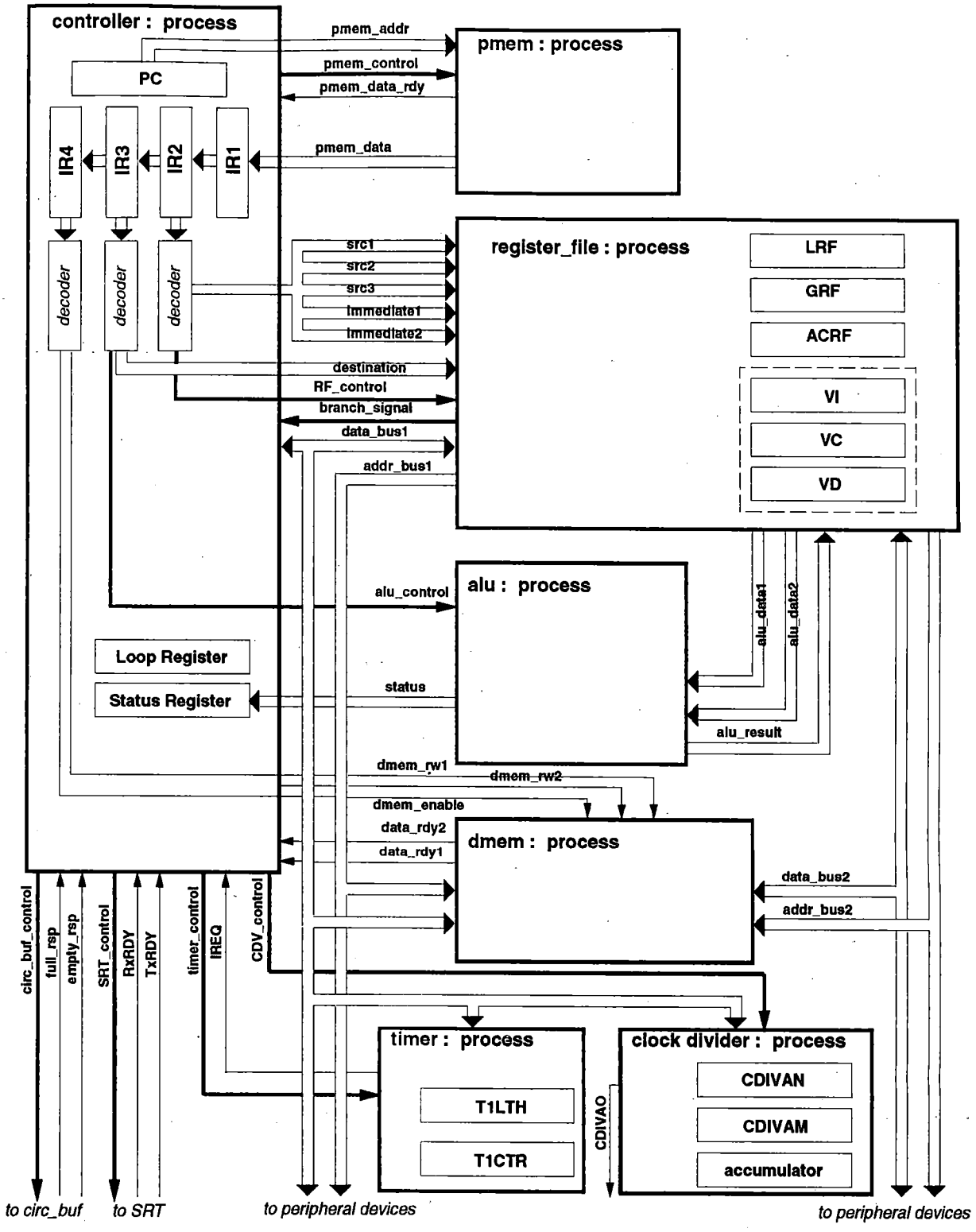

Figure 9: Block diagram for pipelined model.

4 Specification Level Model

The SLM models the system at a high level of abstraction, without incorporating architectural details of the design. For instance, neither functional blocks nor internal buses are specified in the model. In addition to utilization for functional verification and compiler validation, the SLM is used to make preliminary design decisions involving tradeoffs between hardware components and software subroutines. In this section, we explain the SLM of the RP with the help of pseudo VHDL code.

As mentioned previously, the SLM models the RP from an instruction execution point of view. Figure 10 outlines the initial VHDL architecture for the specification level model. It consists of two processes called the *clock_generator* process and the *cpu_1* process. Since the memory and register file are accessed only by *cpu_1*, they are modeled as global variables within this process. The register file includes data registers such as VARs and temporary storage registers, timer and clock divider registers, interface registers (for the host and the serial receiver transmitter) as well as control registers such as the program counter and the instruction register. Similarly, at this point, the memory is common for both program and data. Since the design is not partitioned into functional units, it is prudent to group storage in this manner. Both the memory and the register file have been modeled as arrays of integers as opposed to arrays of bitvectors, for the sake of readability and ease of testing.

The clock signal (of frequency 5 MHz), produced by the *clock_generator* process, triggers the *cpu_1* process. The *cpu_1* process models most of the RP behavior: ALU operations, timer and clock divider operations, communication with the peripherals etc. Naturally, only one process is used since the initial specification of the RP consists of a *homogeneous* instruction set, without any notion of partitioning. The *cpu_1* process consists of three procedures. The *fetch_and_execute* procedure fetches an instruction from memory, increments the program counter, decodes the opcode of the instruction, and executes the instruction. Each instruction is fetched and executed at the rising edge of a clock signal. Instruction execution will result in a change in either the memory, the register file or both. Note that there is no concept of distinct functional units such as adders, multipliers, shifters etc. Since the timer and clock divider function in parallel with the execution of instructions they are modelled as separate procedures. Thus, the main code of the *cpu_1* process consists of calls to the three procedures: *fetch_and_execute, timer and clock_divider.* Though the procedures execute in sequence once every *clock* signal, they do so in zero time. They are thus modeled as concurrent behaviors.

Figure 6 gives a conceptual view of the SLM. In this model it is assumed that the RP will communicate with the peripheral devices using software subroutines (the instructions for

```
architecture specification_level of RP is

    - Global signal declarations
    signal : clock;

    - Clock generator process
    clock_generator : process
    begin
        Generate a square pulse with period 200 ns and 50% duty cycle on clock signal
    end process

    - Main/Core process
    cpu_1 : process (clock)

        - Global variable declarations
        variable memory : memory_type;
        variable register_file_1 : register_type;

        - Fetch and execute RP instructions
        procedure fetch_and_execute is
        begin
            Fetch current instruction from memory and place in instruction register
            Increment the value of the program counter
            case opcode is
                when ADD_SUB_IMMED
                    Add and store result in given register
                when MULT_REG_WORD
                    Multiply and store result in given register
                when BRANCH_IMMED
                    If condition is met, branch to given instruction
                .
                .
                .
            end case;
        end fetch_and_execute;

        procedure timer is
        begin
            If timer control is not reset then decrement the timer register.
            If timer register = 0 then generate the interrupt signal and load the timer register with new value
        end timer
        procedure clock_divider is
        begin
            Divide input clock frequency by value given in clock divider register
        end clock_divider

    - Main/Core process begins here
    begin
        if clock = 1 then
            fetch_and_execute;
            timer;
            clock_divider;
        endif;
    end process;
end specification_level;
```

Figure 10: Pseudo code for the SLM model

```
entity RP is
    port(
        -Host Ports
        DATA_BUS    : inout 8_bit_bustype;
        READ        : in port_type;
        WRITE       : in port_type;
        BUSY        : in port_type;
        BUF_ST      : in port_type;
        IRQP        : in port_type;
        FULL        : in port_type;
        EMPTY       : in port_type;
        DONE        : in port_type;

        -Serial Receiver Transmitter Ports
        .
        .
        .

        -External Memory Ports
        .
        .
        .

    )
end RP;
```

Figure 11: Ports on the *RP* entity

communication typically set and reset an I/O signal, wait for an I/O signal to be asserted or deasserted, place data or address on a bus etc). This places certain limitations on the design: (1) in case of slow peripheral devices, the RP has to wait for the data communication (as opposed to performing data computation tasks in parallel), and (2) the data transfer rate can be no more than the execution time of a "communication" subroutine (which may consist of several instructions).

The RP entity depicted in Figure 10 consists of the I/O ports required for communication with the external memory, host, and serial receiver/transmitter (Figure 11). Communication protocols are modeled by associating appropriate delay values with signal assignments on these ports. (This is done within the "communication" subroutines). As long as timing on these ports is modeled accurately, the SLM can serve as an input specification for system designers.

The SLM was simulated using three Livermore Loop benchmarks (Figures 23, 24, and 25). The simulations were used to verify the functional correctness of the model as well as gather some preliminary information regarding the performance of the RP. Since software subroutines were used for communication the data transfer rate (between the host and *cpu_1* process) was notably slow.

4.1 Refined Specification Level Model

In order to increase the data transfer rate we decided to use hardware interfaces for the communication. The RP controls the interfaces by issuing instructions to set, reset or monitor interface signals. Thus the instruction set must be modified. Some instructions used to control or monitor the communication signals between the RP and the peripherals are removed, and instructions used to control the interfaces are added. The communication protocols are also changed.

The refined SLM consists of two additional hardware modules: a host interface, and a serial receiver/transmitter (SRT) interface. Figure 12 gives the pseudo code for the refined SLM and Figure 7 gives a conceptual view of the model. The two interfaces are *completely* extracted from the core (modeled as the *cpu_1* process in Figure 10), and a clearly defined protocol exists between the interfaces and the RP. Thus the interfaces are modeled as two separate processes, which execute in parallel with the RP *cpu_2* process.

The global signal declarations now include all the control signals and data and address buses between the interfaces and the RP core. The RP core in this model (referred to as *cpu_2*) is only slightly different from the *cpu_1* process in the previous model. The difference is in instructions controlling the interfaces. The global variable declarations within the *cpu_2* process consist of the memory and the reduced register file, *register_file_2*. Whereas *register_file_1* consisted of the host and SRT buffers and status registers, these registers are not included in *register_file_2*. The registers used in host communication are declared as variables within the *host_interface* process, and similarly, the registers required for communication with the SRT device are included in the *SRT_interface* process. This localization of the host and SRT registers reduces the communication requirement between the interface and core processes and hence, increases the performance of the RP. For instance, in the host interface, had the tail pointer (required to keep track of an empty location in the circular buffer) been declared in the *cpu_2* process (as opposed to the *host_interface* process), every time data was written to the circular buffer, extra clock cycles would be required to update the tail pointer.

We now elaborate on the host interface to show the difference between the initial SLM model and the refined model. Figure 13 gives the pseudo VHDL code of the host interface. The host can send or receive data from the RP one byte at a time. The RP initiates data transfer by writing a control word to the status register in the host interface. This indicates the mode of transfer (read/write), the number of bytes to be transferred and the address of the memory in the host from where the transfer should begin. If the host device is not processing a previous data transfer request, the interface sends an interrupt signal to the host device. The host device then reads the status information and informs the interface

```
architecture refined_specification_level of RP is

    - Global signal declarations
    Includes signals for communication between the
        host_interface process and the cpu_2 process
        SRT_interface process and the cpu_2 process
        clock_generator process and the cpu_2 process

    clock_generator : process
    begin
        Identical to clock_generator process in previous SLM model
    end process
    host_interface : process
        Variable declarations of circular buffer, status register, head and tail pointers and the 2-bit counter
    begin
        Allows the RP core and the host device to write/read from the circular buffer
        Signals between the RP core process and the interface, and between
        the host device and the interface are used to control the data transfer
    end process
    SRT_interface : process
        Variable declarations of receive and transmit buffers, counters, enable bits, status flags etc.
    begin
        Allows the RP core to transmit or receive data from the SRT device one bit at a time
        Signals between the RP core process and the SRT interface, and between
        the SRT device and the SRT interface are used to control the data transfer
    end process

    - Main/Core process
    cpu_2 : process (clock)

        - Global variable declarations
        variable memory : memory_type;
        variable register_file_2 : register_type;

        procedure fetch_and_execute is
        begin
            Similar to fetch_and_execute procedure in previous SLM model
            (except for a change in the RP instruction set being modeled)
        end fetch_and_execute;
        procedure timer is
        begin
            Identical to timer procedure in previous SLM model
        end timer
        procedure clock_divider is
        begin
            Identical to clock_divider procedure in previous SLM model
        end clock_divider
    - Main/Core process begins here
    begin
        if clock = 1 then
            fetch_and_execute;
            timer;
            clock_divider;
        endif;
    end process;
end refined_specification_level;
```

Figure 12: Pseudo code for the refined SLM model

```
host_interface : process
    - Register Declarations
    variable circular_buffer : buffer_type;
        - 8 × 32 circular buffer
    variable circular_status : register_type;
        - for starting address, number of bytes etc
    variable head_pointer : register_type;
        - 3 bit register
    variable tail_pointer : register_type;
        - 3 bit register
    variable counter : register_type;
        - 2 bit register, for keeping track of next byte to transfer

    procedure Update_FULL_signal
    begin
        Compare head_pointer and tail_pointer to determine if buffer is full
        Accordingly set/reset the FULL signal
    end

    procedure Update_EMPTY_signal
    begin
        Compare head_pointer and tail_pointer to determine if buffer is empty
        Accordingly set/reset the EMPTY signal
    end

begin

    wait for rising edge of clock signal;
    if RP writes to circular_status then
        if BUSY = 0 then      - making sure that host is not busy processing the last request
            INTR := 1;        - interrupt the host device
            counter := 0;     - initialize the counter
        else
            Inform the RP that the host is busy
        end if;
    end if;
    if READ signal is asserted then
        if EMPTY = 0 then     - indicates that the circular_buffer is not empty
            Send byte[counter] of word[head_pointer] in circular_buffer to DATA_BUS
            When data on DATA_BUS is ready for reading inform the host by asserting the DONE signal
            counter := (counter + 1) mod 4
            Update_EMPTY_signal
        else
            Wait for RP to write a word to the circular buffer
    end if;
    if WRITE signal is asserted then
        if FULL = 0 then      - indicates that the circular_buffer is not full
            Put contents of DATA_BUS in byte[counter] of word[head_pointer] of circular_buffer
            After reading DATA_BUS inform the host by asserting the DONE signal
            counter := (counter + 1) mod 4
            Update_FULL_signal
        else
            Wait for RP to read a word from the circular buffer
    end if;

end process
```

Figure 13: Pseudo code of *host_interface* process

when it is ready to transfer data. The interface has to keep track of which byte to transfer. The *counter, head_pointer* and *tail_pointer* are used for this purpose. After transferring a word to (from) the host device, the interface updates the *tail_pointer (head_pointer)* and the *EMPTY (FULL)* line, if necessary.

For example, if the RP requires 50 pieces of data from the host device, then, in the initial SLM, the user would have to program the RP to interrupt the host device, send the status word to the host device, and for each *byte* of data transfer (there are 50×4 such bytes) issue appropriate control signals (this is approximately, a sequence of 5 RP instructions for each byte transfer). Since the data transfer is handled by the RP core single handedly, it cannot perform any other task while the communication is in progress.

In the refined SLM, the user has to program the RP to send the status word to the host interface and, for each *word* transfer, issue appropriate control signals to read from the circular buffer within the *host_interface* process. This sequence also consists of approximately 5 RP instructions, but it is issued once every word (32 bit width) rather than for every byte. This reduces the length of the communication code by a factor of four.

The refined SLM model was tested using the same benchmarks. It was considerably faster than the previous model (Table 2). Furthermore, we used the SLM to obtain other performance paramenters. For instance, we measured the effectiveness of the vector access registers and the automatic loopback register. We found that both the VARs and the automatic loopback register increase performance dramatically on our benchmarks (Table 3 and 4). This is discussed in more detail in Section 6.

5 Functional Level Model

The basic idea in *functional level modeling* is to decompose the design into several functional blocks which communicate via signals/buses. As mentioned in Section 3, a functional level model is useful for verifying timing, locating critical paths, and selecting the number of pipeline stages (if any) needed in the design. It should also be noted that grouping or partitioning of the design at the functional level determines the target architecture. In other words, the functional blocks of the design and the interconnection between them must be completely specified.

We have written two functional level models for the RISC Processor, non-pipelined (NPM) and pipelined (PM), which are described in the following subsections.

5.1 The Non-pipelined Model

We shall divide our discussion of NPM into three main sections. First, we describe how we partitioned the RP design into functional level components and how those components are

modeled in VHDL. Secondly, we describe each RP functional block in some detail, as well as the interconnection (busing structure) between the functional blocks. We then discuss the the general structure of the NPM code, as well as testing of NPM and the need for a pipelined model.

5.1.1 Partitioning of the RP Design

In the non-pipelined version of the RP functional level model, the RP core is decomposed into seven functional blocks: controller, register file, ALU, program memory (PMEM), data memory (DMEM), timer, and clock divider. Figure 8 shows the block diagram for NPM. Decomposition is done in two main phases: storage grouping and behavioral partitioning. In the *storage grouping* phase, variables from the functional level model are arranged into groups that will be implemented as a single storage unit such as a memory or register file. These groupings reduce the interconnection cost in the design since each element of a register file or memory is accessed using a protocol and is not connected directly to any functional unit. It should be noted, however, that there is a cost/performance tradeoff involved in storage grouping since the access of memory and register files will be slower than access to single registers. In the *behavioral partitioning* phase, design behaviors which may execute in parallel with one another are partitioned into functional blocks. Decomposition of the RP design is performed as follows.

RP Storage Grouping

Storage grouping divides the RP storage space into three functional blocks. Temporary storage elements, used to hold data entering and leaving functional units, are grouped into a *register file.* Register file elements are characterized by low frequency, random access; hence, they can be grouped together. Frequently accessed registers such as PC, IR, or the loop register, for example, cannot be grouped into the register file since they cause too many conflicts at ports. Introduction of the register file into the design helps to reduce interconnection cost in the RP; however, since many register file elements must be accessed concurrently, the register file needs to have multiple ports (specifically, 7 ports) in order to satisfy timing requirements. For this reason, interconnection cost is only slightly improved.

RP *memories (program and data)* are separated from the register file because they are used for, comparatively, long term storage and, in the case of data memory, provide some shared memory locations between the interfaces and the RP. The shared memory cannot be implemented in the register file because it is too large. (A large register file would necessitate an abnormally long instruction word length.) Data and program memory have been separated from one another in order to reduce the size of the PC and minimize

instruction word length.

RP Behavioral Partitioning

RP behavior is partitioned into four functional blocks: controller, ALU, timer, and clock divider. The behavior of the *controller* is to decode the current instruction and issue signals to control the register file, ALU, data memory, program memory, timer, clock divider, and also the host/SRT interfaces. The control portion of the RP has been separated from the ALU behavior because the ALU behavior operates on data from the register file, whereas the controller requires information from the PC and IR, which cannot be grouped into the register file. The ALU behavior consists of addition, shifting, logical and bitwise operations, and multiplication. These behaviors have been grouped together because they are mutually exclusive and all require data from (and store data to) storage elements in the register file.

The *timer* unit is separated from the controller, ALU, and clock divider because its behavior is not mutually exclusive with that of the other functional blocks (controller, ALU, clock divider). The timer implements the following behavior. It is loaded with an initial value using RP instructions, and then, it counts down on every rising-edge of the RP clock until it reaches zero, at which time it sends an interrupt to the RP. Since the timer behavior may execute at (potentially) every RP clock, it is prudent to consider the timer as a separate functional block. The *clock divider* is separated from the controller, ALU, and timer units because its behavior too is not mutually exclusive with that of the other RP functional blocks. The behavior of the clock divider unit is to produce an output frequency which is $\frac{N}{2M}$ times the RP clock frequency where N and M are values that can be assigned using RP instructions.

5.1.2 RP Functional Blocks and Interconnection

In this section, we discuss the behaviors of the RP functional blocks and how to model them using VHDL. The interconnection between functional blocks (RP busing structure) is also described. In general, we follow the basic modeling conventions described in Section 3.3. For example, functional blocks are modeled as *processes* which communicate with one another using *global signals*. Temporary storage elements such as registers or buffers are modeled as *variables*, and memories/register files are modeled as *array variables*. VHDL descriptions of the seven RP functional blocks (controller, program memory, register file, ALU, data memory, timer, and clock divider) are as follows.

Controller

The RP controller performs two main functions: decoding instructions and controlling

program flow. As shown in Figure 14, instruction decoding is modeled using *IF statements* to group those instructions (according to their opcodes) which require a common value for some control signal. For example, if the current instruction in IR is OP_ADD_SUB_IMMED or OP_ADD_SUB_REG and the option field *op*(1) has value zero, then the ALU control signal *ALU_instr* receives the value ADD, indicating that the ALU must perform an ADD operation. Similarly, if the current instruction in IR has opcode OP_LOAD_IMMED, OP_MOVE_REG, OP_INIT_SI, OP_INIT_SC, or OP_START_LOOP_INDIRECT, then the signal *ALU_instr* gets the value PASS. Generation of control signals for the remaining functional blocks is done in a similar fashion to control generation for the ALU; although, for the sake of brevity, only the ALU case is shown in Figure 14. It should be noted that, in the VHDL model of the controller, there must exist one IF statement for each possible value of each control output signal.

The other main function of the controller is to manage program flow. During each control cycle, an instruction is fetched from the program memory according to the value stored in the PC. The value of the PC is then incremented or (possibly) modified if the current instruction is a BRANCH, JUMP, CALL, RETURN, or WAIT. The PC may also be modified by the controller when automatic loopback is required. In this case, the PC receives the "start of loop" value which is stored in the loop register.

Also, NPM uses a "lumped delay" model, so all circuit delay is incurred within the processes representing the functional blocks. For example, the controller delay is modeled using the "WAIT FOR controller_delay NS" statement shown in Figure 14.

Program Memory

Obviously, the program memory provides storage locations for program instructions. In NPM, all program memory is modeled as program ROM. This is done for simplicity, to eliminate the need for an operating system in the model. The VHDL description of the program memory is shown in Figure 15. Note that the PMEM itself is modeled as an array variable *pmem*. If the *pmem_enable* signal is HIGH, then the program memory delay is incurred, and if the *pmem_read* signal is also HIGH, a PMEM read is performed. During a read operation the data stored at location *pmem_addr* is written to the program bus *pmem_data*. It should also be noted that program memory needs only one port since it is accessed only by the controller process. The size of PMEM is 16K $\times$32 bits.

Register File

The register file performs four basic types of operations: fetching operands for ALU operations, performing comparisons for BRANCH operations, fetching data for memory

```
controller : process
      variable PC : register_type;      - program counter
      variable IR : register_type;      - instruction register
      variable loop_register : register_type;   - loop register
      variable status_register : register_type;  - status register
begin
      wait on rsp_clk_p=1 and not rsp_clk_p'stable;
      - fetch instruction from program memory and store it in IR;
      wait for controller_delay ns;
      - generate control signals for ALU;
            if (ir.opcode = OP_LOAD_IMMED or ir.opcode = OP_MOVE_REG or ir.opcode = OP_INIT_SI
            or ir.opcode = OP_INIT_SC or ir.opcode = OP_START_LOOP_INDIRECT) then
                  ALU_instr <= PASS;
            end if;
            if (ir.opcode = OP_ADD_SUB_IMMED or ir.opcode = OP_ADD_SUB_REG) then
                  if (ir.op(1) = 0) then ALU_instr <= ADD; end if;
                  if (ir.op(1) = 1) then ALU_instr <= SUB; end if;
            end if;
            if (ir.opcode = OP_MULT_IMMED_WORD or ir.opcode = OP_MULT_REG_WORD or
            ir.opcode = OP_MULT_ACC_WORD) then
                  ALU_instr <= MUL;
            end if;
            if (ir.opcode = OP_COMPARE_IMMED_LT) then
                  ALU_instr <= COMPARE;
            end if;
            if (ir.opcode = OP_BITWISE_REG_OR) then
                  ALU_instr <= MY_OR;
            end if;
            if (ir.opcode = OP_BIT_SET_IMMED) then
                  if (ir.op(1) = 0) then ALU_instr <= SET1; end if;
                  if (ir.op(1) = 1) then ALU_instr <= SET0; end if;
                  if (ir.op(1) = 2) then ALU_instr <= TOGGLE; end if;
            end if;
            if (ir.opcode = OP_BIT_TEST_IMMED) then
                  if (ir.op(1) = 0) then ALU_instr <= TEST0; end if;
                  if (ir.op(1) = 1) then ALU_instr <= TEST1; end if;
            end if;
      - generate control signals for register file;
      - generate control signals for data memory;
      - generate control signals for timer;
      - generate control signals for clock divider;
      - generate control signals for host interface;
      - generate control signals for SRT port;
      - if automatic loopback is needed, then update PC;
      - if branch/jump/call/return is taken, then update PC;
      - if an interrupt has occured, then update PC;
end process controller;
```

Figure 14: Pseudo code for controller.

```
program_memory : process
    variable pmem : memory_type;     - program memory
begin
    wait until pmem_enable=1 and not pmem_enable'stable;
    wait for pmem_delay ns;
    if (pmem_read=1) then
        pmem_data ⇐ pmem(pmem_addr);
    end if;
end process program_memory;
```

Figure 15: Pseudo code for program memory.

LOAD/STORE operations, and storing the results of ALU computations. For an example of ALU operand fetching, the REG_IMM operation shown in Figure 16 fetches data from the location on the *src*1 address bus, sends this data to the ALU via the *alu_data*1 bus, and sends data from the *immediate_data*1 bus to the ALU via the *alu_data*2 bus. A REG_IMM operation might be performed during an RP instruction such as OP_ADD_SUB_IMM. In the case of a BRANCH operation, data is fetched from the location given by *src*1 and compared to the value on *immediate_data*1. If the comparison indicates that a branch should occur the *branch_signal* is set; otherwise, it is reset. In the case of a DMEM_INST operation, appropriate data is fetched from the register space and loaded onto *addr_bus*1, *addr_bus*2, *data_bus*1, and/or *data_bus*2. The exact process of selecting which data to load onto each bus is slightly more complicated (requires additional control signals) and has been omitted for the sake of brevity.

Finally, if the control signal *RF_dest* has value DESTINATION then the result of the current ALU computation (taken from the *alu_result* bus) must be stored to the register file location on the address bus *dest*. Depending on the value of *ACC*, the value on *dest* may be the address of an AC register, in which case the result of the ALU computation is accumulated with the value currently stored in AC register *dest*. It should be noted that due to the large number of concurrent data accesses required in the RP register file, seven ports are needed on register file memory. The corresponding register file data buses are *alu_data*1, *alu_data*2, *alu_result*, *addr_bus*1, *data_bus*1, *addr_bus*2, and *data_bus*2. During refinement of NPSM to the register-transfer level, multiphase clocking could be introduced to enable time-multiplexing and merging of buses; however, at the functional level, only a single-clock design is considered.

ALU

The function of the ALU process is to perform arithmetic, logical, compare, bit set, or bit test operations on data inputs received from the register file. Data enters the ALU

```
register_file : process
      variable TDS : register_type;      - temporary data storage
      variable ZERO : register_type;     - hardware zero
      variable AC : register_type;       - accumulators
      variable VI : register_type;       - VAR index registers
      variable VC : register_type;       - VAR context registers
      variable VD : register_type;       - VAR data registers
      procedure UasStore (addr:in integer; data:in integer) is
      begin
            - store data to the address addr
      end;
      function UasFetch (addr:in integer) return integer is
      begin
            - return data fetched from the address addr
      end;
begin
      wait on RF_instr'transaction;
      case RF_instr is
            when REG_IMM ⇒
                  alu_data1 ⇐ UasFetch(src1);
                  alu_data2 ⇐ immediate_data1;
            ⋮
            when BRANCH ⇒
                  if (UasFetch(src1)=immediate_data1) then
                        branch_signal ⇐ '1' - take the branch
                  else
                        branch_signal ⇐ '0' - do not branch
                  end if;
            when DMEM_INST ⇒
                  - place appropriate data onto data_bus 1 and/or addr_bus 1
      end case;
      wait for register_file_delay ns;
      if (RF_dest=DESTINATION) then
            UasStore(alu_result,dest);
      end if;
      if (RF_dest=DESTINATION and ACC='1') then
            UasStore(UasFetch(dest)+alu_result,dest);
      end if;
end process register_file;
```

Figure 16: Pseudo code of register file

```
alu : process
begin
      wait on alu_data1'transaction, alu_data2'transaction;
      wait for ALU_delay ns;
      case ALU_instr is
            when PASS ⇒
                  alu_result ⇐ alu_data1;
            when ARITHMETIC_OPERATION ⇒
                  alu_result ⇐ Rel(alu_data1, alu_data2);
            when COMPARE_OPERATION ⇒
                  status_register ⇐ alu_data1 Rel alu_data2;
            when LOGICAL_OPERATION ⇒
                  alu_result ⇐ Rel(alu_data1, alu_data2);
            when SET 1 ⇒
                  - set bit binary_to_integer(alu_data2) of word alu_data1
            when SET 0 ⇒
                  - reset bit binary_to_integer(alu_data2) of word alu_data1
            when TEST 1 ⇒
                  status_register Leftarrow Rel(alu_data1, alu_data2,'1');
            when TEST 0 ⇒
                  status_register Leftarrow Rel(alu_data1, alu_data2,'0');
      end case;
end process alu;
```

Figure 17: Pseudo code for ALU.

via the data buses *alu_data*1 and *alu_data*2, while data is output using the *alu_result* bus. As shown in Figure 17, the current ALU operation is selected, using a CASE statement, according to the value of the *ALU_instr* signal. The RP ALU performs eight basic type of operations: PASS, ARITHMETIC, COMPARE, LOGICAL, BIT SET, BIT RESET, BIT TEST 1, and BIT TEST 0. The PASS operation simply outputs the data the is received on *alu_data*1, without modification. ARITHMETIC operations (+,-,*,...) perform calculations such as addition, subtraction, or multiplication on the values from *alu_data*1 and *alu_data*2 and write the result to the *alu_result* signal. Similarly, LOGICAL operations (and,or,invert,...) compute logical functions on the ALU inputs and return an output value on *alu_result*. COMPARE operations perform relational functions ($>, <, =, \ldots$) on the *alu_data*1 and *alu_data*2 values and update the status bits depending on the result. The BIT SET (BIT RESET) operations set (reset) the i^{th} bit of the value stored on *alu_data*1 where $i = binary\_to\_integer(alu\_data2)$, and the BIT TEST 1 (BIT TEST 0) instructions test whether the i^{th} bit of *alu_data*1 is '1' ('0'). Note that all BIT SET and BIT TEST operations update the status register. The ALU delay value is modeled using the statement "WAIT FOR ALU_delay NS."

```
data_memory : process
    variable dmem : memory_type;     - data memory
begin
    wait until rsp_clk_p=1 and not rsp_clk_p'stable;
    data_bus 1 ⇐ null;
    data_bus 2 ⇐ null;
    wait until dmem_enable=1 and not dmem_enable'stable;
    wait for dmem_delay ns;
    if (port_select=1 or port_select=3) then
        if (dmem_read 1=1) then
            data_bus 1 ⇐ dmem(addr_bus 1);
            data_rdy 1 ⇐ 1;
        end if;
        if (dmem_write 1=1) then
            dmem(addr_bus 1) ⇐ data_bus 1;
        end if;
    end if;
    if (port_select=2 or port_select=3) then
        if (dmem_read 2=1) then
            data_bus 2 ⇐ dmem(addr_bus 2);
            data_rdy 2 ⇐ 1;
        end if;
        if (dmem_write 2=1) then
            dmem(addr_bus 2) ⇐ data_bus 2;
        end if;
    end if;
end process data_memory;
```

Figure 18: Pseudo code for data memory.

Data Memory

The RP data memory is a dual port, 4096-word × 32-bit RAM with ports connected to the buses *addr_bus*1, *data_bus*1, *addr_bus*2, and *data_bus*2. It is modeled using an array variable *dmem* as shown in Figure 18. If the *dmem_enable* signal is HIGH, a memory read or write operation is performed depending on the value of the *port_select*, and read/write signals. If *port_select* = 1, then the values on *addr_bus*1 and *data_bus*1 are used for the operation. If *port_select* = 2, then the values on *addr_bus*2 and *data_bus*2 are used. If *port_select* = 3, then memory operations occur on both ports. The signals *dmem_read*1, *dmem_write*1, *dmem_read*2, and *dmem_write*2 are used to select the type of operation (read or write) to occur on each port. Data memory delay is incurred using the WAIT FOR *dmem_delay* statement. Also, it should be noted that the data output during a DMEM read operation is valid on the data bus for one clock period.

Timer

The behavior modeled by the timer process is to count down from the value stored in the timer counter register T1CTR until T1CTR = 0, at which time an interrupt is sent to

```
timer : process
      variable T1LTH : register_type;      - timer load register
      variable T1CTR : register_type;      - timer counter register
      variable Csel : register_type;       - clock select
begin
      wait on rsp_clk_p;
      if (read_LTH=1 and rsp_clk_p=1) then
            data_bus 1 ⇐ T1LTH;
      end if;
      if (read_CTR=1 and rsp_clk_p=1) then
            data_bus 1 ⇐ T1CTR;
      end if;
      if (write_LTH=1 and rsp_clk_p=1) then
            T1LTH ⇐ data_bus 1;
      end if;
      if (write_CTR=1 and rsp_clk_p=1) then
            T1CTR ⇐ data_bus 1
      end if;
      if ((Csel=1 and rsp_clk_p=1) or (Csel=0 and rsp_clk_p=1 and ext_clk_p=1)) then
            - decrement T1CTR;
            if (T1CTR=0) then
                  - generate RP interrupt;
                  T1CTR := T1LTH;
            end if;
      end if;
      - generate output frequency;
end process timer;
```

Figure 19: Pseudo code for timer.

```
clock_divider : process
      variable CDIVAM : register_type;
      variable CDIVAN : register_type;
      variable phase : register_type;      - phase generator
begin
      wait on rsp_clk_p;
      if (read_m=1 and rsp_clk_p=1) then
            data_bus 1 ⇐ CDIVAM;
      end if;
      if (read_n=1 and rsp_clk_p=1) then
            data_bus 1 ⇐ CDIVAN;
      end if;
      if (write_m=1 and rsp_clk_p=1) then
            CDIVAM := data_bus 1;
      end if;
      if (write_n=1 and rsp_clk_p=1) then
            CDIVAN := data_bus 1;
      end if;
      if (rsp_clk_p=1) then
            phase := not phase;
      end if;
      if (phase=1 and rsp_clk_p=1) then
            - generate output pulse on CDIVAO;
      end if;
      if (rsp_clk_p=0) then
            CDIVAO Leftarrow 0;
      end if;
end process clock_divider;
```

Figure 20: Pseudo code for clock divider.

the RP controller. Also, when the interrupt is generated, a new value is loaded into T1CTR from the timer load register T1LTH. As shown in Figure 19, T1CTR and T1LTH can be read from or written to depending on the value of the control signals $read\_LTH$, $write\_LTH$, $read\_CTR$, and $write\_CTR$. Also, the timer may perform its decrement operation on T1CTR according to either the RP clock rsp_clk_p or an external clock ext_clk_p which is synchronized the the RP clock.

Clock Divider

The function of the clock divider process is to take the RP clock rsp_clk_p as input and output the frequency $rsp\_clk\_p\frac{N}{2M}$, where N and M are the values stored in the registers CDIVAN and CDIVAN, respectively. Note that CDIVAN and CDIVAM can be read or written by a RP program depending on the values of the control signals $read\_m$, $write\_m$, $read\_n$, and $write\_n$. The output frequency is produced by implementing a phase generator as shown in Figure 20. The CDIVAO output is set HIGH when $phase = 1$ and $rsp\_clk\_p = 1$ and *not* $rsp\_clk\_p'stable$. It is reset when $rsp\_clk\_p = 0$ and *not* $rsp\_clk\_p'stable$.

Interconnection Structure

Note that, in the NPM model, all system buses and their sizes are completely specified. The general RP interconnection structure is as follows. The RISC processor has two primary, 32-bit data (address) buses, *data_bus*1 and *data_bus*2 (*addr_bus*1 and *addr_bus*2). *Data_bus*1 connects the controller, register file, data memory, timer, clock divider, host interface, and SRT interface, whereas *data_bus*2 connects the register file, data memory, and SRT interface. Three other 32-bit data buses, *alu_data*1, *alu_data*2, and *alu_result*, are provided to pass data between the register file and the ALU. As mentioned earlier, the register file has seven ports and is connected the register file data buses *data_bus*1, *data_bus*2, *addr_bus*1, *addr_bus*2, *alu_data*1, *alu_data*2, and *alu_result*. The address buses for the register file are *src*1, *src*2, *src*3, *dest*, *imm*1, and *imm*2, and the appropriate register file address for each data bus is selected depending on the register file control. The last two system buses connect the program memory and the controller. The *pmem_data* bus is a 32-bit data bus (for the instruction word), and *pmem_addr* is a 22-bit address bus for the PC value. The complete RP busing structure is shown in Figure 8. It should be noted that in NPM, delay values for buses/signals are not modeled.

Although the interconnection structure in a functional level model implies a basic architecture for the design, it may not correspond exactly to the final layout. Bus merging and time-multiplexing of buses is often implemented at a later stage in the design process. For example, in the RP design, buses should be merged so that the register file does not require seven ports in the final layout.

5.1.3 General Structure and Testing of NPM

In this section, we shall discuss the overall structure of NPM as well as testing of the model. Figure 21 depicts a portion of the VHDL code for NPM. In each clock cycle, the controller issues the appropriate instructions and data values to the PMEM, register file, ALU, and DMEM according to the value stored in the instruction register. The register file process is activated by the arrival of data on one of its address buses. It then executes a read operation and puts data onto the appropriate register file data bus (*data_bus*1, *addr_bus*1, *data_bus*2, *addr_bus*2, *alu_data*1, and/or *alu_data*2). Since the register file process also models branching hardware, if the current register file instruction is a BRANCH, a signal ('1' or '0') is sent to the controller depending on the result of the "branching" comparison. The register file process then waits for any data that must be stored either from data memory (*data_bus*1, *data_bus*2) or from the ALU (*alu_result*) then terminates. The ALU process is triggered by any transaction on the signals *alu_data*1 and/or *alu_data*2, writes its result to the signal *alu_result*, and then terminates. Both data and program memory processes

are activated by transactions on their address buses (*pmem_addr* in the case of PMEM and *addr_bus*1 or *addr_bus*2 in the case of DMEM) and write results the the corresponding data buses (*pmem_data*, *data_bus*1 or *data_bus*2).

For an example, we consider the execution of the instruction: ADD_SUB_REG R1, R2, R3. This instruction adds (or subtracts, depending on instruction options) the values stored in R1 and R2 and writes the result to register R3. (We consider the case of addition.) First, we must fetch the ADD_SUB_REG instruction from the program memory according to the value stored in the PC. For the ADD_SUB_REG instruction, the controller would load the values R1 and R2 onto the address buses *src*1 and *src*2 and the value R3 onto the *dest* bus. The register file control signals *RF_instr* and *RF_dest* are set to *REG*2 and *DESTINATION*, respectively, and the ALU control signal *alu_instr* gets the value *ADD*. Data is fetched from the R1 and R2 registers and placed onto the ALU data buses *alu_data*1 and *alu_data*2. Once the *alu_result* signal is updated with the result of the computation, the *alu_result* value is stored to register file location R3. This completes the execution of the ADD_SUB_REG instruction. Note that during this computation, the *dmem_enable* signal remains LOW since no memory operation is required.

Recall that one of the main purposes of functional level modeling is to obtain timing data through model simulation and adjust the model/design accordingly. We performed such timing simulations on the NPM model for RP. Simulation results indicating the real-time (time taken in hr:min:sec for the NPM simulation to complete) performance of NPM and the simulated performance (time in terms of ns for an RP program to execute on the NPM model) of NPM on Livermore Loop benchmarks are given in Section 6. Due to the delay values for the RP functional blocks, a 200 ns clock period is required for NPM. As this clock period is "too long" and does not match with RP specifications, we decided to pipeline the RP design, since the data memory, program memory, ALU, and register file functional units all have low utilization. This pipelined design is described in detail in Section 5.2.

5.2 The Pipelined Model

In order to reduce the clock period of the RISC Processor and improve component utilization, we introduce a four-stage pipeline into the design. In Stage 1, we fetch the next instruction from the program memory. Stage 2 is used to fetch ALU operands from the register file as well as to implement branches. In Stage 3, ALU operations are executed, and in Stage 4, memory loads/stores are performed. It should be noted that the timer and clock divider circuits, once initialized, execute in parallel with remaining RP processes. In other words, these basic blocks are not considered as part of the pipeline.

Modeling pipelined designs in VHDL is difficult since VHDL does not have built-in

```
entity E is port( ... ); end E;

architecture A of E is

    - (Global Signal Declarations)

    controller : process
        - (Variable Declarations)
    begin
        - Initiate Control Cycle
            wait until (rsp_clk_p = '1') and not (rsp_clk_p'stable);
        - Update Instruction Registers
            IR := pmem_data;
        ⋮
    end process controller;

    register_file : process
        - (Variable Declarations)
    begin
        - Initiate Register File Cycle
            wait on RF_instruction'transaction;
        ⋮
    end process register_file;

    alu : process
        - (Variable Declarations)
    begin
        - Initiate ALU Cycle
            wait on alu_data1'transaction, alu_data2'transaction;
        ⋮
    end process alu;

    data_memory : process
        - (Variable Declarations)
    begin
        - Initiate DMEM Cycle
            wait until (dmem_enable = '1') and not (dmem_enable'stable);
        ⋮
    end process data_memory;

    program_memory : process
        - (Variable Declarations)
    begin
        - Initiate PMEM Cycle
            wait until (pmem_enable = '1') and not (pmem_enable'stable);
        ⋮
    end process program_memory;

end A;
```

Figure 21: NPM VHDL code excerpt.

```
entity E is port( ... ); end E;

architecture A of E is

    - (Global Signal Declarations)

    controller : process
        - (Variable Declarations)
    begin
        - Initiate Control Cycle
            wait until (rsp_clk_p = '1') and not (rsp_clk_p'stable);
        - Instruction Registers
            ir1 := pmem_data; ir4 := ir3; ir3 := ir2; ir2 := ir1;
        ⋮
    end process controller;

    register_file : process
        - (Variable Declarations)
    begin
        - Initiate Register File Cycle
            wait until (rsp_clk_p = '1') and not (rsp_clk_p'stable);
        ⋮
    end process register_file;

    alu : process
        - (Variable Declarations)
    begin
        - Initiate ALU Cycle
            wait until (rsp_clk_p = '1') and not (rsp_clk_p'stable);
        ⋮
    end process alu;

    data_memory : process
        - (Variable Declarations)
    begin
        - Initiate DMEM Cycle
            wait until (rsp_clk_p = '1') and not (rsp_clk_p'stable);
        ⋮
    end process data_memory;

    program_memory : process
        - (Variable Declarations)
    begin
        - Initiate PMEM Cycle
            wait until (rsp_clk_p = '1') and not (rsp_clk_p'stable);
        ⋮
    end process program_memory;

end A;
```

Figure 22: PM VHDL code excerpt.

constructs for pipelining. Our solution to this problem is to create one process to represent each pipeline stage. Naturally, our four RP pipeline stages correspond directly to the program memory, register file, ALU, and data memory processes from NPM, as these functional blocks have similar delay values. If this is not the case, it may be necessary to restructure the NPM model such that the delay values for functional blocks are similar or can be grouped into pipelined stages such that the maximum delay value through the blocks in each stage is approximately the same as for the remaining stages. In the case of PM, since each pipeline stage corresponds to one process, the four pipeline processes (program memory, register file, ALU, and data memory) are activated on the rising edge of the RP clock as shown in Figure 22, rather than on bus transactions as in NPM. Effectively, these processes execute in parallel in the pipelined model rather than serially as in the non-pipelined case. It should be noted that this method of activating pipeline processes is designed for the case where each stage consists of only one functional block. For pipelines with several functional blocks per stage, a VHDL block containing several communicating processes is used to model a stage, and the clock triggers one or more (but not necessarily all) of the processes in each stage.

Due to the style of pipeline modeling used in PM, the PM VHDL code is very similar to its non-pipelined counterpart. Of course, the primary difference is that program memory, register file, ALU and data memory processes (each corresponding to a pipeline stage) are triggered by the rising edge of the RP clock rather than by bus transactions. The other important deviation from the non-pipelined VHDL model is the introduction of an additional three instruction registers. (There are four in total.) One instruction register is needed for each pipeline stage in order to generate control for the corresponding functional block (program memory, register file, ALU, and data memory). Consequently, the controller must also change so that control signals are generated using data from the appropriate instruction register. A block diagram of the RP pipelined model is shown in Figure 9. It should be noted that two additional operand registers (not shown) were added in the ALU process to separate the operand fetch stage (register file) from the ALU stage of the pipeline. Approximately 1 person week was needed to implement PM.

To demonstrate the functioning of the pipelined model, we consider the RP code example from Section 2.3. The ADD_SUB_IMMED instruction assumed to be in Stage 1 of the pipeline, the MULT_REG_WORD instruction in Stage 2, the STORE_I instruction in Stage 3, and the BRANCH_IMM instruction in Stage 4.

instruction1: ADD_SUB_IMMED, R5, 50, R5; – *add immediate*
instruction2: MULT_REG_WORD, R1, R2, R3; – *multiply register*
instruction3: STORE_I, R3, R4; – *store indirect*
instruction4: BRANCH_IMM, R5, 151, -3; – *branch immediate*

The control signals issued as this time are as follows. The *pmem_read* and *pmem_enable* signals are both set HIGH since we wish to fetch the ADD_SUB_IMMED instruction into instruction register IR1. The *pmem_addr* bus contains the current value of the PC. The main register file control signal *RF_instr* is set to REG2 because the MULT_REG_WORD instruction is in IR2 (Stage 2 of the pipeline). The register file signal *RF_dest* is set to NO_DESTINATION because no data from the ALU or from the data memory will be stored into the register file during the present clock cycle. The current ALU operation is *ALU_instr* = PASS. This is due to the fact that the STORE_I instruction in Stage 3 (IR3) does not require any ALU computation, and the default value for *ALU_instr* is PASS. Finally, the *dmem_enable* signal is set LOW because no memory operation should take place in Stage 4 at the present time. (Data memory control is generated from IR4 which currently contains the BRANCH_IMM instruction.)

The PM model was also tested on the Livermore Loop benchmarks. As in the case of NPM, both real-time (hr:min:sec needed for the simulation to complete) and simulated-time (number of ns needed to execute an RP program on PM) results were recorded. A comparison of PM and NPM simulation results is provided in Section 6; however, we note here that the clock period in the PM model is reduced to 50 *ns*, from 200 *ns* in NPM.

6 Benchmarks and Experimental Results

To demonstrate the effectiveness of our top-down modeling technique, we simulated the RP models (specification level, non-pipelined functional, and pipelined functional) on a SPARC 1 workstation using the ZYCAD VHDL simulator, "zvsim," version 1.0a. The simulation results were used as a basis for comparing the different models. They also gave us a quantitative estimate of the performance gain due to design features such as VAR, and automatic loopback. In this section, we describe the benchmarks and experimental setups used for the simulations, and present experimental results.

6.1 Livermore Loop Benchmarks

We tested the performance of the models, in terms of both real time and simulated time, by executing three loops taken from the Livermore C Kernel. Livermore Loops are standard benchmarks used in supercomputing applications. Loops 1, 4, and 19 were selected as test cases for the RP models. The RP features (such as the VARs) support DSP applications and the selected loops are representative of such applications as they involve computations on large arrays of data. It should be noted that the Livermore Loops are written in C, and

```
main()
{
        register int k;
        double x[1002], y[1002], x[1002];
        double r, t;
        register double q;

        r=4.86;
        t=276.0;
        q=0.0;
        for (k=1; k<=400; k++)
                x[k]=q+y[k]*(r*z[k+10]+t*z[k+11]);
}
```

Figure 23: Livermore Loop 1: Hydro excerpt.

```
main()
{
        register int lw;
        register int j, l;
        double x[1002], y[1002];

        for (l=7; l<=107; l+50) {
                lw=l;
                for (j=30; j<=870; j+=5)
                        x[j-1]- =x[lw++]*y[j];
                x[l-1]=y[5]*x[l-1];
        }
}
```

Figure 24: Livermore Loop 4: Banded linear equations.

translation from C to the RP instruction set has been carried out manually.

Although the three loops do not test the instruction set of the RP exhaustively, they incorporate a large number of the RP features. For example, VARs are used as source and destination operands, the automatic loopback mechanism is tested and the hardware bypass feature is exercised for both VAR and non-VAR operations. The Livermore Loop benchmarks include floating point as well as double precision datatypes. As signal processing operations only require these datatypes to a limited extent, we have found it sufficient to use integer variables (rather than floating point, double precision) in our translations.

The C programs for Livermore Loops 1, 4, and 19 are depicted in Figures 23, 24, and 25. The loops essentially, consist of array computations iterated within one or more loops. LL#1 has only one loop, while LL#4 contains two nested loops. LL#19 consists of two inner loops nested within a single outer loop. Since data initially does not reside in RP, a large percentage of LL#1 is spent in data communication while the major portion of

```
main()
{
      long k, l, i, kb5i;
      double sa[101], sb[101], b5[101], stb5;

      kb5i=0;
      for (l=1; l<=1000; l++) {
            for (k=0; k<101; k++) {
                  b5[k+kb5i]=sa[k]+stb5*sb[K];
                  stb5=b5[k+kb5i]-stb5;
            }
            for (i=0; i<101; i++) {
                  k=101-i;
                  b5[k+kb5i]=sa[k]+stb5*sb[K];
                  stb5=b5[k+kb5i]-stb5;
            }
      }
}
```

Figure 25: Livermore Loop 19: General linear recurrence equations.

LL#19 involves data computations. Thus, LL#1 can be classified as data communication intensive, while LL#19 is data computation intensive.

6.2 Experimental Results

In this section, we discuss the performance of the RP models in terms of both real time and simulated time. Real time is defined as the *CPU* time (in hours:minutes:seconds) required to complete a simulation. (Crudely put, this is the time the designer would have to wait to get the results of a simulation after having started it.) On the other hand, the simulation time is the time in *ns* taken by the RP to complete the program execution. Thus, if the simulation time for one of the experiments is 1000 ns with clock period equal to 25 ns, it implies that the RP required 40 clocks to execute the program.

There are three premises that we wish to verify via our experiments. The first is to show feasibility of the modeling strategy presented in this report. In other words, we show that the real-time needed to simulate our models (SLM, NPM, and PM) is not prohibitively large. The second is to demonstrate: (1) that the hardware interfaces introduced in the refined SLM model enhance RP performance, and (2) that introduction of pipelining produces the expected speedup. The third is to show how design features such as VAR, automatic loopback, and external memory affect RP performance.

It should be noted that to validate our premises we have gathered experimental data from running simulations on three of the models described in this report: SLM (specification level model without interfaces), NPM (non-pipelined functional level model), and PM (pipelined

Model Type	Livermore Loop #1				Livermore Loop #4			
	Commun. (hr:min:sec)	Comput. (hr:min:sec)	Total (hr:min:sec)	$\frac{Inst.}{sec.}$	Commun. (hr:min:sec)	Comput. (hr:min:sec)	Total (hr:min:sec)	$\frac{Inst.}{sec.}$
SLM	00:55:59	00:02:09	01:00:35	13.95	00:24:09	00:03:45	00:32:40	10.19
NPM	01:56:12	00:13:34	02:13:14	2.01	00:42:41	00:21:30	01:03:55	1.85
PM	01:58:08	00:18:52	02:17:05	1.95	00:53:48	00:26:51	01:09:53	1.69
Spdup. SLM over NPM	51.82 %	84.15 %	54.53 %	-	43.42 %	82.56 %	48.89 %	-
Spdup. NPM over PM	1.64 %	28.09 %	2.81 %	-	20.66 %	19.93 %	8.54 %	-

Table 1: Real-time performance of RP models.

functional level model). We use NPM rather than a refined SLM to illustrate the difference between software subroutines and hardware interfaces for communication with peripheral devices. We divide our discussion of results into three sections, one for each premise.

6.2.1 Feasibility of Modeling Methodology

In this section, we demonstrate the real-time feasibility of our modeling methodology. The real-time performance of the three RP models on Livermore Loop benchmarks 1 and 4 is shown in Table 1. For each loop, the time required for communication, (*i.e.* data transfer from HOST to RP and from RP back to HOST), computation (execution of the loop on the data received and stored in RP data memory) is listed along with the total time taken for the complete (both communication and computation) simulation. It should be noted that for Livermore Loop 1 (Livermore Loop 4), communication involves transfer of 800 (439) words from HOST to RP and 400 (3) words from RP to HOST.

As shown in Table 1, the number of instructions per second executed by the RP models decreases with model complexity. For example, SLM executes about 7 times as many instructions per second as NPM; whereas, NPM and PM execute approximately the same number of instructions per second. From Table 1, we can approximate the amount of time needed to simulate a program on one of the RP models using the $\frac{instruction}{second}$ ratios provided.

6.2.2 Design Refinement

The simulation-time performance of the specification level, non-pipelined functional, and pipelined functional models on Livermore Loops 1 and 4 is shown in Table 2. Note that, for both loops, the computation times taken by SLM and NPM are equivalent, while the communication times for NPM are, on the average, 73.03 % faster than those for SLM. This is due to the fact that a hardware interface for HOST communication has been introduced into NPM, while HOST communication is performed using software routines in

Model Type	Livermore Loop #1			Livermore Loop #4		
	Commun. (ns)	Comput. (ns)	Total (ns)	Commun. (ns)	Comput. (ns)	Total (ns)
SLM	9,896,000	243,300	10,139,300	3,679,300	314,100	3,993,400
NPM	2,965,800	243,300	3,209,100	1,103,200	314,100	1,417,300
PM	741,900	60,975	802,875	275,950	78,775	354,725
Spdup. NPM over SLM	70.03 %	00.00 %	68.35 %	70.02 %	00.00 %	64.51 %
Spdup. PM over NPM	74.98 %	74.94 %	74.98 %	74.99 %	74.92 %	74.97 %

Table 2: Simulation-time performance of RP models.

SLM. In other words, the RP can perform data computation in parallel (at least partly) with HOST/SRT communication.

A similar improvement in performance is noted for PM over NPM; however, both communication and computation times are affected. Introduction of a four-stage pipeline in the PM model resulted in an average speedup of 74.98 % over the NPM model. In other words, the pipelined model is about four times ($\frac{NPM}{PM} \approx 4$) as fast in terms of simulated time as the non-pipelined model. This is due to the fact that pipelining increases the utilization of the functional blocks in each stage by allowing the blocks in each stage to execute a different instruction simultaneously.

6.2.3 Performance Gain due to RP Hardware Features

In this section, we demonstrate the effects of VAR, automatic loopback, and external memory on RP performance. The SPLM model is used to test the performance of these features due to its fast real-time execution speed.

Table 3 shows the performance of SPLM on Livermore Loop benchmarks 1, 4, and 19 (both with and without VARs) in terms of simulated time. It should be noted that the automatic loopback feature was used in both of the test cases (with and without VARs). Experimental results show that, for Livermore Loops 1, 4, and 19, a simulated time speedup of 53.03%, on the average, is obtained if VARs are used for array data references.
The performance improvement generated by the use of VARs is due to the fact that, in the absence of VARs, an additional *load* instruction is required for each memory location used as a source operand, and a *store* operation for each memory location used as a destination operand.

We also demonstrate that an average speedup of 49.24% can be achieved by using the automatic loopback feature for inner loops in the Livermore Loop benchmarks. Table 4

Loop No.	With VARs Simulated Time (ns)	Without VARs Simulated Time (ns)	Speedup
1	243,300	639,500	61.95%
4	314,100	612,100	48.69%
19	125,602,600	243,601,800	48.44%

Table 3: Vector access register performance.

Loop No.	With Loopback Simulated Time (ns)	Without Loopback Simulated Time (ns)	Speedup
1	243,300	483,000	49.63%
4	314,100	615,500	48.97%
19	125,602,600	246,802,600	49.11%

Table 4: Automatic loopback performance.

shows the performance of SPLM on Livermore Loops 1, 4, and 19 both with and without the automatic loopback feature. Note that, in both test cases, (with and without automatic loopback) vector access registers were used for array data.
The speedup induced by the automatic loopback mechanism is due to the fact that automatic loopback eliminates the need for test, increment, and jump instructions (loop overhead) at each loop iteration.

SLM was also tested on Livermore Loops 1 and 19 in the case where data is taken from and stored to external memory. We assume that the data initially resides in the external memory. For each iteration of the loop, data is fetched from the external memory, and after the loop computation, results are returned to external memory. In other words, internal memory is never used during execution of the loop.

Communication for Livermore Loop 19 involves transfer of 200 (100) words of data from external memory (RP) to RP (external memory), and communication for Livermore Loop 1 is described in Section 6.2.1. With an external memory speed of 100 ns, the real time required for simulating Livermore Loop #1 (#19, 500 iterations) is 00hr:18min:33sec (57hr:49min:09sec). Note that when internal memory is used for Livermore Loop #1, only 00hr:02min:9sec is needed for loop execution. Hence, loop execution is about 9 times slower when external memory, as opposed to internal memory, is used.

7 Conclusions

In this report, we present a top-down modeling methodology for RISC processors. We discuss four basic models: specification level, refined specification level, functional level (non-pipelined), and functional level (pipelined).

In a SLM, we modeled only functionality as described in the English language specification. For a processor such as RP, this was its instruction set with communication protocols for interfacing with the outside world. The main purpose of the specification level model is to provide high-level documentation of the design, which represents a starting point for design management and marketing and facilitates lifetime maintenance of the product. Also, in the case of a processor, a SLM can be used by compiler designers for compiler validation. In a refined SLM, communication protocols were modeled more accurately, including timing relationships between signals. In the case of embedded designs, system designers may use the refined SLM to test communication and timing between the device and the environment in which it is embedded.

In a functional level model (non-pipelined case), the design is decomposed into several functional blocks which communicate via a well-defined interconnection structure. After the FLM is finalized, design teams may work concurrently on the different functional blocks. Due to the well-defined functional block and interconnection structure as well as the extensive simulation, changes and errors in the completed design are reduced. The FLM functional blocks may be used to construct an initial floorplan for the design and estimate design time for each block. Hardware estimators can also be used to approximate the area as well as performance of the functional blocks. A pipelined FLM is simply a refined version of the non-pipelined FLM developed to improve the performance (in terms of clock speed, critical path) of the non-pipelined version.

We developed the SLM for RP in 4 person weeks and NPM and PM models in an additional 8 weeks, from which we obtained an approximate chip layout. Industrial design practice shows that design details at the functional level are typically known in 8-12 months, while the first layout is obtained in 12 to 18 months. Thus, a proper modeling methodology may reduce design time by a factor of two, because architectural and performance tradeoffs can be made during the functional modeling phase rather than after 12 months of design time.

We believe that our top-down modeling technique will dramatically reduce design time, especially for future revisions of the unit under design. Since this technique provides documentation of the entire design process (from high level to low level), we believe that the cost of maintenance throughout the life cycle of the design will be greatly reduced; however, an

overhead is incurred for maintenance of the models themselves. For example, design changes made at low levels must propagate to the higher level models. This extra effort, however, spent in maintaining the models is more than compensated by reduction in design time, documentation, and number of design errors.

Acknowledgements

This work was partially supported by the Semiconducter Research Corporation grant #91-DJ-269, and we gratefully acknowledge their support. We also extend our gratitude to Bob Larson, George Watson, and Don Harenberg at Rockwell International for their invaluable assistance in processor design. Furthermore, we would like to thank Viraphol Chaiyakul, Loganath Ramachandram and Jian Li for providing us with VHDL descriptions of the clock divider, timer, and SRT interface. We also thank Frank Vahid and Sanjiv Narayan for their insightful comments regarding VHDL modeling.

References

[1] D. D. Gajski, Editor, *Silicon Compilation,*, Addison-Wesley, 1988.

[2] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.

[3] J. S. Lis, *Ph.D. Dissertation*, University of California, 1990.

[4] J. S. Lis and D. D. Gajski, *Structured Modeling for VHDL Synthesis,* Technical Report #89-14, University of California, 1989.

[5] D. L. Perry, *VHDL,* McGraw-Hill, Inc., 1991.

[6] B. Preas and M Lorenzetti, *Physical Design Automation of VLSI Systems,* Benjamin/Cummings, 1988.

[7] Z. Nawabi, *VHDL: Analysis and Modeling of Digital Systems,* McGraw-Hill, Inc., 1993.

[8] *Standard VHDL Language Reference Manual,* New York: The Institute of Electrical and Electronics Engineers, Inc., 1988.