High Performance Discrete Event Simulation

Fred Love and Rebecca Curtis, Missouri University of Science and Technology

Abstract—Supercomputing paradigms and high performance computers have played an intricate role in bridging the gap between application demands and what technology can provide. The emergence of the CDC600 supercomputer marks the first major gain towared narrowing the computation gap. This paper explores the performance features of the CDC6600 as well as the enhancements provided by its successor, the CDC7600. A discrete event simulator written in SystemC will be used the vehicle of exploration.

Index Terms—High Performance Computing, Performance Measures, Hardware Simulation

1 Introduction

THE demand for increasing computational capability continues to necessitate the development of higher performance computer technology. A survey of the computational requirements for various scientific applications suggests that existing computers are insufficient to handle future workloads. This trend requires the advancement of higher performance computing systems to both address these shortcomings and to keep pace with successive applications. This paper briefly explores the genesis of applying supercomputer technology, and then describes a SystemC based event simulation engine for two pioneering architectures.

The next section will offer an overview of the CDC6600 and CDC7600 to provide reader with foundation necessary to discuss the event simulator. Section 3 briefly introduces the SystemC library to facilitate exploring implementation details of the simulator. Section 4 closely examines the simulator implementation and mock execution results for select program inputs. Lastly, the paper concludes with a few words the need for greater computational ability.

2 Cray SuperComputers

2.1 CDC6600

1964 marked the genesus of the supercomputer with the development of the CDC6600. Its design philosophy relied on a highly parallel, multifunctional CPU that is isolated from I/O operations. The CPU was supported by an array of peripheral processors to handle these I/O operations. The memory subsystem is organized in a hierarchical fashion that employs interleaving. The system-level block diagram shown in Figure 1 highlights these attributes:

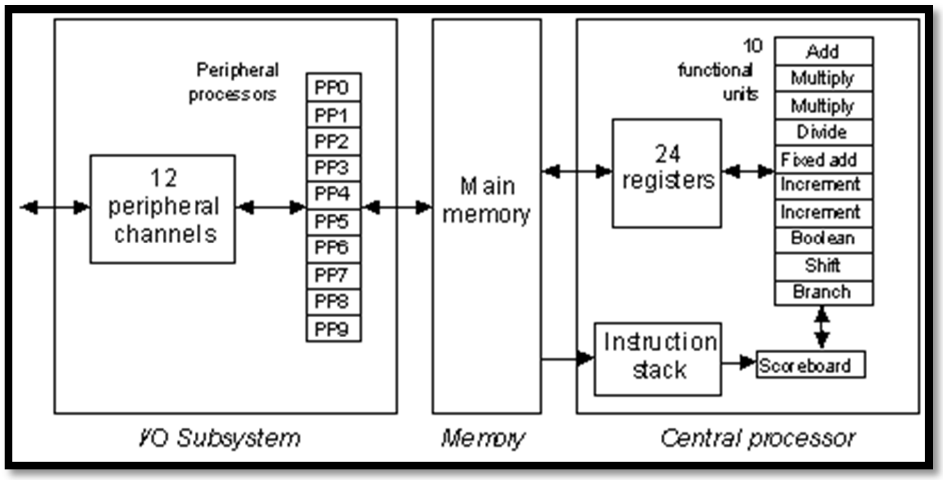


Figure 1: CDC6600 Block Diagram

The initial delivery of the CDC6600 had an estimated performance of approximately 1 MFLOPS. This represented a 3X increase over its computing predecessor, the IBM 7030.

2.2 CDC7600

The five year reign as world's fastest computer held by the CDC6600 ended in 1969 with the introduction of the CDC 7600. Similar to its predecessor the CDC7600 utilizes a multifunctional processor, separates I/O operations from CPU operations, and deploys hierarchical memory organization with interleaving. Contrastly, the CDC7600 applies pipelined functional units with the exception of the divide unit. Figure 2 presents the organization:

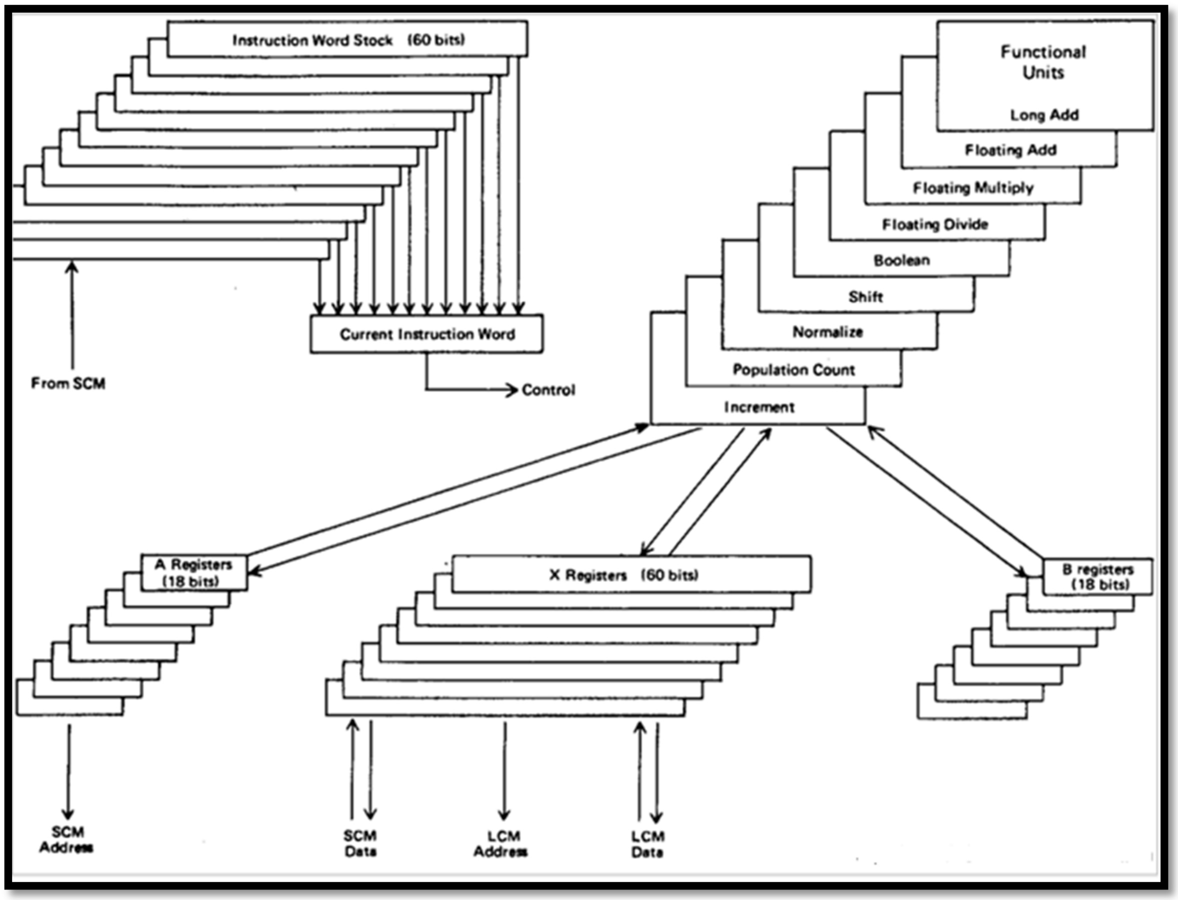


Figure 2: CDC7600 Block Diagram

Upon its introduction, the CDC7600 boasted an approximate 8X performance increase over the incumbent CDC6600. The memory hierarchy hosted main memory that offered 3X performance improvement along with 2X performance increase given by backup memory.

The following section will briefly introduce the SystemC library in preparation for a more detailed discussion of the CDC6600/7600 discrete event simulator.

3 Overview of SystemC

Although commonly referred to an independent language, System C is actually a library that includes a set of classes, data types, and macros built on-top of C++ that allow simulation of concurrent processes. This it makes it an ideal choice for modeling hardware which frequently exhibits concurrent behavior. In addition to syntax support, the SystemC architecture offers a native simulation kernel that utilizes components such as modules, communication channels, and events. The complete SystemC architecture is depicted in Figure 3.

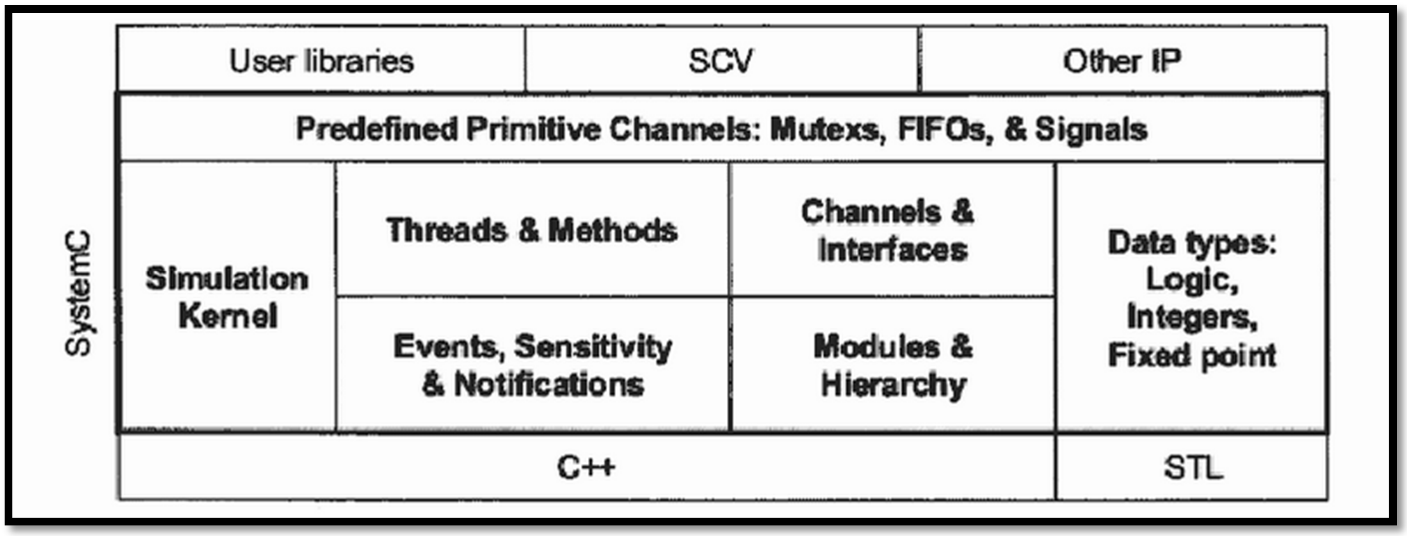

User libraries
SCV
Other IP
Predefined Primitive Channels: Mutexs, FIFOs, & Signals
SystemC
Simulation Kernel
Threads & Methods
Channels & Interfaces
Data types: Logic, Integers, Fixed point
Events, Sensitivity & Notifications
Modules & Hierarchy
C++
STL

Figure 3: SystemC Library Architecture

These features have allowed the library to become a good choice for system-level modeling, functional verification, and high-level synthesis. When accompanied with industrial tools such as Cadence's Cynthesizer, a SystemC model may be compiled into hardware described by a register transfer language (RTL).

The SystemC architecture and its features are built upon the simulation kernel. The kernel has two primary phases of operation: elaboration and execution. The elaboration phase is characterized by establishing module/channel connectivity and the initialization of all data structures. The execution phase is marked by handing control to the kernel to coordinate concurrent process activity. The distinction between the two phases is a call to the *sc_start* function. All statements included before this call belong to the elaboration phases while the remaining are invoked during the execution phases by the simulation kernel. Figure 5 illustrates this process.

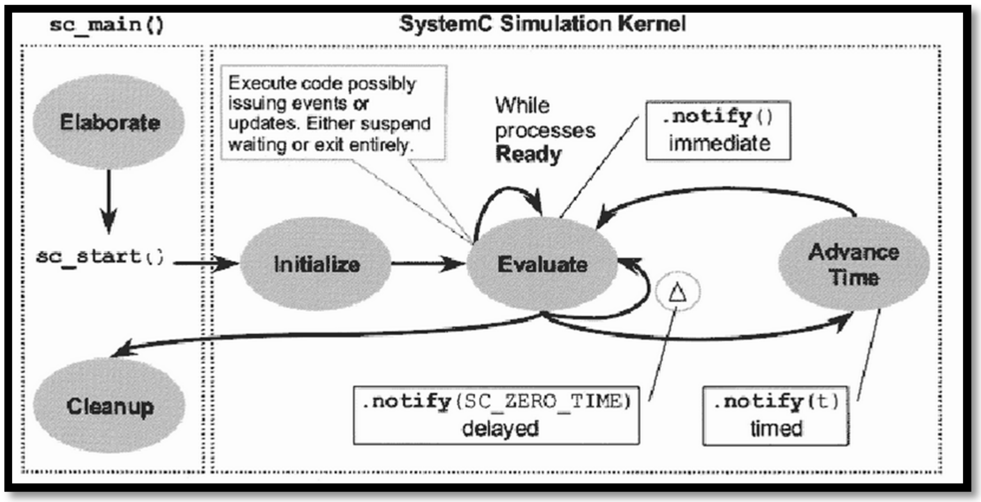

sc_main()
SystemC Simulation Kernel
Elaborate
sc_start()
Cleanup
Execute code possibly issuing events or updates. Either suspend waiting or exit entirely.
While processes Ready
.notify() immediate
Initialize
Evaluate
Advance Time
Δ
.notify(SC_ZERO_TIME) delayed
.notify(t) timed

Figure 5: SystemC Simulation Kernel

The following section will discuss the details of the CDC6600/7600 SystemC event simulator and present results.

4 SIMULATOR IMPLEMENTATION

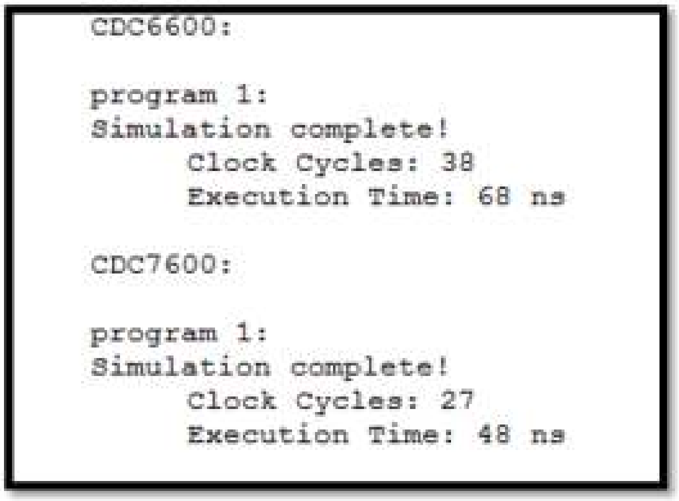
Both the CDC6600 and CDC7600 were modeled using several SystemC modules representing processor components that communicated through the use of primitive channels listed in Figure 3. For example, a scoreboard module exhibited hierarchical communication and control by using separate instruction, hazard, and unit select fifos. This enabled the scoreboard to process decoded instructions and monitor for potential conflcits. For clarity, a code segment for an abbreviated top level scoreboard module is listed in Figure 6. FIFO and unit instantiation has been highlighted both for readability and to emphasize the use of hierarchy and communication channels:

```
SC_MODULE (SCOREBOARD){

     //ports, processes, internal data
     sc_in_clk clock;
     sc_inout< bool > end;
     sc_fifo<Instruction> fifo_buffer;
     sc_fifo<Instruction> hazard_buffer;
     sc_fifo<Operation> unit_select_buffer;
     sc_inout< bool > is_CDC6600;

     //functional units
     MULTIPLIER * m_mult;
     DIVIDER * m_div;
     FIXED_ADD * m_fixed_add;
     FLOATING_ADD * m_floating_add;
     INCREMENTER * m_inc;
     SHIFTER * m_shift;
     BOOLEAN * m_bool;
     BRANCHER * m_branch;

     //Functional unit status register - fred
     FUNC_UNIT_STATUS * unit_stat_reg;
     bool wait;
     int wait_time;
     Instruction cur_instr;
     bool fetch_instr;

     //Constructor
     SC_CTOR(SCOREBOARD){

          fetch_instr = false;
          //attempt to issue every clock cycle
          SC_METHOD (issue_stage);
          sensitive << clock.pos();

          //Instruction input buffer
          sc_fifo<Instruction> fifo_buffer (2);
          sc_fifo<Instruction> hazard_buffer (32);
          sc_fifo<Operation> unit_select_buffer (32);
          //initializing functional units
          m_mult = new MULTIPLIER("Multiplier0");
          m_mult->clock(clock);
          m_mult->is_CDC6600(is_CDC6600);

          m_div = new DIVIDER("Divider0");
          m_div->clock(clock);;
          m_div->is_CDC6600(is_CDC6600);

          //Adding functional unit status register
          unit_stat_reg = new FUNC_UNIT_STATUS ("Func_Unit_Status_Reg0");
          unit_stat_reg->clock(clock);
          unit_stat_reg->end(end);

     }//end SC_CTOR

};//end SCOREBOARD
```

Figure 6: Scoreboard SystemC Module

This patterned use of module hierarchy and channels was followed throughout the design.

Three selectable programs were optioned to the user to be simulated against either the CDC6600 or CDC7600. Because of the performance gains described in section 2, the CDC7600 is expected to execute the same program in less time. The simulator proved accurate after executing all three programs on both computers. The simulation results for the first program is shown in Figure 7:



```
CDC6600:

program 1:
Simulation complete!
        Clock Cycles: 38
        Execution Time: 68 ns

CDC7600:

program 1:
Simulation complete!
        Clock Cycles: 27
        Execution Time: 48 ns
```

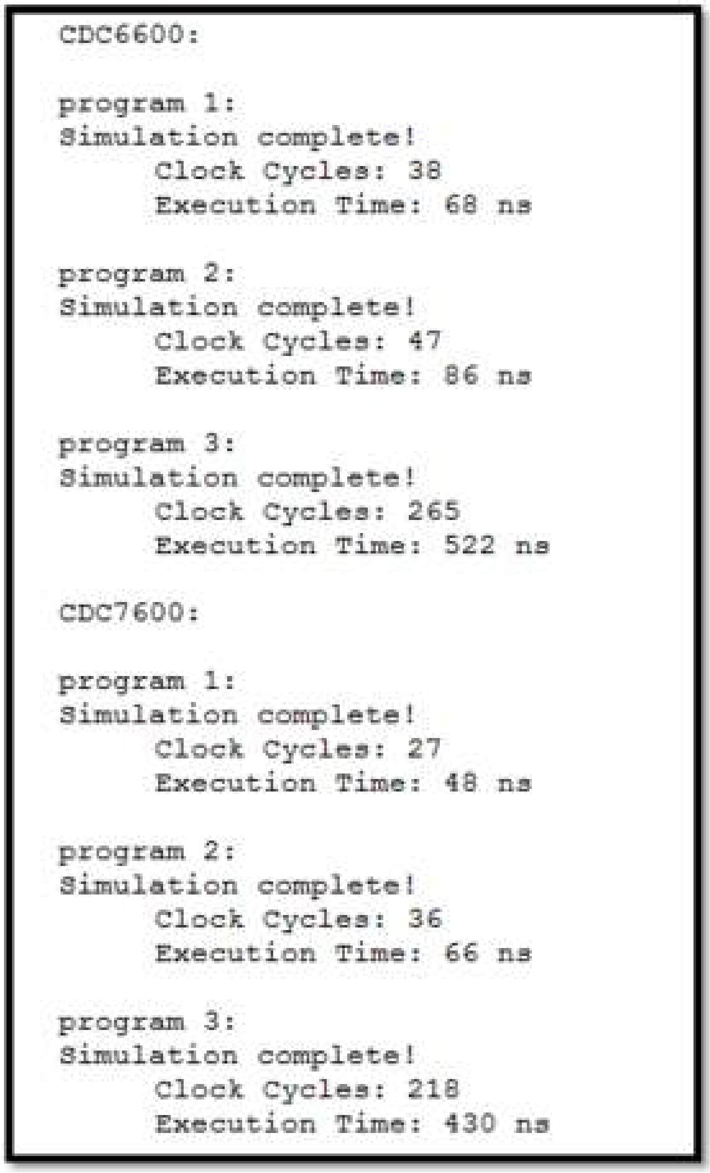
Figure 7: Program 1 Simulation Results

As expected the CDC7600 outperforms it predecessor. The complete simulation report is included in the Appendix.

5 CONCLUSION

As emerging applications continue to require greater computational capability, the trend towards developing high performance computers much more powerful than the previous generation must also continue. The progression from the CDC6600 to its successor was a first foray into the necessary evolution of high performance computing. The SystemC discrete event simulator presented highlights the performance gains associated with the next-generation supercomputer of that time.

APPENDIX



```
CDC6600:

program 1:
Simulation complete!
      Clock Cycles: 38
      Execution Time: 68 ns

program 2:
Simulation complete!
      Clock Cycles: 47
      Execution Time: 86 ns

program 3:
Simulation complete!
      Clock Cycles: 265
      Execution Time: 522 ns

CDC7600:

program 1:
Simulation complete!
      Clock Cycles: 27
      Execution Time: 48 ns

program 2:
Simulation complete!
      Clock Cycles: 36
      Execution Time: 66 ns

program 3:
Simulation complete!
      Clock Cycles: 218
      Execution Time: 430 ns
```