Appendix B HECTOR 1600 Data Path Simulator

The HECTOR 1600 is a 16-bit microprocessor built in a 4-month time-span by five graduate students and one faculty advisor at North Caroline State University. Due to the short design cycle and engineering inexperience, HECTOR was built as simply as possible. However, the simple design not only enabled the machine to be built by the required deadline, but also caused the instruction set to become orthogonal, powerful, and easy-to-use.

Introduction

This appendix describes the BSVC version of the HECTOR 1600 data path simulator. The BSVC version is based on the HECTOR 1600 simulator written by Greg DeHoogh. The simulator is broken into five C++ classes: `Hector`, `DataPath`, `RegisterSet`, `ALU`, and `ControlUnit`. The layout of these classes is shown in Figure 13. Each of these classes will be discussed in the following sections.

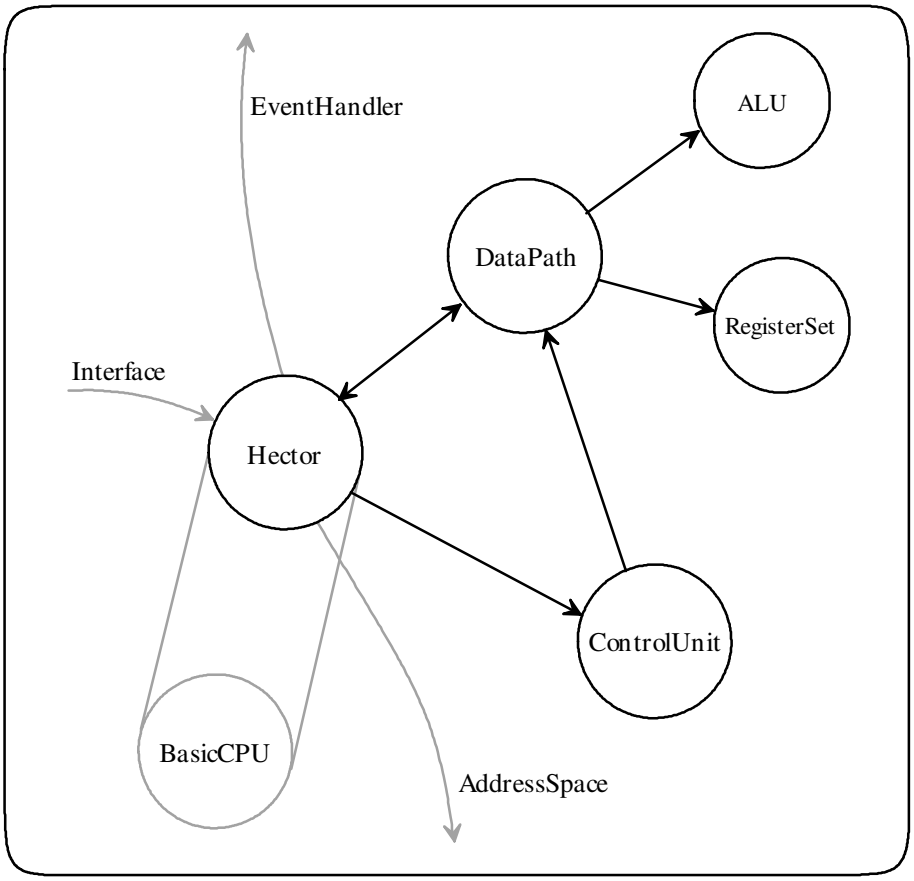

FIGURE 13 HECTOR 1600 Simulator Class Layout

Hector Class

The `Hector` class is derived from the framework's `BasicCPU` class. It provides all the functionality needed by the BSVC framework. The most important part of this class is the `ExecuteInstruction` method that fetches and executes the next instruction.

Hector Class Declaration

```
 1  ////////////////////////////////////////////////////////////////////////////
 2  //
 3  // Hector.hxx - This is the Hector 1600 CPU class
 4  //
 5  // By: Bradford W. Mott
 6  // December 2,1993
 7  //
 8  ////////////////////////////////////////////////////////////////////////////
 9
10  #ifndef HECTOR_HXX
```

```
11  #define HECTOR_HXX
12
13  #include "BasicCPU.hxx"
14  #include "BasicDevice.hxx"
15
16  #include "ControlUnit.hxx"
17
18  ////////////////////////////////////////////////////////////////////////////
19  // The Hector 1600 class
20  ////////////////////////////////////////////////////////////////////////////
21  class Hector : public BasicCPU {
22    private:
23      DataPath*    data_path;        // Hector's data path
24      ControlUnit* control_unit;     // Hector's control unit
25
26    public:
27      Hector(AddressSpace*);         // Constructor
28      virtual ~Hector();             // Destructor
29
30      // Execute a single instruction: On success it returns NULL, on
31      //     failure it returns a pointer to the reason.
32      const char* ExecuteInstruction(String& trace_record, int trace_flag);
33
34      // Handle an interrupt request from a device
35      void InterruptRequest(BasicDevice* device, int level);
36
37      // Preform a system reset
38      void Reset();
39
40      // Return the name of the program counter register
41      char* const NameOfProgramCounter();
42
43      // Return the value of the program counter register
44      unsigned long ValueOfProgramCounter();
45
46      // Set the named register to the given value
47      void SetRegister(String name, String hex_value);
48
49      // Clear the statistics
50      void ClearStatistics();
51
52      // Build the register list for the RegisterInformationList object
53      void BuildRegisterInformationList(RegisterInformationList*);
54
55      // Build the statistics list for the StatisticalInformationList object
56      void BuildStatisticalInformationList(StatisticalInformationList*);
57  };
58  #endif
59
```

DataPath Class

The `DataPath` class models the HECTOR 1600's data path shown in Figure 14. The `DataPath` class provides methods to reset the data path's signals, set the data path's signals, and perform a clock cycle. The register transfer methods in the `ControlUnit` set up the appropriate register transfer sequences in the `DataPath` and then call the `Clock` method which performs a clock cycle operation on the data path.

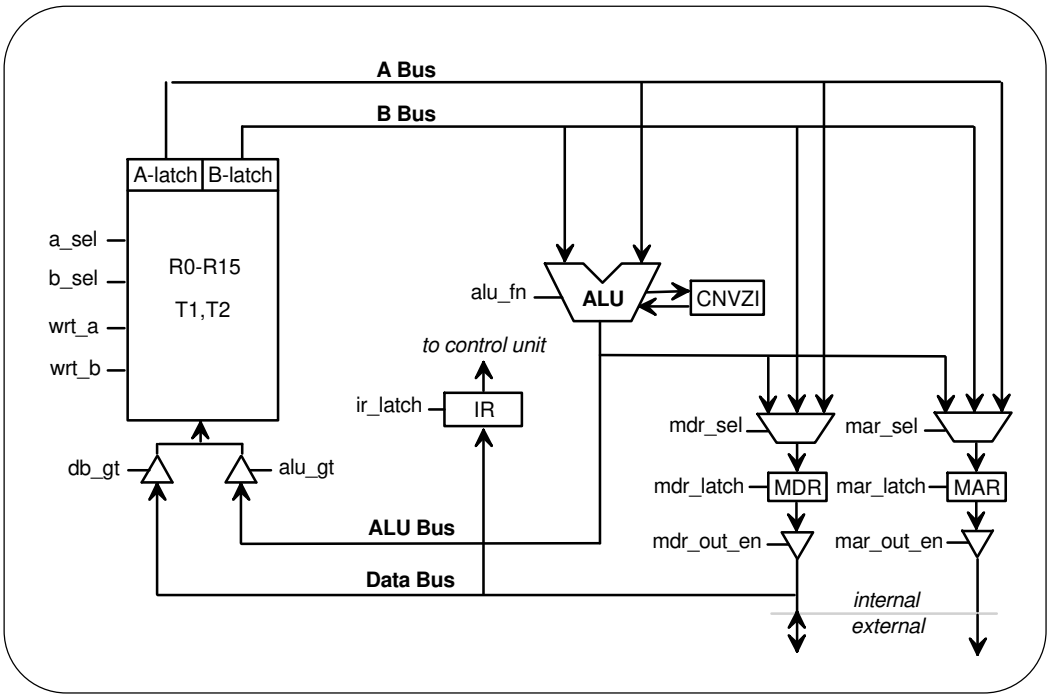

FIGURE 14 HECTOR 1600 Data Path

DataPath Class Declaration

```
1  ////////////////////////////////////////////////////////////////////////////
2  //
3  // DataPath.hxx - Data Path for the Hector 1600 CPU
4  //
5  // By: Bradford W. Mott
6  // December 3,1993
7  //
8  ////////////////////////////////////////////////////////////////////////////
9
10 #ifndef DATAPATH_HXX
11 #define DATAPATH_HXX
12
13 #include "BasicCPU.hxx"
14 #include "ALU.hxx"
15 #include "RegisterSet.hxx"
16
17 enum DataPathBus { NONE, A_BUS, B_BUS, ALU_BUS };
18
19 ////////////////////////////////////////////////////////////////////////////
20 // The Data Path for the Hector 1600 CPU
21 ////////////////////////////////////////////////////////////////////////////
22 class DataPath {
23   private:
24     BasicCPU*    cpu;              // Pointer to the CPU
25
26     int          A_SEL;            // A BUS Register Select
27     int          B_SEL;            // B BUS Register Select
28     unsigned int WRT_A;            // Write Register A
29     unsigned int WRT_B;            // Write Register B
30     unsigned int DB_GT;            // Data Bus Gate
31     unsigned int ALU_GT;           // ALU Bus Gate
32     ALUFunction  ALU_FN;           // ALU Function Select
33     DataPathBus  MDR_SEL;          // Mux value for Memory Data Register
```

```
34        unsigned int MDR_LATCH;         // Memory Data Register Latch
35        unsigned int MDR_OUT_EN;        // Memory Data Register Output Enable
36        unsigned int MAR_LATCH;         // Memory Address Register Latch
37        unsigned int MAR_OUT_EN;        // Memory Address Register Output Enable
38        DataPathBus  MAR_SEL;           // Mux value for Memory Address Register
39        unsigned int IR_LATCH;          // Instruction Register Latch
40
41        unsigned int a_bus;             // Value on the A Bus
42        unsigned int b_bus;             // Value on the B Bus
43        unsigned int alu_bus;           // Value on the ALU Bus
44        unsigned int data_bus;          // Value on the Data Bus
45        unsigned int wrt_bus;           // Value on the Write Bus
46
47        unsigned long number_of_reads;  // Number of memory reads
48        unsigned long number_of_writes; // Number of memory writes
49        unsigned long number_of_cycles; // Number of cycles
50
51        // Read a word from memory
52        int Peek(unsigned long address, unsigned int& value);
53
54        // Write a word to memory
55        int Poke(unsigned long address, unsigned int value);
56
57      public:
58        RegisterSet  register_set;      // The register set
59        ALU          alu;               // Hector's ALU
60        unsigned int ir;                // Instruction Register
61        unsigned int mar;               // Memory Address Register
62        unsigned int mdr;               // Memory Data Register
63
64        DataPath(BasicCPU*);
65        ~DataPath();
66
67        // Reset the data path's signals
68        void ResetSignals();
69
70        // Called at the end of each cycle to perform the actual cycle operations.
71        //   Returns NULL or a pointer to an error message.
72        const char* Clock();
73
74        // Return the number of read memory accesses
75        inline unsigned long NumberOfReads()
76        { return (number_of_reads); }
77
78        // Return the number of write memory accesses
79        inline unsigned long NumberOfWrites()
80        { return (number_of_writes); }
81
82        // Return the number of cycles
83        inline unsigned long NumberOfCycles()
84        { return (number_of_cycles); }
85
86        // Clear the statistics (reads, writes, cycles)
87        inline void ClearStatistics()
88        { number_of_reads=number_of_writes=number_of_cycles=0; }
89
90        // Selects a register to be put on the A_BUS
91        void a_sel(int reg);
92
93        // Selects a register to be put on the B_BUS
94        void b_sel(int reg);
95
96        // Latch WRT_BUS into A register
97        void wrt_a();
98
99        // Latch WRT_BUS into B register
100       void wrt_b();
101
102       // Gate DATA_BUS onto WRT_BUS
103       void db_gt();
104
105       // Gate ALU BUS onto WRT_BUS
106       void alu_gt();
107
```

```
108       // Select an ALU Function
109       void alu_fn(ALUFunction function);
110
111       // Select Bus
112       void mdr_sel(DataPathBus bus);
113
114       // Latch selected bus contents into MDR
115       void mdr_latch();
116
117       // Allow MDR to drive the Data Bus
118       void mdr_out_en();
119
120       // Select Bus
121       void mar_sel(DataPathBus bus);
122
123       // Latch selected bus contents into MAR
124       void mar_latch();
125
126       // Allow MAR to drive the Address Bus
127       void mar_out_en();
128
129       // Latch IR
130       void ir_latch();
131   };
132   #endif
133
```

RegisterSet Class

The `RegisterSet` class models the register file in the HECTOR 1600 microprocessor. It provides methods to retrieve and set the value of a register. An object of this class is contained in the `DataPath` class.

RegisterSet Class Declaration

```
 1   ////////////////////////////////////////////////////////////////////////////
 2   //
 3   // RegisterSet.hxx - The Hector 1600's Register Set
 4   //
 5   // By: Bradford W. Mott
 6   // December 3,1993
 7   //
 8   ////////////////////////////////////////////////////////////////////////////
 9
10   #ifndef REGISTERSET_HXX
11   #define REGISTERSET_HXX
12
13   #include <assert.h>
14
15   // Register information structure
16   struct RegisterData {
17     char* name;
18     char* description;
19   };
20
21   // Indices into the register array
22   const int SWI=11;
23   const int NMI=12;
24   const int MSK=13;
25   const int SP=14;
26   const int PC=15;
27   const int T1=16;
28   const int T2=17;
29
30   ////////////////////////////////////////////////////////////////////////////
31   // The Hector 1600's Register Set Class
32   ////////////////////////////////////////////////////////////////////////////
33   class RegisterSet {
```

```
34      private:
35        // Number of user registers (User's view doesn't count T1 & T2)
36        const int number_of_user_registers;
37
38        // Total number of registers
39        const int number_of_registers;
40
41        // Array of static information for each user register
42        static RegisterData register_data[];
43
44        // Pointer to an array of values for each register
45        unsigned long *register_value;
46
47      public:
48        RegisterSet();
49        ~RegisterSet();
50
51        // Return the Number Of User Registers
52        inline int NumberOfUserRegisters()
53        { return(number_of_user_registers); }
54
55        // Return the Number Of Registers
56        inline int NumberOfRegisters()
57        { return(number_of_registers); }
58
59        // Get a Register value from the register file
60        inline unsigned long GetRegister(int index)
61        {
62          assert(index < number_of_registers);
63          return(register_value[index]);
64        }
65
66        // Store a value in one of the registers
67        inline void SetRegister(int index, unsigned long value)
68        {
69          assert(index < number_of_registers);
70          register_value[index]=value & 0xffff;
71        }
72
73        // Get the RegisterData structure for the indexed registers
74        inline RegisterData GetRegisterData(int index)
75        {
76          assert(index < number_of_user_registers);
77          return(register_data[index]);
78        }
79    };
80    #endif
81
```

ALU Class

The `ALU` class models the ALU in the Hector 1600 microprocessor. It provides methods to invoke the ALU and methods to test condition codes. It also contains methods to manipulate the ALU flags. An object of this class is contained in the `DataPath` class.

ALU Class Declaration

```
1   ////////////////////////////////////////////////////////////////////////////
2   //
3   // ALU.hxx - Arithmetic Logic Unit
4   //
5   // By: Bradford W. Mott
6   // December 3,1993
7   //
8   ////////////////////////////////////////////////////////////////////////////
9
```

```
10   #ifndef ALU_HXX
11   #define ALU_HXX
12
13   // ALU Function codes
14   enum ALUFunction {
15     ADD        = 0x00,   ADDC       = 0x01,   SUB        = 0x02,
16     AND        = 0x03,   SUBC       = 0x04,   OR         = 0x05,
17     XOR        = 0x06,   CMP        = 0x0a,   BTST       = 0x0b,
18     NOT_A      = 0x80,   NEG_A      = 0x81,   INC_A      = 0x82,
19     DEC_A      = 0x83,   SHL_A      = 0x90,   ROL_A      = 0x91,
20     SHR_A      = 0x92,   ROR_A      = 0x93,   PASS_A     = 0x100,
21     SWAP_A     = 0x101,  ADD_NCC    = 0x102,  INC_A_NCC  = 0x103,
22     INC_B_NCC  = 0x104,  DEC_A_NCC  = 0x105,  DEC_B_NCC  = 0x106,
23     A_CC       = 0x107,  CC_OUT     = 0x108,  PASS_A_NCC = 0x10a,
24     SUB_NCC    = 0x10b,  DO_NOTHING = 0xffff
25   };
26
27   // ALU Condition Codes
28   enum ALUConditionCode {
29     CC_VC = 0,  CC_PL = 1,  CC_GE = 2,  CC_F  = 3, CC_LE = 4,  CC_NE = 5,
30     CC_LS = 6,  CC_CC = 7,  CC_VS = 8,  CC_MI = 9, CC_LT = 10, CC_T  = 11,
31     CC_GT = 12, CC_EQ = 13, CC_HI = 14, CC_CS = 15
32   };
33
34   // ALU Flags
35   const unsigned long C_FLAG=0x8000;
36   const unsigned long V_FLAG=0x4000;
37   const unsigned long N_FLAG=0x2000;
38   const unsigned long Z_FLAG=0x1000;
39   const unsigned long I_FLAG=0x0800;
40
41   ////////////////////////////////////////////////////////////////////////////
42   // The Arithmetic Logic Unit
43   ////////////////////////////////////////////////////////////////////////////
44   class ALU {
45     private:
46
47       // ALU flags (16 bits: CVNZI-----------)
48       unsigned long flags;
49
50     public:
51       ALU()
52       { flags=I_FLAG; }
53
54       ~ALU()
55       {}
56
57       // Get the ALU's flags
58       inline unsigned long GetFlags()
59       { return(flags); }
60
61       // Set the ALU's flags
62       inline void SetFlags(unsigned long value)
63       { flags=value & 0xf800; }
64
65       // Compute a result with the given operation and inputs
66       unsigned long Compute(ALUFunction f, unsigned long a, unsigned long b);
67
68       // Test the given Condition Code
69       int TestConditionCode(ALUConditionCode cc, const char* &description);
70   };
71   #endif
72
```

ControlUnit Class

The `ControlUnit` class contains instruction execution methods, register transfer methods, and a method to execute the next instruction. The `Hector` class references an object of this class and uses it to execute instructions.

ControlUnit Class Declaration

```
1   ////////////////////////////////////////////////////////////////////////////
2   //
3   // ControlUnit.hxx - Control Unit for the Hector 1600 CPU
4   //
5   // By: Bradford W. Mott
6   // December 3,1993
7   //
8   ////////////////////////////////////////////////////////////////////////////
9
10  #ifndef CONTROLUNIT_HXX
11  #define CONTROLUNIT_HXX
12
13  #include "String.h"
14  #include "DataPath.hxx"
15
16  ////////////////////////////////////////////////////////////////////////////
17  // The Control Unit for the Hector 1600 CPU
18  ////////////////////////////////////////////////////////////////////////////
19  class ControlUnit {
20    private:
21      // Mnemonic to be added to trace record after opcode decoded and executed
22      String mnemonic;
23
24      // Routine to Execute instructions
25      const char* ControlUnit::ExecuteGroup1(int trace_flag);
26      const char* ControlUnit::ExecuteGroup2(int trace_flag);
27      const char* ControlUnit::ExecuteGroup3(int trace_flag);
28      const char* ControlUnit::ExecuteBRA(int trace_flag);
29      const char* ControlUnit::ExecuteJSR(int trace_flag);
30      const char* ControlUnit::ExecuteSWAP(int trace_flag);
31      const char* ControlUnit::ExecuteCLR(int trace_flag);
32      const char* ControlUnit::ExecuteMOVE(int trace_flag);
33      const char* ControlUnit::ExecuteTEST(int trace_flag);
34      const char* ControlUnit::ExecuteSTF(int trace_flag);
35      const char* ControlUnit::ExecuteLDF(int trace_flag);
36      const char* ControlUnit::ExecutePUSH(int trace_flag);
37      const char* ControlUnit::ExecuteSEC(int trace_flag);
38      const char* ControlUnit::ExecuteCLC(int trace_flag);
39      const char* ControlUnit::ExecuteSEI(int trace_flag);
40      const char* ControlUnit::ExecuteCLI(int trace_flag);
41      const char* ControlUnit::ExecuteRTI(int trace_flag);
42      const char* ControlUnit::ExecuteSWI(int trace_flag);
43      const char* ControlUnit::ExecuteEXCH(int trace_flag);
44      const char* ControlUnit::ExecuteSRCH(int trace_flag);
45
46      // Register transfer functions
47      const char* reg_to_reg(int reg_a, int reg_b);
48      const char* reg_indirect_to_reg(int reg_a, int reg_b);
49      const char* reg_to_reg_indirect(int reg_a, int reg_b);
50      const char* reg_to_mar_indirect(int reg);
51      const char* mar_indirect_to_reg(int reg);
52      const char* mdr_to_mar_indirect();
53      const char* inc_reg(int reg);
54      const char* reg_op_reg_to_reg(int reg_a, int reg_b);
55      const char* reg_op_reg_to_mdr(int reg_a, int reg_b);
56      const char* reg_plus_reg_to_reg(int reg_a, int reg_b);
57      const char* reg_plus_reg_to_mar(int reg_a, int reg_b);
58      const char* dec_reg(int reg);
59      const char* reg_op_reg(int reg_a, int reg_b);
60      const char* op_reg_to_reg(int reg);
61      const char* op_reg_to_mdr(int reg);
62      const char* swap_reg_to_reg(int reg_a, int reg_b);
63      const char* swap_reg_to_mdr(int reg);
64      const char* clr_reg(int reg);
65      const char* clr_mdr();
66      const char* flags_to_reg(int reg);
67      const char* flags_to_mdr();
68      const char* reg_to_flags(int reg);
69      const char* pass_reg_to_reg(int reg_a, int reg_b);
70      const char* reg_to_reg_indirect_with_inc(int reg_a, int reg_b);
71      const char* mdr_to_mar_indirect_with_inc(int reg);
72      const char* reg_to_reg_indirect_with_dec(int reg_a, int reg_b);
```

```
73
74        // Decode and execute the instruction in IR
75        const char* DecodeAndExecute(int trace_flag);
76
77      public:
78        // CPU's data path
79        DataPath& data_path;
80
81        // Class Constructor
82        ControlUnit(DataPath& d)
83            : data_path(d)
84        {}
85
86        ~ControlUnit()
87        {}
88
89        // Fetch, decode, and execute the next instruction
90        const char* ExecuteInstruction(String& trace_record, int trace_flag);
91
92    };
93    #endif
94
```

Modifying the Simulator

There are many possibilities for changes to the HECTOR 1600 design, because of this it is not possible to describe all the necessary modifications to the simulator to accommodate them. However, the comments that follow cover the general areas.

Once changes to the microprocessor design have been decided upon the first modifications will probably be to the `DecodeAndExecute` method of the `ControlUnit` class and the `Clock` method of the `DataPath` class. The changes to `DecodeAndExecute` will depend on changes to the instruction set; while the changes to `Clock` will depend on the changes to the data path.

Modifications should then proceed to adding instruction methods, deleting instruction methods, and modifying instruction methods in the `ControlUnit` class.

Finally, new register transfer functions can be added to the `ControlUnit` class if needed and others can be changed.

Contacts

If you have questions about the HECTOR 1600 architecture contact:

Dr. Thomas K. Miller
tkm@eos.ncsu.edu

If you have questions about the design or implementation of the BSVC HECTOR 1600 simulator contact:

Bradford W. Mott
bwmott@eos.ncsu.edu