PROJECT DESIGN:

1802 Microprocessor on Altera Cyclone V SoC

Prepared by Jennifer Bi,
Nelson Gomez,
Kundan Guha,
and Justin Wong

Embedded Systems 4840

May 17, 2019

Contents

1 Introduction

1.1 Overview

We implement the CPD1802 Microprocessor on the Altera Cyclone V SoC FPGA using System Verilog. Our implementation has limited functionality, in particular, we do not support interrupt and DMA CPU states. For the parts that we did implement, our implementation remains faithful to the original COSMAC specification. We verified our 1802 implementation with a test suite written in Verilator. We also provide a small software interface on the SoC's ARM Hard Processor System for starting/restting the 1802 as well as loading memory. In particular, this will enable us to load and run programs.

Our original goal was to implement the CPD1861 Video Display Controller in addition to the 1802. This would have enabled us to load an existing implementation of the Chip8 emulator to be able to run Chip8 games on it. As we wrote in our proposal, another goal was to learn about SystemVerilog and FPGA implementation in the process. Because of poor time management and being unfamiliar with FPGA development tools, we were not able to reach the initial goal.

1.2 Background

The CDP1802 COSMAC microprocessor, was introduced and popularized by RCA in the 1970s and 1908s. The system was sold as a single-board personal computer, with two hexadecimal LED displays, 8 toggle switches, and 256 bytes of RAM. RCA also released the CDP1861 Video Display Controller, which integrated with the 1802 to provide display 64x128 pixels using the 1802's DMA and interrupt routines. The 1802 could be used to run a Chip-8 emulator, which would interpret Chip-8 programs and games into 1802 instructions.

1.3 References

1. CDP1802 Microprocessor Specification: *http://www.cosmacelf.com/publications/data-sheets/cdp1802.pdf*
2. CDP1861 Video Display Controller: *http://www.cosmacelf.com/publications/data-sheets/cdp1861.pdf*
3. *http://www.cs.columbia.edu/ sedwards/classes/2016/4840-spring/designs/Chip8.pdf*

4. Complete CHIP-8 Interpreter Listing, which we can load onto our 1802: *http://cosmacelf.com/forumarchive/files/CHIP-8/ELF%20CHIP-8%20Interpreter.pdf*

5. *http://laurencescotford.co.uk/wp-content/uploads/2013/07/RCA1802-Instruction-Set.pdf*

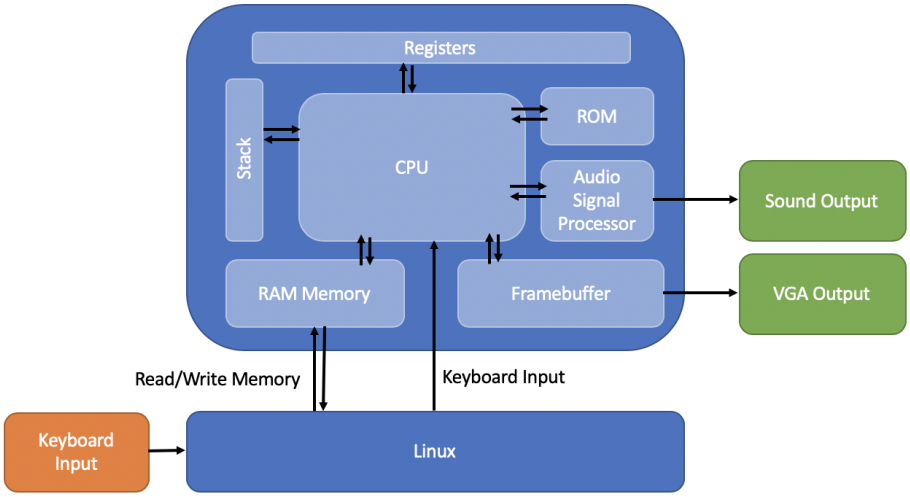
6. *http://bitsavers.trailing-edge.com/components/rca/cosmac/MPM-201A_User_Manual_for_the_CDP1802_COSMAC_Microprocessor_1976.pdf*

2 System Design and Implementation

We use SystemVerilog, a hardware description language to program our FPGA implementation. The CDP1802 including the CPU, registers, and RAM are all implemented on the FPGA; these comprise the hardware component of our project. The software component of our project runs in a Linux environment on the dual-core ARM Cortex-A9 processor on the De1-SoC. The software component is comprised of a custom device driver and user-programs to load memory into the 1802 processor. Finally, our implementation is tested and verified in various different environments on our personal laptops.

2.1 Hardware-Software Interfaces

We allow the user to interact with the system by keyboard input in the Linux environment, via our custom device driver. We use an Avalon Memory-Mapped slave port that reads from and writes to 1802 hardware via the System interconnect fabric. In particular we reserve two memory addresses not relevant to 1802's execution from which we can reset and start/stop the execution of the hardware system.



3 CPU

The 1802 has an array of sixteen 16-bit registers, and an accumulator D to hold to any intermediate computation results. Each CPU instruction uses two machine cycles: fetch and execute. During the fetch cycle, the 4-bit P register selects one of the sixteen registers as the current program counter, and the instruction is read out from memory using the PC/address. The 4-bit X register selects one of the sixteen register as the memory address to the operand for ALU or I/O operations. Registers can also store immediate data.

There are four control modes: `LOAD, RESET, PAUSE, RUN`. We have a RUN switch in software, which allows us to switch between RESET and RUN.

The CPU block diagram from the 1802 specification is shown below. Timing diagrams from the spec were also useful to us, and are described in detail below.

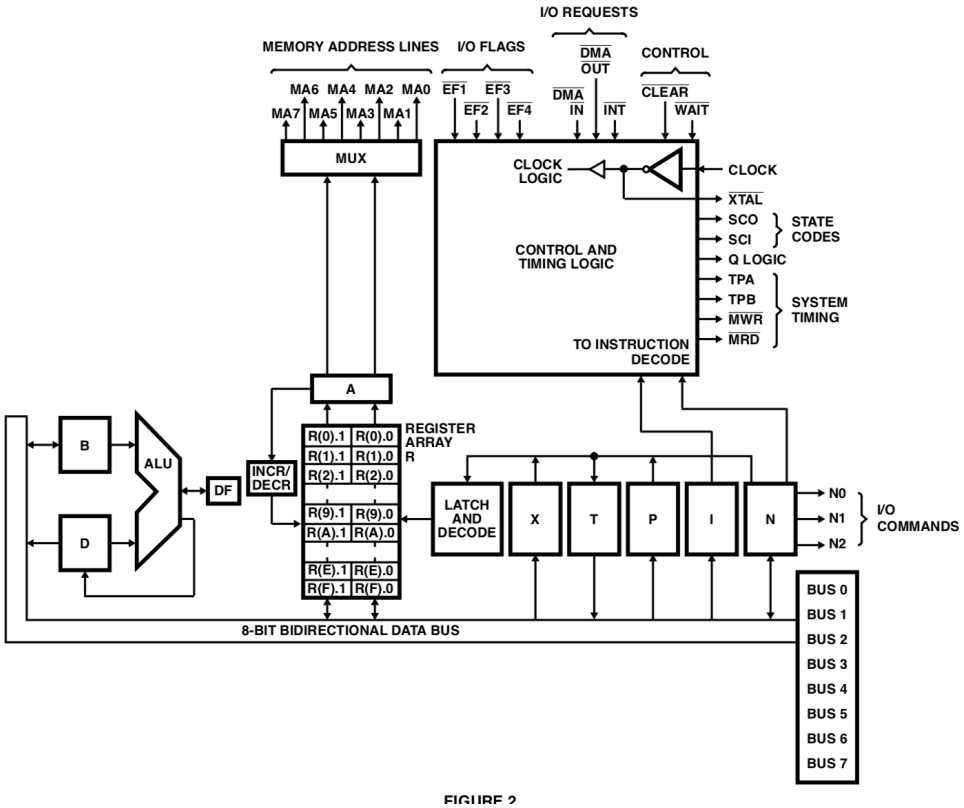
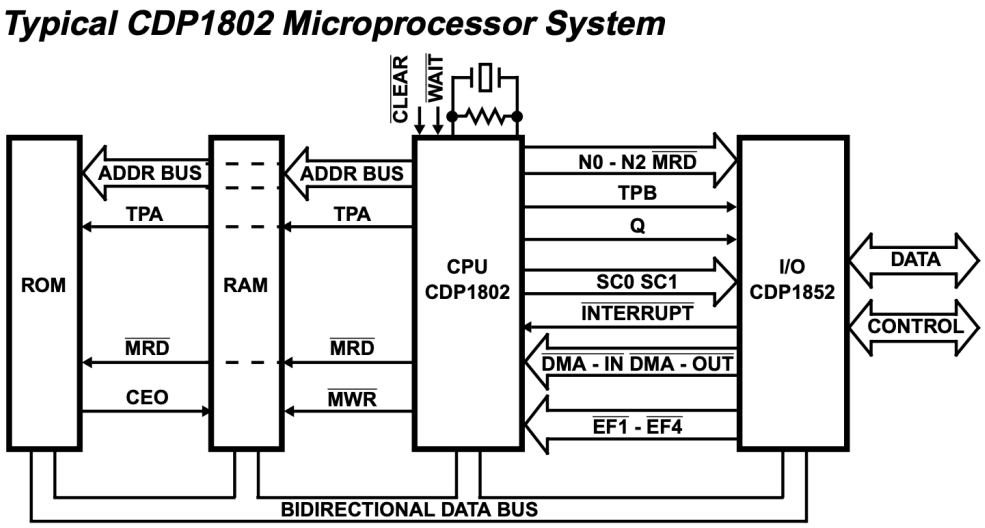


FIGURE 2

4 Memory Overview

We implemented a 4096x8 bit (4KB) RAM. The 1802 specification has 8 memory address lines (MA0-MA7), and uses TPA signal to latch the higher-order byte of a 16-bit memory address. The lower-order byte appears after the TPA ends. In our implementation, we used the

The original 1802 has a byte-wide input/output port, according to the CDP1852 specification. The I/O is used to strobe data in and out from peripheral devices. The 1802 selects by turning the MRD (memory read) line high; the I/O port places the data from the peripheral device onto the data bus. The 1802 also addresses the memory so the data is read from the data bus into memory. Our implementation does not support I/O opcodes according to the original specification, and instead uses a software interface to load memory.



We chose to have 4KB RAM since we originally planned to load the Chip-8 interpreter written for Cosmac Elf, which would present the illusion of 4K of memory. The interpreter will be stored at the first 512 bytes of RAM. Chip-8 programs begin at address 0x20 (512).

5 Peripherals

5.1 Graphics

We learned the details of the CDP1861 Video Display Controller, as well as the DMA/interrupt handshaking performed between the CPU and video controller. However, due to time constraints, we were not able to implement these to completion. The CDP1861 Video Display Controller (also known as Pixie graphics) generates composite vertical and horizontal sync and allows for programmable vertical resolution for up to 64 x 128 segments. The 1861 uses the `INTERRUPT` input and I/O command lines to perform handshaking with the 1802 to set up DMA transfers (discussed in section 6). Chip-8 only had 64x32-pixel monochrome display, so it will not use the full vertical resolution.

The Video Display Controller has a framebuffer which will generate NTSC-rate video output to the VGA monitor.

We implemented a 64x32 resolution VGA display that can interface with the CPU. However, the DMA/interrupt routine was not complete. We used a framebuffer implemented with Megawizard dual-port RAM. The 64X32 display requires 1x2048-bit RAM. We use only 1 bit for on/off, rather than using 8-bits for luminance.

5.2 Sound

A one-bit output from the microprocessor, the Q line, driven by software produces sounds through an attached speaker. Although the original chip's design is to have a single tone, we will look to customize the tone. we were not able to implemement audio.

5.3 Hex Display

The traditional COSMAC has 2 hex displays. We chose instead to use the 6 hex displays on the De1-SoC board as debugging tools. The first two hex displays the contents of the D-register, the middle two display the program counter (P register) and the rightmost two display the address line to ram.

6 ISA

6.1 1802 ISA

Below is a summary of 1802's ISA and we intend to be faithful to the original design. The opcodes are two bytes labeled 'I' and 'N'. For this section, we refer to the 16 2-byte registers as R(X) for X the index of the register. And, we refer to 5 special purpose registers N,P,X,I, and D. Where IP is the opcode and D generally is used for data and X used as an index. R(P) is the current program counter. All instructions are 2 cycles except for CX instructions.

1802 INSTRUCTION MATRIX

		'N'															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
'I'	0	IDL	"LDN" LOAD 'D' FROM REGISTER R(N) – EXCEPT R0														
	1	"INC" INCREMENT REGISTER R(N)															
	2	"DEC" DECREMENT REGISTER R(N)															
	3	BR	BRANCH ON Q , Z , F			BRANCH ON EF1 ~ EF4 = 1				SKP	BRANCH NOT ON Q , Z , F			BRANCH ON EF1 ~ EF4 = 0			
	4	"LDA" LOAD 'D' FROM ADDRESS IN REGISTER R(N) & ADVANCE															
	5	"STR" STORE 'D' INTO ADDRESS POINTED TO BY REGISTER R(N)															
	6	IRX	OUTPUT							-	INPUT						
	7	CONTROL & MEMORY REF.				ARITHMETIC w/ CARRY				CONTROL		RST/SET 'Q'		ARITHMETIC, IMMED. w/ CARRY			
	8	"GLO" GET LOW BYTE OF REGISTER R(N), PUT IN 'D'															
	9	"GHI" GET HIGH BYTE OF REGISTER R(N), PUT IN 'D'															
	A	"PLO" PUT 'D' INTO LOW BYTE OF REGISTER R(N)															
	B	"PHI" PUT 'D' INTO HIGH BYTE OF REGISTER R(N)															
	C	LONG BRANCHES				NOP	LONG SKIPS			LONG BRANCHES				LONG SKIPS			
	D	"SEP" SET REGISTER 'P' FROM LOWER BYTE OF INSTRUCTION (N)															
	E	"SEX" SET REGISTER 'X' FROM LOWER BYTE OF INSTRUCTION (N)															
	F	LOGIC				ARITHMETIC				LOGIC & ARITHMETIC IMMEDIATE				ARITHMETIC IMMED.			

00:IDL Wait for DMA or interrupt
0N: (except N=0) Load Load data from the address stored in register N, R(N) to the D register.
1N: INC Increment value of register N, R(N) .
2N: DEC Decrements value of register N, R(N).
30: BR Set R(P) to memory value stored at R(P).
31: BQ Branch if Q = 1.
32: BZ If register D equal 0 branch to memory value at R(P) otherwise advance R(P) by one.
33: BDF Branch if DF = 1
34: B1 Branch if external flag 1 equal 1
35: B2 Branch if external flag 2 equal 1
36: B3 Branch if external flag 3 equal 1
37: B4 Branch if external flag 4 equal 1
38: SKP Advance R(P) by 1.
39: BNQ Branch if Q = 0
3A: BNZ If register D not equal 0 branch to memory value at R(P) otherwise advance R(P) by one.
3B: BNF Branch if DF not equal 1
3C: BN1 Branch if external flag 1 is 0
3D: BN2 Branch if external flag 2 is 0
3E: BN3 Branch if external flag 3 is 0
3F: BN4 Branch if external flag 4 is 0
4N: LDA Load data from the address stored in R(N) (2 cycles) to the D register and advances the address in register N.
5N: STR Store data in D register to address in register N (2 cycles).
60: IRX For whatever index, x, stored in register X, increment R(X) by 1.
61-67 Write memory value at R(X) to bus and increment R(X).
68 undefined behavior
69-6F Write bus value in R(X) and also in D register.
70 Return from a function. Set X,P to value stored at R(X) advance R(X) and set IE to 1.
71 Return and disable interrupts. IE = 0.
72: LDXA Load to D register address stored in R(X) and advance value of R(X).
73: STXD Store value in D register into memory at address stored in R(X) and decrements value of R(X).
74: ADC Add memory at address in R(P), D register, and carry bit (DF). If there is a carry then carry bit (DF) is set to 1 and R(P) is advanced.
75: SDB Subtract D register value and 1 if DF = 0 from memory stored in address held in register R(X) and store back in D register..
76:SHRC Shift right with carry. Shift right but the most significant bit is now the carry bit (DF).
77: SMB Subtract memory stored in address held in register R(X) and 1 if DF = 0 from D register value and store back in D register.

78: SAV Save register T to address in R(X)
79: MARK Save XP to T register and XP to address in R(2) then set X to P and decrement R(2)
7A: REQ Reset Q to 0
7B: SEQ Set Q to 1
7D: SDBI Subtract D register value and 1 if DF = 0 from memory stored in address held in register R(P) and increment R(P)'s value.
7E:SHLC Shift left with carry. Shift left but the least significant bit is now the carry bit (DF).
7F: SMBI Subtract memory stored in address held in register R(P) and 1 if DF = 0 from D register value and increment R(P)'s value.
8N: GLO Get lower (GLO) byte R(N) and store it in D register.
9N: GHI Get higher (GHI) byte of R(N) and store it in D register.
AN: PLO Write to lower byte of R(N) with value of D register.
BN: PHI
Write to higher byte of R(N) with value of D register.
C0:LBR Long branch (ie set R(P) upper byte to memory at R(P) and R(P) lower byte to memory R(P+1)
C1: LBQ Long barnch if Q = 1
C2: LBZ If D = 0, long branch
C3: LBDF If DF = 1, long branch
C4: NOP continue
C5: LNQ Long skip if Q=0
C6: LSNZ if D not 0, long skip
C7: LSNF Long skip if DF = 0
C8: NLBR Long skip R(P) +=2.
C9: LBNQ Long branch if Q = 0
CA: LBNZ If D not 0, long branch
CB: LBNF If DF = 0, long branch
CC: LSIE Long skip if IE =1 (interrupt enabled)
CD: LSQ Long skip if Q = 1
CE: LSZ Long Skip if D = 0
CF: LSDF Long skip if DF = 1
DN: SEP Set P to N.
EN: SEX Set X to N.
F0 For value in X register, load from address in R(X) to D register.
F1: OR Computes the OR of value of D register and memory in address held in R(X).
F2: AND Computes the AND of value of D register and memory in address held in R(X).
F3: XOR Computes the XOR of value of D register and memory in address held in R(X).
F4: XOR Computes ADD of value of D register and memory in address held in R(X).
F5: SD Computes memory value in address held in R(X) minus D register's value. Afterward DF = 0 if carrying was needed.

F6: SHR Bit shift D register to the right and hold lowest order bit in carry bit (DF).
F7: SM Subtract memory value in address held in R(X) from D and store in D.
F8: LDI For index held in P register, load from address in R(P) into D register. Then increment the value of R(P).
F9: ORI Computes the OR of value of D register and memory held in address held in R(P) and also increments value in P register.
FA: ANI Computes the AND of value of D register and memory held in address held in R(P) and also increments value in P register.
FB: XRI Computes the XOR of value of D register and memory held in address held in R(P) and also increments value in P register.
FC: ADI Computes ADD of value of D register and memory held in address held in R(P) and also increments value in P register.
FD: SDI Computes memory value in address held in R(P) minus D register's value and advance address in R(P). Afterward DF = 0 if carrying was needed.
FE: SHL Shift left and place most significant bit in carry bit (DF).
FF: SMI Subtract memory value in address held in R(P) from D and store in D and increment address in R(P).
INTERRUPT save x and p into register T (XP), set P to 1 and X to 2 and IE to 0
DMA-IN Read from bus to memory addressed by R(0) then advance R(0).
DMA-OUT Write to buss from memory addressed by R(0) and advance R(0).

7 Interrupts and DMA

The interrupt, DMA in, and DMA out inputs are sampled by the CPU. Interrupts can have higher priority over regular I/O by internal flags in the I/O controller. When an interrupt is serviced, the register r(1) is used as the PC. The interrupt action requires one machine cycle. DMA The state transition diagram from the 1802 specification shows the priority of DMA and interrupts.

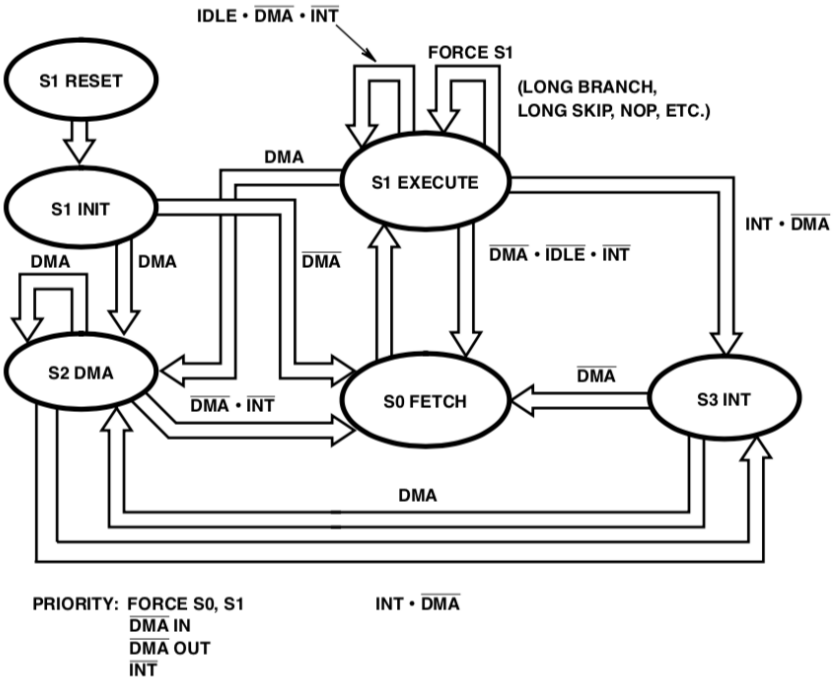


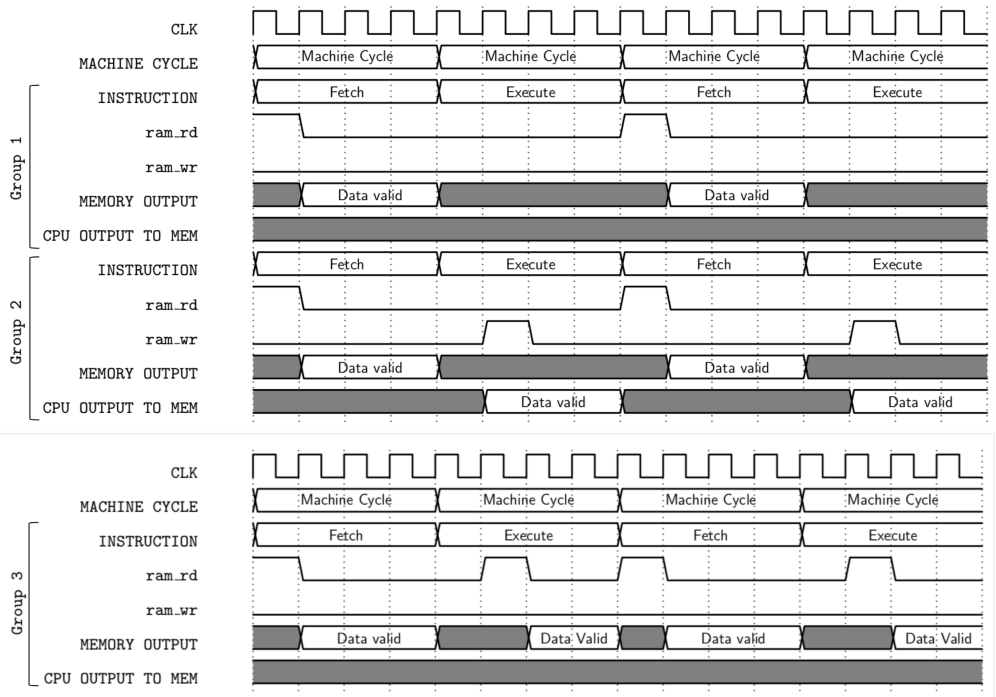
FIGURE 25. STATE TRANSITION DIAGRAM

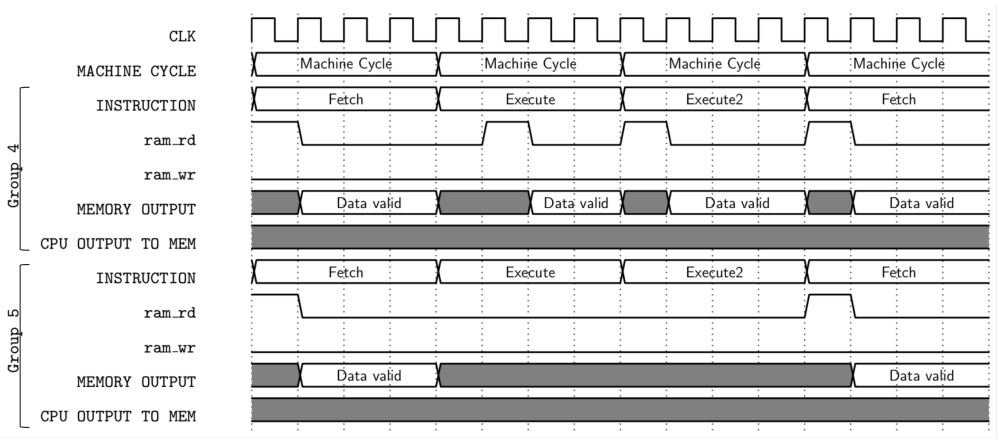
8 Timing diagrams

The following timing diagrams show the detailed fetch-execute cycle behavior of various opcodes. The opcodes can be group by behavior, as shown in the table:

Group	Behavior
1	Read/Non-memory
2	Read/Write
3	Read/Read
4	Read/Read/Read
5	Read/Non-memory/Non-memory

Unlike the 1802 which as 8 clock cycles per machine cycle, we use 4 clock cycles per machine cycle. In the fetch cycle, the clock cycle 0 updates the address to be read, clock cycle 1 reads valid data from RAM, clock cycles 2,3 are stalls. In the execute cycle, clock cycle 0 is used for branch instructions to check branch conditions, clock cycle 1 either updates the address to read, the register to read into, and turns the read-enable high, or updates the address to be written, the data to write, and turns write-enable high. In clock cycle 1, any register manipulation also occurs. In clock cycle 2, ALU results become available to ALU opcodes. Below are timing diagrams for each group.




CLK
MACHINE CYCLE
Machine Cycle
Machine Cycle
Machine Cycle
Machine Cycle
Group 4
INSTRUCTION
Fetch
Execute
Execute2
Fetch
ram_rd
ram_wr
MEMORY OUTPUT
Data valid
Data valid
Data valid
Data valid
CPU OUTPUT TO MEM
Group 5
INSTRUCTION
Fetch
Execute
Execute2
Fetch
ram_rd
ram_wr
MEMORY OUTPUT
Data valid
Data valid
CPU OUTPUT TO MEM

9 Testing

9.1 Verilator

For software based high level testing of our designs we used Verilator. Verilator is an open source project to take (System) Verilog projects and synthesize them into C++ or System C libraries capable of a high level simulation of the project. While by no means a substitute for actually testing on the hardware, Verilator allowed us to test small changes faster and setup extensive unit tests of individual features. The Verilator programs are C++ programs that instantiate the provided classes of the Verilator-generated project libraries, designed as a state machine simulation is as simple as toggling the class's "clock" variable from 0 to 1 and back inside a loop. For every step it is possible to access the states of any of the components of our design, giving us an in-depth picture of the execution of our design.

We discovered some limitations of Verilator at the lower-levels of SystemVerilog synthesis. Namely, when we started full testing on hardware we ran into issues where code that may have worked in Verilator will not exactly work on the hardware, which limited the long-term usefulness of our Verilator tests.

9.2 TinyElf

We can compare the processor behavior (register state, PC, program output) with an 1802 Instruction-level simulator, TinyElf (link). TinyElf works well for this purpose as it allows us to view the memory state, registers, breakpoints, traces, hex display, and run modes.

10 Code Listings

10.1 Hardware files

```
module processor(
               input logic          clk,
               input logic [11:0]   a,
               input logic          reset,
               input logic [7:0]    din,
               input logic          we,
               input                chipselect,
               output logic [7:0]   VGA_R, VGA_G, VGA_B,
               output logic         VGA_CLK, VGA_HS, VGA_VS,
                                    VGA_BLANK_n,
               output logic         VGA_SYNC_n,
               output logic [7:0]   dout,
               output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
               output logic [9:0] LEDR);

    logic         ram_we;
    logic [7:0]   bus_from_cpu;
    logic [7:0]   bus_to_cpu;
    logic [11:0] address_from_cpu;
    logic         sw_we;
    logic  [7:0]     dout_;
    logic state;

    assign sw_we = chipselect && we;

    cpu  c (.clk(clk),
        .reset(reset),
        .we(we),
        .addr(a),
        .bus_from_ram(bus_to_cpu),
        .bus_to_ram(bus_from_cpu),
        .ram_we(ram_we),
        .address_to_ram(address_from_cpu),
        .state_out(state),
        .*);

`ifdef VERILATOR
   test_ram m (
        .address_a(a),
        .address_b(address_from_cpu),
        .clock(clk),
        .data_a(din),
        .data_b(bus_from_cpu),
        .wren_a(sw_we),
        .wren_b(ram_we),
        .q_a(dout_),
        .q_b(bus_to_cpu)
    );
`else
```

```
50      mem m ( // _a is for software, _b is for cpu
51          .address_a(a),
52          .address_b(address_from_cpu),
53          .clock(clk),
54          .data_a(din),
55          .data_b(bus_from_cpu),
56          .wren_a(sw_we),
57          .wren_b(ram_we),
58          .q_a(dout_),
59          .q_b(bus_to_cpu)
60      );
61  `endif
62
63      report r ( we,state,a,dout_,dout);
64
65  endmodule
66
67
68  // Less complex and non-Altera-specific memory module used only for testing.
69  // Adapted from http://verilogcodes.blogspot.com/2017/11/verilog-code-for-dual-port-ram-with.html
70  // because I don't have the patience to code this up myself at this point :^)
71
72  module test_ram
73      (   input clock,            //clock
74          input [7:0] data_a,     //Input data to port 0
75          input [7:0] data_b,     //Input data to port 1
76          input [11:0] address_a,  //address for port 0
77          input [11:0] address_b,  //address for port 1
78          input wren_a,           //enable port 0.
79          input wren_b,           //enable port 1
80          output [7:0] q_a,       //output data from port 0.
81          output [7:0] q_b        //output data from port 1
82      );
83
84  //memory declaration.
85  reg [7:0] mem[4095:0];
86
87  //writing to the RAM
88  always_ff @(posedge clock)
89  begin
90      if(wren_a)    //check enable signal and if write enable is ON
91          mem[address_a] <= data_a;
92
93      // No guarantees if both signals are high at the same time
94      // with the same address.
95      if(wren_b)
96          mem[address_b] <= data_b;
97  end
98
99  //always reading from the ram, irrespective of clock.
100 assign q_a = mem[address_a];
101 assign q_b = mem[address_b];
102
103 endmodule
104
105
106 module hex7seg(input logic [3:0] a,
107            output logic [6:0] y);
108
109    always_comb
110       case (a)
111          4'h0:    y = 7'b100000_0;
```

```
          4'h1:    y = 7'b111100_1;
          4'h2:    y = 7'b010010_0;
          4'h3:    y = 7'b011000_0;
          4'h4:    y = 7'b001100_1;
          4'h5:    y = 7'b001001_0;
          4'h6:    y = 7'b000001_0;
          4'h7:    y = 7'b111100_0;
          4'h8:    y = 7'b000000_0;
          4'h9:    y = 7'b001100_0;
          4'hA:    y = 7'b000100_0;
          4'hB:    y = 7'b000001_1;
          4'hC:    y = 7'b100011_0;
          4'hD:    y = 7'b010000_1;
          4'hE:    y = 7'b000011_0;
          4'hF:    y = 7'b000111_0;
          default: y = 7'b111111_1;
      endcase

endmodule

module report (input logic we,
               input logic state,
               input logic [11:0] a,
               input logic [7:0]  dout_,
               output logic [7:0] dout);
always_comb begin
  dout[6:0] = dout_[6:0];
  dout[7] = (dout_[7] )//| (state & a[4] & ~a[3] & ~a[2] & ~a[1] & a[0]) ;
  end
endmodule

module alu(input logic        clk,
           input logic        reset,
           input logic  [7:0] D_register,
           input logic  [3:0] I_register,
           input logic  [3:0] N_register,
           input logic        DF,
           input logic  [7:0] operand,
           output logic [7:0] result,
           output logic       carry,
           output logic       out_rdy
);
    logic [8:0] tmp;
    logic valid;

    always_comb begin
        valid = 1;

        unique case ({I_register, N_register})
            // ADC(I): add with carry
            8'h74, 8'h7c: {carry, result} = {1'b0, D_register} + {1'b0, operand} + {8'b0, DF};

            // SDB/SDI: subtract with borrow
            8'h75, 8'h7d: {carry, result} = ({1'b0, operand} + {1'b0, ~D_register} + {8'b0, DF} + 1)

            // SMB(I): subtract memory with borrow
            8'h77, 8'h7f: {carry, result} = ({1'b0, D_register} + {1'b0, ~operand} + {8'b0, DF} + 1)

            // OR(I): logical or
            8'hf1, 8'hf9: {carry, result} = {DF, operand | D_register};

            // AND/ANI: logical and
```

```
174             8'hf2, 8'hfa: {carry, result} = {DF, operand & D_register};
175
176             // XOR/XRI: exclusive or
177             8'hf3, 8'hfb: {carry, result} = {DF, operand ^ D_register};
178
179             // ADD/ADI: add without carry
180             8'hf4, 8'hfc: {carry, result} = {1'b0, operand} + {1'b0, D_register};
181
182             // SD(I): subtract without borrow
183             8'hf5, 8'hfd: {carry, result} = ({1'b0, operand} + {1'b0, ~D_register} + 1);
184
185             // SM(I): subtract memory without borrow
186             8'hf7, 8'hff: {carry, result} = ({1'b0, D_register} + {1'b0, ~operand} + 1);
187
188             default: {carry, result, valid} = {1'b0, 8'b0, 1'b0};
189         endcase
190
191         out_rdy = valid;
192     end
193 endmodule
194
195
196 module cpu(input logic         clk,
197            input logic         reset,
198            input logic         we,
199            input logic [11:0]   addr,
200            input logic [7:0]   bus_from_ram,
201            output logic [7:0]  bus_to_ram,
202            output logic        ram_we,
203            output logic [11:0] address_to_ram,
204            output logic        state_out,
205            output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5,
206      output logic [9:0] LEDR
207 );
208
209     logic [15:0] R_registers[15:0];
210     logic [3:0] N_register;
211     logic [3:0] P_register;
212     logic [3:0] X_register;
213     logic [3:0] I_register;
214     logic [7:0] D_register;
215     logic [7:0] T_register;
216     logic B_registers[3:0];
217     logic Q;
218     logic DF; //carry bit
219     logic IE;
220     logic idle;
221   logic ram_rd; // not a real read-en to memory, just for cpu
222
223     logic [7:0] alu_operand;
224     logic [7:0] alu_result;
225     logic       alu_carry;
226     logic       alu_out_rdy;
227
228     alu a (.operand(alu_operand), .result(alu_result), .carry(alu_carry), .out_rdy(alu_out_rdy), .*)
229
230     enum logic [2:0] {S_FETCH, S_EXECUTE, S_EXECUTE2, S_DMA, S_INTERRUPT} state;
231
232     // n.b: the CLEAR and WAIT signals are active low and need to be inverted
233     // Also, Quartus won't synthesize this if it's an enum because we assign a non-enum packed array
234     // to it later, so the constants are defined separately.
235     logic [1:0] mode = 2'h1;
```

```
236
237       hex7seg h0( .a(address_to_ram[3:0]), .y(HEX0) ),
238               h1( .a(address_to_ram[7:4]), .y(HEX1) ),
239               h2( .a(R_registers[0][3:0]), .y(HEX2) ),
240               h3( .a(R_registers[0][7:4]), .y(HEX3) ),
241               h4( .a(D_register[3:0]), .y(HEX4) ),
242               h5( .a(D_register[7:4]), .y(HEX5) );
243
244       localparam logic [1:0] M_RUN = 2'h3;
245       localparam logic [1:0] M_PAUSE = 2'h2;
246       localparam logic [1:0] M_RESET = 2'h1;
247       localparam logic [1:0] M_LOAD = 2'h0;
248
249       localparam logic [4:0] RAM_D_REG   = 5'b10000;
250       localparam logic [4:0] RAM_T_REG   = 5'b10001;
251       localparam logic [4:0] RAM_ALU_OP  = 5'b10010;
252       localparam logic [4:0] RAM_XP_REG  = 5'b10011;
253     logic [4:0] R_reg_sel = RAM_D_REG;
254
255       logic should_branch;
256       logic should_branch_comb;
257       logic should_skip;
258
259       always_comb begin
260     LEDR[9:8] = mode;
261     LEDR[7] = DF;
262     LEDR[6] = Q;
263     LEDR[5] = IE;
264     LEDR[4] = idle;
265     LEDR[3:0] = 4'b0001;
266       end
267
268       logic [1:0] clock_counter;
269
270       // Most of the S_EXECUTE (S1) logic is here, it's just cleaner to
271       // have it as a task rather than as a deeply nested pile of code.
272       task execute;
273           // 'unique' acts as a hint to the synthesizer. Some more reading here:
274           // https://www.verilogpro.com/systemverilog-unique-priority/
275           unique casez (I_register)
276               4'h0:
277                   begin
278                       // IDL (FIXME: CPU ignores idle signal)
279                       if (N_register == 4'h0) idle <= 1;
280                       // LDN - load 'D' from R(N)
281                       else
282                       begin
283                        address_to_ram <= R_registers[N_register][11:0];
284                        ram_rd <= 1;
285                        R_reg_sel <= RAM_D_REG;
286                       end
287                   end
288
289               // INC/DEC R(N)
290               4'h1: R_registers[N_register] <= R_registers[N_register] + 1;
291               4'h2: R_registers[N_register] <= R_registers[N_register] - 1;
292
293               // Branch instructions
294               4'h3:
295                   begin
296             if (should_branch)
297             begin
```

```
298             address_to_ram <= R_registers[P_register][11:0];
299             ram_rd <= 1;
300             R_reg_sel <= {1'b0, P_register};
301           end
302
303                     // If not branching, move program counter forward a byte;
304
305                     // the program counter gets advanced during FETCH to skip
306                     // the jump entirely.
307                     //
308                     // If we ARE branching, this will get overwritten when
309                     // the ram_task is processed.
310                     R_registers[P_register] <= R_registers[P_register] + 1;
311                 end
312
313             // Load Advance (LDA)
314             4'h4:
315                 begin
316           address_to_ram <= R_registers[N_register][11:0];
317           ram_rd <= 1;
318           R_reg_sel <= RAM_D_REG;
319                     R_registers[N_register] <= R_registers[N_register] + 1;
320                 end
321
322             // Store via N (STR)
323             4'h5:
324         begin
325           address_to_ram <= R_registers[N_register][11:0];
326           bus_to_ram <= D_register;
327           ram_we <= 1;
328         end
329             4'h6:
330                 begin
331                     unique casez (N_register)
332                         // IRX: increment R(X) by 1
333                         4'h0: R_registers[X_register] <= R_registers[X_register] + 1;
334
335                         // 61-67: write R(X) to bus and increment R(X)
336                         // FIXME: we don't have a bus, performs a NOP instead
337                         4'b0001,
338                         4'b001?,
339                         4'b01??,
340
341                         // 69-6F: write bus value to R(X) and D registers
342                         // (opcode 68 is undefined, but the case logic is nicer if we just
343                         // make it another BUS->R(X) instruction)
344                         // FIXME: we don't have a bus, performs a NOP instead
345                         4'b1???: ;
346                     endcase
347                 end
348
349             4'h7,
350             4'hf:
351                 begin
352                     unique case ({I_register[3], N_register})
353                         // RET: return and enable interrupts
354                         // DIS: return and disable interrupts
355                         5'h00,
356                         5'h01:
357                             begin
358                 address_to_ram <= R_registers[X_register][11:0];
359                 ram_rd <= 1;
```

```
360                     R_reg_sel <= RAM_XP_REG;
361                                     IE <= ~(N_register[0]);
362                                     // Incrementing R(X) is done on a later clock cycle.
363                                 end
364
365                             // LDXA: load via X and advance
366                             5'h02:
367                                 begin
368                     address_to_ram <= R_registers[X_register][11:0];
369                     ram_rd <= 1;
370                     R_reg_sel <= RAM_D_REG;
371                                     R_registers[X_register] <= R_registers[X_register] + 1;
372                                 end
373
374                             // STXD: store via X and decrement
375                             5'h03:
376                                 begin
377                     address_to_ram <= R_registers[X_register][11:0];
378                     bus_to_ram <= D_register;
379                     ram_we <= 1;
380                                     R_registers[X_register] <=  R_registers[X_register] - 1;
381                                 end
382
383                             // SAV: save
384                             5'h08:
385                 begin
386                     address_to_ram <= R_registers[X_register][11:0];
387                     ram_rd <= 1;
388                     R_reg_sel <= RAM_T_REG;
389                 end
390                             // MARK: push X,P to stack
391                             5'h09:
392                                 begin
393                                     T_register <= {X_register, P_register};
394                     address_to_ram <= R_registers[2][11:0];
395                     bus_to_ram <= {X_register, P_register};
396                     ram_we <= 1;
397                                     P_register <= X_register;
398                                     R_registers[2] <= R_registers[2] - 1;
399                                 end
400
401                             // REQ: Q = 0
402                             5'h0a: Q <= 0;
403
404                             // SEQ: Q = 1
405                             5'h0b: Q <= 1;
406
407                             // LDX: load via X
408                             5'h10:
409                 begin
410                     address_to_ram <= R_registers[X_register][11:0];
411                     ram_rd <= 1;
412                     R_reg_sel <= RAM_D_REG;
413                 end
414
415                             // LDI: Load immediate
416                             5'h18:
417                                 begin
418                     address_to_ram <= R_registers[P_register][11:0];
419                     ram_rd <= 1;
420                     R_reg_sel <= RAM_D_REG;
421                                     R_registers[P_register] <= R_registers[P_register] + 1;
```

```
422                         end
423
424                     ////////////////// Logic/Arithmetic operations \\\\\\\\\\\\\\\\\
425
426                     // A lot of arithmetic operations need to read from memory before
427                     // calculating the result. Instructions that need to read via the
428                     // same register are all executed with the same ram_task; the
429                     // actual arithmetic is done on the third clock cycle of the
430                     // S_EXECUTE state.
431                     //
432                     // The actual logic for arithmetic and logical operations isn't
433                     // here. For that, check the `alu` module.
434                     //
435                     // Read via R(X):
436                     // 0x74 | ADC | add with carry
437                     // 0x75 | SDB | subtract with borrow
438                     // 0x77 | SMB | subtract memory with borrow
439                     // 0xf1 | OR  | logical or
440                     // 0xf2 | AND | logical and
441                     // 0xf3 | XOR | exclusive or
442                     // 0xf4 | ADD | add without carry
443                     // 0xf5 | SD  | subtract without borrow
444                     // 0xf7 | SM  | subtract memory without borrow
445                     5'h04,
446                     5'h05,
447                     5'h07,
448                     5'h11,
449                     5'h12,
450                     5'h13,
451                     5'h14,
452                     5'h15,
453                     5'h17:
454                 begin
455                 address_to_ram <= R_registers[X_register][11:0];
456                 ram_rd <= 1;
457                 R_reg_sel <= RAM_ALU_OP;
458                 end
459
460                     // Read via R(P):
461                     // 0x7c | ADCI | add immediate with carry
462                     // 0x7d | SDI  | subtract immediate with borrow
463                     // 0x7f | SMBI | subtract memory immediate with borrow
464                     // 0xf9 | ORI  | logical or w/ immediate
465                     // 0xfa | ANI  | logical and w/ immediate
466                     // 0xfb | XRI  | exclusive or w/ immediate
467                     // 0xfc | ADI  | add immediate without carry
468                     // 0xfd | SDI  | subtract immediate without borrow
469                     // 0xff | SMI  | subtract memory immediate without borrow
470                     5'h0c,
471                     5'h0d,
472                     5'h0f,
473                     5'h19,
474                     5'h1a,
475                     5'h1b,
476                     5'h1c,
477                     5'h1d,
478                     5'h1f:
479                         begin
480                   address_to_ram <= R_registers[P_register][11:0];
481                   ram_rd <= 1;
482                   R_reg_sel <= RAM_ALU_OP;
483                               R_registers[P_register] <= R_registers[P_register] + 1;
```

```
484                                 P_register <= 0;
485                             end
486
487                         // SHRC: ring shift right (0x76)
488                         // SHR: Shift right without carry (0xf6)
489                         5'h06,
490                         5'h16:
491                             begin
492                                 DF <= D_register[0];
493                                 D_register <= {D_register[0] & ~I_register[3], D_register[7:1]};
494                             end
495
496                         // SHLC: ring shift left (0x7e)
497                         // SHL: Shift left without carry (0xfe)
498                         5'h0e,
499                         5'h1e:
500                             begin
501                                 DF <= D_register[7];
502                                 D_register <= {D_register[6:0], D_register[7] & ~I_register[3]};
503                             end
504                     endcase
505                 end
506
507             // GET LOW
508             4'h8:   D_register <= R_registers[N_register][7:0];
509             // GET HIGH
510             4'h9:   D_register <= R_registers[N_register][15:8];
511             // PUT LOW
512             4'ha:   R_registers[N_register][7:0] <= D_register;
513             // PUT HIGH
514             4'hb:   R_registers[N_register][15:8] <= D_register;
515
516             // SEP
517             4'hd:   P_register <= N_register;
518             // SEX
519             4'he:   X_register <= N_register;
520
521             ////////////////////// Long branches \\\\\\\\\\\\\\\\\\\\
522             4'hc:
523                 // Kind of cheating by using the ALU operand as a scratch register
524             begin
525                 if (should_branch_comb)
526                 begin
527                     R_registers[P_register] <= R_registers[P_register] + (should_skip ? 2 :
    0);
528                     ram_rd <=1;
529                     R_reg_sel <= RAM_ALU_OP;
530                 end
531         end
532         endcase
533     endtask
534
535     task branch_execute;
536     if (I_register == 4'h3)
537     begin
538         // We don't know what the address to jump to is yet.
539         // Read the byte referenced by the program counter
540         // into the program counter.
541         unique case (N_register)
542             4'h0: should_branch <= 1'b1;
543             4'h1: should_branch <= (Q == 1);
544             4'h2: should_branch <= (D_register == 8'h0);
```

```
545             4'h3: should_branch <= (DF == 1);
546             4'h4: should_branch <= (B_registers[0] == 1);
547             4'h5: should_branch <= (B_registers[1] == 1);
548             4'h6: should_branch <= (B_registers[2] == 1);
549             4'h7: should_branch <= (B_registers[3] == 1);
550             4'h8: should_branch <= 1'b0;
551             4'h9: should_branch <= (Q == 0);
552             4'ha: should_branch <= (D_register != 0);
553             4'hb: should_branch <= (DF != 1);
554             4'hc: should_branch <= (!B_registers[0]);
555             4'hd: should_branch <= (!B_registers[1]);
556             4'he: should_branch <= (!B_registers[2]);
557             4'hf: should_branch <= (!B_registers[3]);
558         endcase
559     end
560     else if (I_register == 4'hc)
561     ////////////////////// Long branches \\\\\\\\\\\\\\\\\\\\
562     begin
563         unique case (N_register[2:0])
564             3'h0: should_branch <= 1;
565             3'h1: should_branch <= Q;
566             3'h2: should_branch <= (D_register == 8'h0);
567             3'h3: should_branch <= DF;
568             3'h4: should_branch <= ~IE;
569             3'h5: should_branch <= ~Q;
570             3'h6: should_branch <= (D_register != 8'h0);
571             3'h7: should_branch <= ~DF;
572         endcase
573         // Are these skip instructions?
574         if (N_register[2])
575             begin
576                 should_skip <= should_branch;
577                 should_branch <= 0;
578             end
579         else
580         should_skip <= ~should_branch;
581     end
582     endtask
583 
584     always_comb
585     begin
586         // C8-CF are the same operations as C0-C7 with inverted conditions,
587         // with the exception of C4, which is a no-op.
588         should_branch_comb = (should_branch ^ N_register[3]) & (N_register != 4'h4);
589     end
590 
591 
592     always_ff @(posedge clk)
593     begin
594         if (we && (addr== 12'h010) && mode == M_LOAD ) begin
595            mode <= M_RUN;
596         end
597         else if (we && (addr==12'h010) && mode != M_LOAD) begin
598            mode <= M_LOAD;
599         end
600         else if (we && (addr== 12'h014) ) begin
601            mode <= M_RESET;
602         end
603         if (reset || mode == M_RESET)
604         begin
605             I_register <= 0;
606             N_register <= 0;
```

```
607             Q <= 0;
608             IE <= 1;
609             bus_to_ram <= 0;
610             ram_we <= 0;
611             ram_rd <= 0;
612             address_to_ram <= 12'h0;
613             state <= S_FETCH;
614
615             clock_counter <= 0;
616
617             // FIXME: Datasheet p. 3-21 says that registers X, P, and R(0)
618             // are initialized on the first machine cycle _after_ RESET is
619             // terminated; do we need to adhere strictly to that or is it
620             // fine to do this all in one go?
621             X_register <= 0;
622             P_register <= 0;
623             R_registers <= '{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
624             mode <= M_LOAD;
625             state_out <=0;
626         idle <= 0;
627           end
628           else if (mode == M_LOAD)
629           begin
630
631               state_out <= 0;
632           end
633           else if (mode == M_RUN)
634           begin
635               state_out <= 1;
636               unique case (state)
637                   S_FETCH:
638                       begin
639                           if (clock_counter == 0)
640                           begin
641                               ram_we <= 0;
642                               address_to_ram <= R_registers[P_register][11:0];
643                           end
644                           // Stall for one cycle to match the EXECUTE machine cycle
645                           else if (clock_counter == 1)
646                           begin
647                               I_register <= bus_from_ram[7:4];
648                               N_register <= bus_from_ram[3:0];
649                               R_registers[P_register] <= R_registers[P_register] + 1;
650                           end
651         else if (clock_counter == 3)
652                               state <= S_EXECUTE;
653                       end
654
655                   S_EXECUTE:
656                       begin
657         if (clock_counter == 0)
658                             branch_execute();
659         else if (clock_counter == 1)
660                             execute();
661         else if (clock_counter == 2)
662                           begin
663                               if (ram_rd) // clock counter == 2 and ram_rd
664                               begin
665                                   unique case (R_reg_sel)
666                                       RAM_D_REG:  D_register <= bus_from_ram;
667                                       RAM_T_REG:  T_register <= bus_from_ram;
668                                       RAM_ALU_OP: alu_operand <= bus_from_ram;
```

```
669                                     RAM_XP_REG: {X_register, P_register} <= bus_from_ram;
670                                     default:    R_registers[R_reg_sel[3:0]] <= {R_registers[R_reg_se
671                                 endcase
672                             end
673                             ram_rd <= 0;
674                             ram_we <= 0;
675                         end
676                         else if (clock_counter == 3)
677                         begin
678                             // Await the results of any arithmetic operations.
679                             // If we have no ALU results, discard the output.
680                             if (alu_out_rdy)
681                             begin
682                                 D_register <= alu_result;
683                                 DF <= alu_carry;
684                             end
685
686                             // This is also where we handle the RET and DIS
687                             // instructions' increment on R(X).
688                             if (I_register == 4'h7 && (N_register & 4'b1110) == 4'b0000)
689                                 R_registers[X_register] <= R_registers[X_register] + 1;
690
691                             // User manual says everything, including the NOP,
692                             // goes to the EXEC2 state.
693                             if (I_register == 4'hc) begin
694                              state <= S_EXECUTE2;
695                             end
696                             else if (~idle) state <= S_FETCH;
697                         end
698                     end
699
700                 S_EXECUTE2: //TODO dead cycles
701                     begin
702                         if (clock_counter == 0)
703                         begin
704                             address_to_ram <= R_registers[P_register][11:0];
705         ram_rd <= 1;
706                         end
707                         else if (clock_counter == 2)
708                         begin
709                             if (should_branch) R_registers[P_register] <= {bus_from_ram, alu_operand
710         ram_rd <= 0;
711                         end
712                         else if (clock_counter == 3)
713                             state <= S_FETCH;
714                     end
715
716                 // TODO: DMA
717                 S_DMA:
718                     begin
719                     end
720
721                 // TODO: interrupt
722                 S_INTERRUPT:
723                     begin
724                     end
725           endcase
726
727           if (clock_counter == 3) clock_counter <= 0;
728           else clock_counter <= clock_counter + 1;
729       end
730   end
```

```
731 endmodule
```

processor.sv

```
1  // ====================================================================
2  // Copyright (c) 2013 by Terasic Technologies Inc.
3  // ====================================================================
4  //
5  // Modified 2019 by Stephen A. Edwards
6  //
7  // Permission:
8  //
9  //   Terasic grants permission to use and modify this code for use in
10 //   synthesis for all Terasic Development Boards and Altera
11 //   Development Kits made by Terasic.  Other use of this code,
12 //   including the selling ,duplication, or modification of any
13 //   portion is strictly prohibited.
14 //
15 // Disclaimer:
16 //
17 //   This VHDL/Verilog or C/C++ source code is intended as a design
18 //   reference which illustrates how these types of functions can be
19 //   implemented.  It is the user's responsibility to verify their
20 //   design for consistency and functionality through the use of
21 //   formal verification methods.  Terasic provides no warranty
22 //   regarding the use or functionality of this code.
23 //
24 // ====================================================================
25 //
26 //  Terasic Technologies Inc
27
28 //  9F., No.176, Sec.2, Gongdao 5th Rd, East Dist, Hsinchu City, 30070. Taiwan
29 //
30 //
31 //                     web: http://www.terasic.com/
32 //                     email: support@terasic.com
33 module soc_system_top(
34
35  ///////// ADC /////////
36  inout          ADC_CS_N,
37  output         ADC_DIN,
38  input          ADC_DOUT,
39  output         ADC_SCLK,
40
41  ///////// AUD /////////
42  input          AUD_ADCDAT,
43  inout          AUD_ADCLRCK,
44  inout          AUD_BCLK,
45  output         AUD_DACDAT,
46  inout          AUD_DACLRCK,
47  output         AUD_XCK,
48
49  ///////// CLOCK2 /////////
50  input          CLOCK2_50,
51
52  ///////// CLOCK3 /////////
53  input          CLOCK3_50,
54
55  ///////// CLOCK4 /////////
56  input          CLOCK4_50,
57
58  ///////// CLOCK /////////
59  input          CLOCK_50,
```

```
60
61  ///////// DRAM /////////
62  output [12:0] DRAM_ADDR,
63  output [1:0]  DRAM_BA,
64  output        DRAM_CAS_N,
65  output        DRAM_CKE,
66  output        DRAM_CLK,
67  output        DRAM_CS_N,
68  inout [15:0]  DRAM_DQ,
69  output        DRAM_LDQM,
70  output        DRAM_RAS_N,
71  output        DRAM_UDQM,
72  output        DRAM_WE_N,
73
74  ///////// FAN /////////
75  output        FAN_CTRL,
76
77  ///////// FPGA /////////
78  output        FPGA_I2C_SCLK,
79  inout         FPGA_I2C_SDAT,
80
81  ///////// GPIO /////////
82  inout [35:0]  GPIO_0,
83  inout [35:0]  GPIO_1,
84
85  ///////// HEX0 /////////
86  output [6:0]  HEX0,
87
88  ///////// HEX1 /////////
89  output [6:0]  HEX1,
90
91  ///////// HEX2 /////////
92  output [6:0]  HEX2,
93
94  ///////// HEX3 /////////
95  output [6:0]  HEX3,
96
97  ///////// HEX4 /////////
98  output [6:0]  HEX4,
99
100 ///////// HEX5 /////////
101 output [6:0]  HEX5,
102
103 ///////// HPS /////////
104 inout         HPS_CONV_USB_N,
105 output [14:0] HPS_DDR3_ADDR,
106 output [2:0]  HPS_DDR3_BA,
107 output        HPS_DDR3_CAS_N,
108 output        HPS_DDR3_CKE,
109 output        HPS_DDR3_CK_N,
110 output        HPS_DDR3_CK_P,
111 output        HPS_DDR3_CS_N,
112 output [3:0]  HPS_DDR3_DM,
113 inout [31:0]  HPS_DDR3_DQ,
114 inout [3:0]   HPS_DDR3_DQS_N,
115 inout [3:0]   HPS_DDR3_DQS_P,
116 output        HPS_DDR3_ODT,
117 output        HPS_DDR3_RAS_N,
118 output        HPS_DDR3_RESET_N,
119 input         HPS_DDR3_RZQ,
120 output        HPS_DDR3_WE_N,
121 output        HPS_ENET_GTX_CLK,
```

```
122   inout          HPS_ENET_INT_N,
123   output         HPS_ENET_MDC,
124   inout          HPS_ENET_MDIO,
125   input          HPS_ENET_RX_CLK,
126   input [3:0]    HPS_ENET_RX_DATA,
127   input          HPS_ENET_RX_DV,
128   output [3:0]   HPS_ENET_TX_DATA,
129   output         HPS_ENET_TX_EN,
130   inout          HPS_GSENSOR_INT,
131   inout          HPS_I2C1_SCLK,
132   inout          HPS_I2C1_SDAT,
133   inout          HPS_I2C2_SCLK,
134   inout          HPS_I2C2_SDAT,
135   inout          HPS_I2C_CONTROL,
136   inout          HPS_KEY,
137   inout          HPS_LED,
138   inout          HPS_LTC_GPIO,
139   output         HPS_SD_CLK,
140   inout          HPS_SD_CMD,
141   inout [3:0]    HPS_SD_DATA,
142   output         HPS_SPIM_CLK,
143   input          HPS_SPIM_MISO,
144   output         HPS_SPIM_MOSI,
145   inout          HPS_SPIM_SS,
146   input          HPS_UART_RX,
147   output         HPS_UART_TX,
148   input          HPS_USB_CLKOUT,
149   inout [7:0]    HPS_USB_DATA,
150   input          HPS_USB_DIR,
151   input          HPS_USB_NXT,
152   output         HPS_USB_STP,
153
154   ///////// IRDA /////////
155   input          IRDA_RXD,
156   output         IRDA_TXD,
157
158   ///////// KEY /////////
159   input [3:0]    KEY,
160
161   ///////// LEDR /////////
162   output [9:0]   LEDR,
163
164   ///////// PS2 /////////
165   inout          PS2_CLK,
166   inout          PS2_CLK2,
167   inout          PS2_DAT,
168   inout          PS2_DAT2,
169
170   ///////// SW /////////
171   input [9:0]    SW,
172
173   ///////// TD /////////
174   input          TD_CLK27,
175   input [7:0]    TD_DATA,
176   input          TD_HS,
177   output         TD_RESET_N,
178   input          TD_VS,
179
180
181   ///////// VGA /////////
182   output [7:0]   VGA_B,
183   output         VGA_BLANK_N,
```

```
184   output          VGA_CLK,
185   output  [7:0]   VGA_G,
186   output          VGA_HS,
187   output  [7:0]   VGA_R,
188   output          VGA_SYNC_N,
189   output          VGA_VS
190 );
191
192     soc_system soc_system0(
193       .clk_clk                       ( CLOCK_50 ),
194       .reset_reset_n                 ( 1'b1 ),
195
196       .hps_ddr3_mem_a                ( HPS_DDR3_ADDR ),
197       .hps_ddr3_mem_ba               ( HPS_DDR3_BA ),
198       .hps_ddr3_mem_ck               ( HPS_DDR3_CK_P ),
199       .hps_ddr3_mem_ck_n             ( HPS_DDR3_CK_N ),
200       .hps_ddr3_mem_cke              ( HPS_DDR3_CKE ),
201       .hps_ddr3_mem_cs_n             ( HPS_DDR3_CS_N ),
202       .hps_ddr3_mem_ras_n            ( HPS_DDR3_RAS_N ),
203       .hps_ddr3_mem_cas_n            ( HPS_DDR3_CAS_N ),
204       .hps_ddr3_mem_we_n             ( HPS_DDR3_WE_N ),
205       .hps_ddr3_mem_reset_n          ( HPS_DDR3_RESET_N ),
206       .hps_ddr3_mem_dq               ( HPS_DDR3_DQ ),
207       .hps_ddr3_mem_dqs              ( HPS_DDR3_DQS_P ),
208       .hps_ddr3_mem_dqs_n            ( HPS_DDR3_DQS_N ),
209       .hps_ddr3_mem_odt              ( HPS_DDR3_ODT ),
210       .hps_ddr3_mem_dm               ( HPS_DDR3_DM ),
211       .hps_ddr3_oct_rzqin            ( HPS_DDR3_RZQ ),
212
213       .hps_hps_io_emac1_inst_TX_CLK ( HPS_ENET_GTX_CLK ),
214       .hps_hps_io_emac1_inst_TXD0    ( HPS_ENET_TX_DATA[0] ),
215       .hps_hps_io_emac1_inst_TXD1    ( HPS_ENET_TX_DATA[1] ),
216       .hps_hps_io_emac1_inst_TXD2    ( HPS_ENET_TX_DATA[2] ),
217       .hps_hps_io_emac1_inst_TXD3    ( HPS_ENET_TX_DATA[3] ),
218       .hps_hps_io_emac1_inst_RXD0    ( HPS_ENET_RX_DATA[0] ),
219       .hps_hps_io_emac1_inst_MDIO    ( HPS_ENET_MDIO  ),
220       .hps_hps_io_emac1_inst_MDC     ( HPS_ENET_MDC   ),
221       .hps_hps_io_emac1_inst_RX_CTL  ( HPS_ENET_RX_DV ),
222       .hps_hps_io_emac1_inst_TX_CTL  ( HPS_ENET_TX_EN ),
223       .hps_hps_io_emac1_inst_RX_CLK  ( HPS_ENET_RX_CLK ),
224       .hps_hps_io_emac1_inst_RXD1    ( HPS_ENET_RX_DATA[1]  ),
225       .hps_hps_io_emac1_inst_RXD2    ( HPS_ENET_RX_DATA[2]  ),
226       .hps_hps_io_emac1_inst_RXD3    ( HPS_ENET_RX_DATA[3]  ),
227
228       .hps_hps_io_sdio_inst_CMD      ( HPS_SD_CMD           ),
229       .hps_hps_io_sdio_inst_D0       ( HPS_SD_DATA[0]       ),
230       .hps_hps_io_sdio_inst_D1       ( HPS_SD_DATA[1]       ),
231       .hps_hps_io_sdio_inst_CLK      ( HPS_SD_CLK           ),
232       .hps_hps_io_sdio_inst_D2       ( HPS_SD_DATA[2]       ),
233       .hps_hps_io_sdio_inst_D3       ( HPS_SD_DATA[3]       ),
234
235       .hps_hps_io_usb1_inst_D0       ( HPS_USB_DATA[0]      ),
236       .hps_hps_io_usb1_inst_D1       ( HPS_USB_DATA[1]      ),
237       .hps_hps_io_usb1_inst_D2       ( HPS_USB_DATA[2]      ),
238       .hps_hps_io_usb1_inst_D3       ( HPS_USB_DATA[3]      ),
239       .hps_hps_io_usb1_inst_D4       ( HPS_USB_DATA[4]      ),
240       .hps_hps_io_usb1_inst_D5       ( HPS_USB_DATA[5]      ),
241       .hps_hps_io_usb1_inst_D6       ( HPS_USB_DATA[6]      ),
242       .hps_hps_io_usb1_inst_D7       ( HPS_USB_DATA[7]      ),
243       .hps_hps_io_usb1_inst_CLK      ( HPS_USB_CLKOUT       ),
244       .hps_hps_io_usb1_inst_STP      ( HPS_USB_STP          ),
245       .hps_hps_io_usb1_inst_DIR      ( HPS_USB_DIR          ),
```

```
246       .hps_hps_io_usb1_inst_NXT     ( HPS_USB_NXT         ),
247
248       .hps_hps_io_spim1_inst_CLK    ( HPS_SPIM_CLK  ),
249       .hps_hps_io_spim1_inst_MOSI   ( HPS_SPIM_MOSI ),
250       .hps_hps_io_spim1_inst_MISO   ( HPS_SPIM_MISO ),
251       .hps_hps_io_spim1_inst_SS0    ( HPS_SPIM_SS   ),
252
253       .hps_hps_io_uart0_inst_RX     ( HPS_UART_RX     ),
254       .hps_hps_io_uart0_inst_TX     ( HPS_UART_TX     ),
255
256       .hps_hps_io_i2c0_inst_SDA     ( HPS_I2C1_SDAT     ),
257       .hps_hps_io_i2c0_inst_SCL     ( HPS_I2C1_SCLK     ),
258
259       .hps_hps_io_i2c1_inst_SDA     ( HPS_I2C2_SDAT     ),
260       .hps_hps_io_i2c1_inst_SCL     ( HPS_I2C2_SCLK     ),
261
262       .hps_hps_io_gpio_inst_GPIO09  ( HPS_CONV_USB_N ),
263       .hps_hps_io_gpio_inst_GPIO35  ( HPS_ENET_INT_N ),
264       .hps_hps_io_gpio_inst_GPIO40  ( HPS_LTC_GPIO ),
265
266       .hps_hps_io_gpio_inst_GPIO48  ( HPS_I2C_CONTROL ),
267       .hps_hps_io_gpio_inst_GPIO53  ( HPS_LED ),
268       .hps_hps_io_gpio_inst_GPIO54  ( HPS_KEY ),
269       .hps_hps_io_gpio_inst_GPIO61  ( HPS_GSENSOR_INT ),
270       .vga_r (VGA_R),
271       .vga_g (VGA_G),
272       .vga_b (VGA_B),
273       .vga_clk (VGA_CLK),
274       .vga_hs (VGA_HS),
275       .vga_vs (VGA_VS),
276       .vga_blank_n (VGA_BLANK_N),
277       .vga_sync_n (VGA_SYNC_N)
278    );
279
280    // The following quiet the "no driver" warnings for output
281    // pins and should be removed if you use any of these peripherals
282
283    assign ADC_CS_N = SW[1] ? SW[0] : 1'bZ;
284    assign ADC_DIN = SW[0];
285    assign ADC_SCLK = SW[0];
286
287    assign AUD_ADCLRCK = SW[1] ? SW[0] : 1'bZ;
288    assign AUD_BCLK = SW[1] ? SW[0] : 1'bZ;
289    assign AUD_DACDAT = SW[0];
290    assign AUD_DACLRCK = SW[1] ? SW[0] : 1'bZ;
291    assign AUD_XCK = SW[0];
292
293    assign DRAM_ADDR = { 13{ SW[0] } };
294    assign DRAM_BA = { 2{ SW[0] } };
295    assign DRAM_DQ = SW[1] ? { 16{ SW[0] } } : 16'bZ;
296    assign {DRAM_CAS_N, DRAM_CKE, DRAM_CLK, DRAM_CS_N,
297      DRAM_LDQM, DRAM_RAS_N, DRAM_UDQM, DRAM_WE_N} = { 8{SW[0]} };
298
299    assign FAN_CTRL = SW[0];
300
301    assign FPGA_I2C_SCLK = SW[0];
302    assign FPGA_I2C_SDAT = SW[1] ? SW[0] : 1'bZ;
303
304    assign GPIO_0 = SW[1] ? { 36{ SW[0] } } : 36'bZ;
305    assign GPIO_1 = SW[1] ? { 36{ SW[0] } } : 36'bZ;
306
307    assign HEX0 = { 7{ SW[1] } };
```

```
308     assign HEX1 = { 7{ SW[2] } };
309     assign HEX2 = { 7{ SW[3] } };
310     assign HEX3 = { 7{ SW[4] } };
311     assign HEX4 = { 7{ SW[5] } };
312     assign HEX5 = { 7{ SW[6] } };
313
314     assign IRDA_TXD = SW[0];
315
316     assign LEDR = { 10{SW[7]} };
317
318     assign PS2_CLK = SW[1] ? SW[0] : 1'bZ;
319     assign PS2_CLK2 = SW[1] ? SW[0] : 1'bZ;
320     assign PS2_DAT = SW[1] ? SW[0] : 1'bZ;
321     assign PS2_DAT2 = SW[1] ? SW[0] : 1'bZ;
322
323     assign TD_RESET_N = SW[0];
324
325 //    //assign {VGA_R, VGA_G, VGA_B} = { 24{ SW[0] } };
326 //    //assign {VGA_BLANK_N, VGA_CLK,
327 //      VGA_HS, VGA_SYNC_N, VGA_VS} = { 5{ SW[0] } };
328
329
330 endmodule
```

soc_system_top.sv The following files are included in the tarball submission but not in the code listing: processor_hw.tcl, soc_system.tcl, soc_system.qsys, mem.v, soc_system_board_info.xml

10.2 Software files

```
1  #ifndef _FPGA_RAM_H
2  #define _FPGA_RAM_H
3
4  #include <linux/ioctl.h>
5
6  #define FPGA_RAM_MAGIC 'q'
7
8  typedef struct {
9    unsigned int address;
10   unsigned int data;
11   unsigned int readdata;
12 } fpga_ram_arg_t;
13
14 /* ioctls and their arguments */
15 #define FPGA_RAM_WRITE _IOW(FPGA_RAM_MAGIC, 1, fpga_ram_arg_t *)
16 #define FPGA_RAM_READ  _IOR(FPGA_RAM_MAGIC, 2, fpga_ram_arg_t *)
17 #define FPGA_RAM_READ_L  _IOR(FPGA_RAM_MAGIC, 3, fpga_ram_arg_t *)
18 #define FPGA_RAM_WRITE_L  _IOR(FPGA_RAM_MAGIC, 4, fpga_ram_arg_t *)
19
20 #endif
```

fpga_ram.h

```
1  /* * Device driver for FPGA memory
2   *
3   * A Platform device implemented using the misc subsystem
4   *
5   * Stephen A. Edwards
6   * Columbia University
7   *
8   * Modified by 1802 Team
```

```
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod fpga_ram.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree fpga_ram.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "fpga_ram.h"

#define DRIVER_NAME "vga_ball"
#define VIRT_OFF(x,a) ((x)+(a))
#define VIRT_OFF_LONG(x, a) ((x)+(4*(a)))

/*
 * Information about our device
 */
struct fpga_ram_dev {
  struct resource res; /* Resource: our registers */
  void __iomem *virtbase; /* Where registers can be accessed in memory */
    fpga_ram_arg_t ram_args;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_ram(fpga_ram_arg_t *ram_args)
{

    iowrite8(ram_args->data, VIRT_OFF(dev.virtbase, ram_args->address));
    dev.ram_args = *ram_args;

}

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_ram_long(fpga_ram_arg_t *ram_args)
{

    iowrite32(ram_args->data, VIRT_OFF_LONG(dev.virtbase, ram_args->address));
```

```
71      dev.ram_args = *ram_args;
72
73  }
74
75
76  static void read_ram(fpga_ram_arg_t *ram_args)
77  {
78      ram_args->data = ioread8(VIRT_OFF(dev.virtbase, ram_args->address));
79  }
80
81  static void read_ram_long(fpga_ram_arg_t *ram_args)
82  {
83      ram_args->data = ioread32(VIRT_OFF_LONG(dev.virtbase, ram_args->address));
84  }
85
86  /*
87   * Handle ioctl() calls from userspace:
88   * Read or write the segments on single digits.
89   * Note extensive error checking of arguments
90   */
91  static long fpga_ram_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
92  {
93    fpga_ram_arg_t vla;
94
95    switch (cmd) {
96    case FPGA_RAM_WRITE:
97      if (copy_from_user(&vla, (fpga_ram_arg_t *) arg,
98              sizeof(fpga_ram_arg_t)))
99        return -EACCES;
100     write_ram(&vla);
101     break;
102
103   case FPGA_RAM_WRITE_L:
104     if (copy_from_user(&vla, (fpga_ram_arg_t *) arg,
105             sizeof(fpga_ram_arg_t)))
106       return -EACCES;
107     write_ram_long(&vla);
108     break;
109   case FPGA_RAM_READ:
110                 //printk("addr:%d\n", arg);
111         if (copy_from_user(&vla,(fpga_ram_arg_t *) arg,
112                             sizeof (fpga_ram_arg_t)))
113                  return -EACCES;
114         read_ram(&vla);
115     if (copy_to_user((fpga_ram_arg_t *) arg, &vla,
116          sizeof(fpga_ram_arg_t)))
117       return -EACCES;
118     break;
119
120   case FPGA_RAM_READ_L:
121                 //printk("addr:%d\n", arg);
122         if (copy_from_user(&vla,(fpga_ram_arg_t *) arg,
123                             sizeof (fpga_ram_arg_t)))
124                  return -EACCES;
125         read_ram_long(&vla);
126     if (copy_to_user((fpga_ram_arg_t *) arg, &vla,
127          sizeof(fpga_ram_arg_t)))
128       return -EACCES;
129     break;
130   default:
131     return -EINVAL;
132   }
```

```
133
134   return 0;
135 }
136
137 /* The operations our device knows how to do */
138 static const struct file_operations fpga_ram_fops = {
139   .owner    = THIS_MODULE,
140   .unlocked_ioctl = fpga_ram_ioctl,
141 };
142
143 /* Information about our device for the "misc" framework -- like a char dev */
144 static struct miscdevice fpga_ram_misc_device = {
145   .minor    = MISC_DYNAMIC_MINOR,
146   .name   = DRIVER_NAME,
147   .fops   = &fpga_ram_fops,
148 };
149
150 /*
151  * Initialization code: get resources (registers) and display
152  * a welcome message
153  */
154 static int __init fpga_ram_probe(struct platform_device *pdev)
155 {
156     // no need to write anything   fpga_ram_arg_t zeros = {  };
157   int ret;
158
159   /* Register ourselves as a misc device: creates /dev/vga_ball */
160   ret = misc_register(&fpga_ram_misc_device);
161
162   /* Get the address of our registers from the device tree */
163   ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
164   if (ret) {
165     ret = -ENOENT;
166     goto out_deregister;
167   }
168
169   /* Make sure we can use these registers */
170   if (request_mem_region(dev.res.start, resource_size(&dev.res),
171              DRIVER_NAME) == NULL) {
172     ret = -EBUSY;
173     goto out_deregister;
174   }
175
176   /* Arrange access to our registers */
177   dev.virtbase = of_iomap(pdev->dev.of_node, 0);
178   if (dev.virtbase == NULL) {
179     ret = -ENOMEM;
180     goto out_release_mem_region;
181   }
182
183   /* Set an initial mem val?  */
184       //  write_ram(&zeros);
185
186   return 0;
187
188 out_release_mem_region:
189   release_mem_region(dev.res.start, resource_size(&dev.res));
190 out_deregister:
191   misc_deregister(&fpga_ram_misc_device);
192   return ret;
193 }
194
```

```
195 /* Clean-up code: release resources */
196 static int fpga_ram_remove(struct platform_device *pdev)
197 {
198   iounmap(dev.virtbase);
199   release_mem_region(dev.res.start, resource_size(&dev.res));
200   misc_deregister(&fpga_ram_misc_device);
201   return 0;
202 }
203
204 /* Which "compatible" string(s) to search for in the Device Tree */
205 #ifdef CONFIG_OF
206 static const struct of_device_id fpga_ram_of_match[] = {
207   { .compatible = "csee4840,vga_ball-1.0" },
208   {},
209 };
210 MODULE_DEVICE_TABLE(of, fpga_ram_of_match);
211 #endif
212
213 /* Information for registering ourselves as a "platform" driver */
214 static struct platform_driver fpga_ram_driver = {
215   .driver = {
216     .name = DRIVER_NAME,
217     .owner  = THIS_MODULE,
218     .of_match_table = of_match_ptr(fpga_ram_of_match),
219   },
220   .remove = __exit_p(fpga_ram_remove),
221 };
222
223 /* Called when the module is loaded: set things up */
224 static int __init fpga_ram_init(void)
225 {
226   pr_info(DRIVER_NAME ":␣init\n");
227   return platform_driver_probe(&fpga_ram_driver, fpga_ram_probe);
228 }
229
230 /* Calball when the module is unloaded: release resources */
231 static void __exit fpga_ram_exit(void)
232 {
233   platform_driver_unregister(&fpga_ram_driver);
234   pr_info(DRIVER_NAME ":␣exit\n");
235 }
236
237 module_init(fpga_ram_init);
238 module_exit(fpga_ram_exit);
239
240 MODULE_LICENSE("GPL");
241 MODULE_AUTHOR("Stephen␣A.␣Edwards,␣Columbia␣University,␣1802␣team");
242 MODULE_DESCRIPTION("FPGA␣RAM␣driver");
```

fpga_ram.c

```
1  /*
2   * Userspace program that communicates with the fpga_ram device driver
3   * through ioctls
4   *
5   * Stephen A. Edwards
6   * Columbia University
7   *
8   * Modified by 1802 Team
9   *
10  */
11
12 #include <stdio.h>
```

```c
#include "fpga_ram.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

#define __BUFF_LEN__ 100

int fpga_ram_fd;

/* Read and print the background color */
void print_mem(unsigned char addr) {
  fpga_ram_arg_t real_vla;
  real_vla.address = addr;
  real_vla.data = 0x04;
  if (ioctl(fpga_ram_fd, FPGA_RAM_READ, &real_vla)) {
      perror("ioctl(FPGA_RAM_READ) failed");
      return;
  }
  printf("%02x\n",
  real_vla.data);

}

/* Set the background color */
void set_mem(fpga_ram_arg_t *vla)
{
  fpga_ram_arg_t real_vla;
  real_vla.address = vla->address;
  real_vla.data = vla->data;
  if (ioctl(fpga_ram_fd, FPGA_RAM_WRITE, &real_vla)) {
      perror("ioctl(FPGA_RAM_WRITE) failed");
      return;
  }
}

int main(int argc, char *argv[])
{
  if(argc == 1) {
    fprintf(stderr, "Error: Program requires file name to load\n");
    return -1;
  }
  FILE *prog = fopen(argv[1], "rb");
  if(!prog) {
    perror("Error opening file: ");
    return -1;
  }

  static const char filename[] = "/dev/vga_ball";
  printf("FPGA RAM Userspace program started\n");
  fpga_ram_arg_t my_mem_args;
  my_mem_args.data = 0x07;
  my_mem_args.address = 0x07;

  if ( (fpga_ram_fd = open(filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", filename);
    fclose(prog);
    return -1;
  }

```

```
  unsigned char buf[__BUFF_LEN__] = {0};
  size_t rd;
  unsigned char cond = 1;
  while(cond) {
    if( (rd = fread(buf, 1, __BUFF_LEN__, prog)) != __BUFF_LEN__) {
      if(ferror(prog)) {
        fprintf(stderr, "Error reading file, exiting.\n");
        fclose(prog);
        return 1;
      }
      cond = 0;
    }
    for(size_t i = 0; i < rd; ++i) {
      my_mem_args.data = buf[i];
      my_mem_args.address = i;
      set_mem(&my_mem_args);
    }
  }

  for (int i = 0; i < 32; i ++){

     printf("%d: ", i);
     print_mem(i);
  }

  for (int i = 0; i < 32; i ++){

     printf("%d: ", i);
     print_mem(i);
  }
  printf("FPGA RAM Userspace program terminating\n");
  return 0;
}
```

```
/*
 * Userspace program that communicates with the fpga_ram device driver
 * through ioctls
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * Modified by 1802 Team
 *
 */

#include <stdio.h>
#include "fpga_ram.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

int fpga_ram_fd;

/* Read and print the background color */
void print_mem(unsigned char addr) {
  fpga_ram_arg_t real_vla;
  real_vla.address = addr;
  real_vla.data = 0x04;
  if (ioctl(fpga_ram_fd, FPGA_RAM_READ, &real_vla)) {
```

```
29        perror("ioctl(FPGA_RAM_READ) failed");
30        return;
31    }
32    printf("%02x\n",
33    real_vla.data);
34
35  }
36
37  /* Set the background color */
38  void set_mem(fpga_ram_arg_t *vla)
39  {
40    fpga_ram_arg_t real_vla;
41    real_vla.address = vla->address;
42    real_vla.data = vla->data;
43    //printf("OUTSIDE: %02x\n" , real_vla.data);
44    if (ioctl(fpga_ram_fd, FPGA_RAM_WRITE, &real_vla)) {
45        perror("ioctl(FPGA_RAM_WRITE) failed");
46        return;
47    }
48  }
49
50  int main()
51  {
52    static const char filename[] = "/dev/vga_ball";
53    printf("FPGA RAM Userspace program started\n");
54    fpga_ram_arg_t my_mem_args;
55
56    if ( (fpga_ram_fd = open(filename, O_RDWR)) == -1) {
57      fprintf(stderr, "could not open %s\n", filename);
58      return -1;
59    }
60
61       my_mem_args.data = 0x01;
62       my_mem_args.address = 0x00 + 20;
63       set_mem(&my_mem_args);
64
65    memset(&my_mem_args, 0, sizeof(fpga_ram_arg_t));
66    for (int i = 0; i < 32; i ++){
67
68       printf("%d: ", i);
69       print_mem(i);
70    }
71
72    printf("FPGA RAM Userspace program terminating\n");
73    return 0;
74  }
```

reset.c

```
1  ifneq (${KERNELRELEASE},)
2
3  # KERNELRELEASE defined: we are being compiled as part of the Kernel
4          obj-m := fpga_ram.o
5
6  else
7
8  # We are being compiled as a module: use the Kernel build system
9
10   KERNEL_SOURCE := /usr/src/linux-headers-$(shell uname -r)
11         PWD := $(shell pwd)
12
13 default: module hello print turnon print_long
14
```

```
module:
  ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules

clean:
  ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
  ${RM} hello

TARFILES = Makefile README fpga_ram.h fpga_ram.c hello.c print.c
TARFILE = lab3-sw.tar.gz
.PHONY : tar
tar : $(TARFILE)

$(TARFILE) : $(TARFILES)
  tar zcfC $(TARFILE) .. $(TARFILES:%=lab3-sw/%)

endif
```

Makefile

The following files are additional tests and can be found in the tarball:basic5.c, hello_writes_3.c,test.sh,test_AN.c, basic6.c ,test_2N.c , test_DN.c, basic.c, print.c,test_30.c, test_F0.c,basic1.c,print_long.c ,test_4N.c , test_F8.c, basic2.c , test_5N.c,turnon.c, basic3.c ,setup.c, test_60.c,basic4.c,switchon.c,test_9N_BN.c

10.3 Verilog files

```
#include <stdio.h>
#include "Vprocessor.h"

#ifndef _UTILS_H_
#define _UTILS_H_

#define PROC(x) processor__DOT__ ## x
#define VCPU(x) processor__DOT__c__DOT__ ## x

#define S_FETCH 0
#define S_EXECUTE 1
#define S_EXECUTE2 2

void print_arr(SData *arr, int len) {
    for(int i = 0;i < len; ++i)
        fprintf(stderr, "%04x ", arr[i]);
}

void print_mem(CData *arr, int len) {
    for(int i = 0;i < len; ++i)
        fprintf(stderr, "%02x ", arr[i]);
}

void print_registers(Vprocessor *tb) {
    fprintf(stderr,
            "Misc. registers: state=%02x counter=%02x, N=%02x X=%02x I=%02x D=%02x\n",
            tb->processor__DOT__c__DOT__state,
            tb->processor__DOT__c__DOT__clock_counter,
            tb->processor__DOT__c__DOT__N_register,
            tb->processor__DOT__c__DOT__X_register,
            tb->processor__DOT__c__DOT__I_register,
            tb->processor__DOT__c__DOT__D_register);

    fprintf(stderr, "R_registers: ");
    print_arr(tb->processor__DOT__c__DOT__R_registers, 16);
```

```
}

void print_state(Vprocessor *tb) {
    print_registers(tb);
    fprintf(stderr, "\nMemory: ");
    print_mem(tb->processor__DOT__m__DOT__mem, 32);
    fprintf(stderr, "\n");
}

void clock_cycle(Vprocessor *tb) {
    tb->clk = 0;
    tb->eval();
    tb->clk = 1;
    tb->eval();
}

void machine_cycle(Vprocessor *tb) {
    for (int i = 0; i < 3; i++) {
        clock_cycle(tb);
    }
}

#endif
```

utils.h

```
#include <assert.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#include <functional>
#include <string>

#include "Vprocessor.h"
#include "verilated.h"
#include "verilator_test.h"
#include "utils.h"

#define RESET "\x1B[0m"
#define RED   "\x1B[31m"
#define GREEN "\x1B[32m"

#define NBRANCH_PROG(x) (x), sizeof(x), sizeof(x)
#define PROG(x) (x), sizeof(x)

#define STRINGIZE(x) #x
#define LINE(x) STRINGIZE(x)
#define ASSERT(x) \
    if (!(x)) throw (const char *)(#x " (" __FILE__ ":" LINE(__LINE__) ")");

typedef Vprocessor*        Vtp;
typedef const Vprocessor*  CVtp;

void run_test(Vprocessor *tb,
        const char *name,
        const char *prog,
        size_t prog_len,
        size_t instr_cycles,
        void (*setup)(Vtp, void*),
        void (*verifier)(CVtp, void*),
        void *data)
```

```
38  {
39      try {
40          memset(tb->processor__DOT__m__DOT__mem, 0, sizeof(tb->processor__DOT__m__DOT__mem));
41          memset(tb->VCPU(R_registers), 0xCC, 32);
42          tb->VCPU(alu_operand) = 0xCC;
43          tb->VCPU(alu_result) = 0xCC;
44          tb->VCPU(D_register) = 0xCC;
45
46          tb->reset = 1;
47          clock_cycle(tb);
48
49          ASSERT(tb->VCPU(R_registers[0]) == 0);
50          ASSERT(tb->VCPU(I_register) == 0);
51          ASSERT(tb->VCPU(N_register) == 0);
52          ASSERT(tb->VCPU(P_register) == 0);
53          ASSERT(tb->VCPU(Q) == 0);
54          ASSERT(tb->PROC(bus_from_cpu) == 0);
55          ASSERT(tb->PROC(ram_we) == 0);
56          ASSERT(tb->PROC(address_from_cpu) == 0);
57          ASSERT(tb->VCPU(clock_counter) == 0);
58          ASSERT(tb->VCPU(state) == S_FETCH);
59
60          tb->reset = 0;
61          ASSERT(tb->VCPU(clock_counter) == 0);
62
63          memcpy(tb->processor__DOT__m__DOT__mem, prog, prog_len);
64          setup(tb, data);
65
66          for (int i = 0; i < instr_cycles; i++) {
67              machine_cycle(tb);
68              machine_cycle(tb);
69
70              if (tb->VCPU(I_register) == 0xC) {
71                  machine_cycle(tb);
72              }
73
74              ASSERT(tb->VCPU(state) == S_FETCH);
75          }
76
77          verifier(tb, data);
78          fprintf(stderr, RESET GREEN "[P] %s\n" RESET, name);
79      }
80      catch (const char *e) {
81          fprintf(stderr, RESET RED "[F] %s\n" RESET, name);
82          fprintf(stderr, "--> %s\n", e);
83          print_state(tb);
84      }
85  }
86
87  void tb_noop(Vtp tb, void*){}
88
89  void run_basic_test(Vprocessor *tb,
90          const char *name,
91          const char *prog,
92          size_t prog_len,
93          size_t instr_cycles,
94          void (*f)(CVtp, void*),
95          void *data)
96  {
97      run_test(tb, name, prog, prog_len, instr_cycles, tb_noop, f, data);
98  }
99
```

```
100 int main(int argc, char *argv[])
101 {
102     Verilated::commandArgs(argc, argv);
103     Vprocessor *tb = new Vprocessor;
104     char buf[32];
105 
106     ////////////////////////////////////////////////////////////////////////
107     // 00 (IDL) Check that CPU stops after instruction
108     //
109     // We're on CentOS 6 with G++ 4.4, so no lambdas. Emulate them with
110     // structs with static functions instead, which is kind of gross.
111 
112     struct IDLTest1 {
113         static void verify_00_IDL(CVtp tb, void*) {
114             //ASSERT(tb->VCPU(idle));
115             ASSERT(tb->VCPU(R_registers[1] == 0));
116             ASSERT(tb->VCPU(R_registers[0] == 2));
117         }
118     };
119 
120     run_basic_test(tb,
121         "00␣IDL",
122         NBRANCH_PROG("\x00\x01\x01\x01"),
123         IDLTest1::verify_00_IDL,
124         NULL);
125 
126     ////////////////////////////////////////////////////////////////////////
127     // 01-0F (LDN)
128 
129     struct LDNTest1 {
130         static void verify_0N_LDN(CVtp tb, void *data) {
131             ASSERT(tb->VCPU(D_register) == 0xB3);
132         }
133 
134         static void setup_0N_LDN(Vtp tb, void *data) {
135             tb->VCPU(R_registers[(size_t)data]) = 0xB3;
136         }
137     };
138 
139     for (int x = 0x01; x <= 0x0F; x++) {
140         char c = (char)x;
141         sprintf(buf, "%02X␣LDN␣R(N)␣->␣D", x);
142         run_test(tb, buf, &c, 1, 1,
143                 LDNTest1::setup_0N_LDN,
144                 LDNTest1::verify_0N_LDN,
145                 (void*)(x & 0x0F));
146     }
147 
148     ////////////////////////////////////////////////////////////////////////
149     // 10-1F (INC)
150 
151     char inc_prog1[7];
152 
153     // Testing INC R(0) is hard since R(0) is the default program counter.
154     // Need to do setup to switch P to R(3), reset R(0), and increment.
155     char inc_prog2[] = "\xF8\x07\xA3\xF8\x00\xB3\xD3\xA0\x10\x10\x10\x10\x10";
156 
157     struct INCTest1 {
158         static void setup_1N_INC(Vtp tb, void* data) {
159             tb->VCPU(R_registers[(size_t)data]) = 0;
160         }
161 
```

```
162         static void verify_1N_INC(CVtp tb, void *data) {
163             ASSERT(tb->VCPU(R_registers[(size_t)data]) == 7);
164         }
165     };
166 
167     struct INCTest2 {
168         static void verify_10_INC(CVtp tb, void*) {
169             ASSERT(tb->VCPU(R_registers[0]) == 5);
170         }
171     };
172 
173     run_basic_test(tb,
174         "10␣INC␣R(0)",
175         NBRANCH_PROG(inc_prog2),
176         INCTest2::verify_10_INC,
177         NULL);
178 
179     for (int x = 0x11; x <= 0x1F; x++) {
180         sprintf(buf, "%02X␣INC␣R(N)", x);
181         memset(inc_prog1, x, sizeof(inc_prog1));
182 
183         run_test(tb,
184             buf,
185             NBRANCH_PROG(inc_prog1),
186             INCTest1::setup_1N_INC,
187             INCTest1::verify_1N_INC,
188             (void*)(x & 0x0F));
189     }
190 
191     ////////////////////////////////////////////////////////////////////////
192     // 20-2F (DEC)
193 
194     char dec_prog1[3];
195 
196     // Same issue as INC on R(0) - R(0) is the program counter
197     char dec_prog2[] = "\xF8\x07\xA3\xF8\x00\xB3\xD3\xA0\x20\x20\x20";
198 
199     struct DECTest1 {
200         static void setup_2N_DEC(Vtp tb, void* data) {
201             tb->VCPU(R_registers[(size_t)data]) = 13;
202         }
203 
204         static void verify_2N_DEC(CVtp tb, void* data) {
205             tb->VCPU(R_registers[(size_t)data]) == 10;
206         }
207     };
208 
209     struct DECTest2 {
210         static void verify_20_DEC(CVtp tb, void*) {
211             tb->VCPU(R_registers[0]) == 0xFFFD;
212         }
213     };
214 
215     run_basic_test(tb,
216         "20␣DEC␣R(0)",
217         NBRANCH_PROG(dec_prog2),
218         DECTest2::verify_20_DEC,
219         NULL);
220 
221     for (int x = 0x21; x <= 0x2F; x++) {
222         sprintf(buf, "%02X␣DEC␣R(N)", x);
223         memset(dec_prog1, x, sizeof(dec_prog1));
```

```
224
225         run_test(tb,
226             buf,
227             NBRANCH_PROG(dec_prog1),
228             DECTest1::setup_2N_DEC,
229             DECTest1::verify_2N_DEC,
230             (void*)(x & 0x0F));
231     }
232
233     ////////////////////////////////////////////////////////////////////////
234     // 30-3F (SHORT BRANCH)
235
236     char br_test_progs1[13][20] = {
237         // 30: unconditional
238         // End condition: R(0) == 0x04 after two instructions
239         "\x30\x0F\x30\x06\x00\x00\x00\x00\x00\x30\x0C\x30\x05\x00\x00\x30\x04",
240
241         // 31: if Q == 1 || 39: if Q == 0
242         // End condition: R(0) == 0x08 after six instructions
243         "\x31\x00\x39\x06\x30\x00\x7B\x31\x0B\x30\x00\x7A\x39\x08",
244
245         // 32: if D == 0 || 3A: if D != 0
246         // End condition: R(0) == 0x0F after six instructions
247         "\xF8\x00\x32\x05\x00\xF8\x03\x3A\x0A\x00\xF8\x00\x32\x0F",
248
249         // 33: if DF == 1 || 3B: if DF == 0
250         // End condition: R(0) == 0x0C after five instructions
251         "\x33\xFF\x3B\x05\x00\xF8\x01\x76\x33\x0C\x30\xFE",
252
253         // 38: SKP
254         // End condition: R(0) == 0x02 after one instruction
255         "\x38\x00\x00",
256
257         // 34-37: branch if external flag N == 1
258         // 3C-3F: inverse
259         // End conditions: R(0) == 0x08 after two instructions
260         "\x3C\xFF\x34\x08",
261         "\x3D\xFF\x35\x08",
262         "\x3E\xFF\x36\x08",
263         "\x3F\xFF\x37\x08",
264         "\x34\xFF\x3C\x08",
265         "\x35\xFF\x3D\x08",
266         "\x36\xFF\x3E\x08",
267         "\x37\xFF\x3F\x08"
268     };
269
270     const char *br_prog_names[13] = {
271         "30␣BRANCH␣UNCOND",
272         "31␣BQ␣+␣39␣BNQ",
273         "32␣BZ␣+␣3A␣BNZ",
274         "33␣BDF␣+␣3B␣BNF",
275         "38␣SKP",
276         "34␣B1",
277         "35␣B2",
278         "36␣B3",
279         "37␣B4",
280         "3C␣BN1",
281         "3D␣BN2",
282         "3E␣BN3",
283         "3F␣BN4"
284     };
285
```

```
286     int br_prog_instrs[13] = {2, 6, 6, 5, 1, 2, 2, 2, 2, 2, 2, 2, 2};
287     static int br_prog_targets[13] = {0x04, 0x08, 0x0F, 0x0C, 0x02, 0x08, 0x08, 0x08, 0x08, 0x08, 0x
288
289     struct ShortBranchTest1 {
290         static void setup_3N_BR(Vtp tb, void *data) {
291             size_t x = (size_t)data;
292
293             if (x >= 5 && x <= 8) {
294                 tb->VCPU(B_registers[x - 5]) = 1;
295             }
296             else if (x >= 9) {
297                 tb->VCPU(B_registers[x - 9]) = 0;
298             }
299         }
300
301         static void verify_3N_BR(CVtp tb, void *data) {
302             ASSERT(tb->VCPU(R_registers[0]) == br_prog_targets[(size_t)data]);
303         }
304     };
305
306     for (int x = 0; x < 13; x++) {
307         run_test(tb,
308             br_prog_names[x],
309             PROG(br_test_progs1[x]),
310             br_prog_instrs[x],
311             ShortBranchTest1::setup_3N_BR,
312             ShortBranchTest1::verify_3N_BR,
313             (void*)x);
314     }
315
316     ////////////////////////////////////////////////////////////////////////
317     // 40-4F (LOAD & ADVANCE)
318
319     char lda_test_prog1[] = "\x40\xE0\x00\x00\x00\x40\x09\x00\x00\xE0";
320
321     struct LoadAdvTest1 {
322         static void setup_4N_LDA(Vtp tb, void *data) {
323             size_t x = (size_t)data & 0x0F;
324             if (x != 0) {
325                 tb->VCPU(R_registers[x]) = 0x09;
326             }
327         }
328
329         static void verify_4N_LDA(CVtp tb, void *data) {
330             size_t x = (size_t)data & 0x0F;
331             ASSERT(tb->VCPU(D_register) == 0xE0);
332             ASSERT(tb->VCPU(R_registers[x]) == (x ? 0x0A : 0x02));
333         }
334     };
335
336     for (int x = 0x40; x <= 0x4F; x++) {
337         lda_test_prog1[0] = lda_test_prog1[5] = x;
338         sprintf(buf, "%02X␣LDA␣*R(%d)␣->␣D", x, (x & 0x0F));
339         run_test(tb, buf,
340                 PROG(lda_test_prog1), 1,
341                 LoadAdvTest1::setup_4N_LDA,
342                 LoadAdvTest1::verify_4N_LDA,
343                 (void*)x);
344     }
345
346     ////////////////////////////////////////////////////////////////////////
347     // 50-5F (STORE TO R(N))
```

```
348
349     char str_test_prog1[] = "\x50\x00\x00\x00\x00\x00";
350
351     struct StoreTest1 {
352         static void setup_5N_STR(Vtp tb, void *data) {
353             tb->VCPU(D_register) = 0x2C;
354
355             size_t x = (size_t)data & 0x0F;
356             if (x != 0) {
357                 tb->VCPU(R_registers[x]) = 0x05;
358             }
359         }
360
361         static void verify_5N_STR(CVtp tb, void *data) {
362             size_t x = (size_t)data & 0x0F;
363             ASSERT(tb->VCPU(D_register) == 0x2C);
364             ASSERT(tb->processor__DOT__m__DOT__mem[x ? 5 : 1] == 0x2C);
365         }
366     };
367
368     for (int x = 0x50; x <= 0x5F; x++) {
369         str_test_prog1[0] = x;
370         sprintf(buf, "%02X␣STR␣D␣->␣*R(%d)", x, (x & 0x0F));
371         run_test(tb, buf,
372                 PROG(str_test_prog1), 1,
373                 StoreTest1::setup_5N_STR,
374                 StoreTest1::verify_5N_STR,
375                 (void*)x);
376     }
377
378     ////////////////////////////////////////////////////////////////////////
379     // 60: INC R(X)
380
381     struct IRXTest1 {
382         static void setup_60_IRX(Vtp tb, void *data) {
383             size_t x = (size_t)data & 0x0F;
384             if (x != 0) {
385                 tb->VCPU(R_registers[x]) = 0xB9;
386             }
387         }
388
389         static void verify_60_IRX(CVtp tb, void *data) {
390             size_t x = (size_t)data & 0x0F;
391             ASSERT(tb->VCPU(R_registers[x]) == x ? 0xC0 : 0x02);
392         }
393     };
394
395     for (int x = 0; x < 16; x++) {
396         char instr = 0x60;
397
398         sprintf(buf, "60␣IRX,␣X␣==␣%d", x);
399         run_test(tb, buf,
400                 &instr, 1, 1,
401                 IRXTest1::setup_60_IRX,
402                 IRXTest1::verify_60_IRX,
403                 (void*)x);
404     }
405
406     ////////////////////////////////////////////////////////////////////////
407     // 61-6F (BUS I/O)
408     // TODO
409
```

```
410     ////////////////////////////////////////////////////////////////////////
411     // 70-7F (CONTROL)
412     // TODO
413
414     ////////////////////////////////////////////////////////////////////////
415     // 80-8F (GLO)
416
417     struct GetLowTest1 {
418         static void setup_8N_GLO(Vtp tb, void *data) {
419             tb->VCPU(D_register) = 0xCC;
420
421             size_t x = (size_t)data & 0x0F;
422             if (x != 0) {
423                 tb->VCPU(R_registers[x]) = 0x2E74;
424             }
425         }
426
427         static void verify_8N_GLO(CVtp tb, void *data) {
428             size_t x = (size_t)data & 0x0F;
429             ASSERT(tb->VCPU(D_register) == (x ? 0x74 : 0x01));
430         }
431     };
432
433     for (unsigned char x = 0x80; x <= 0x8F; x++) {
434         sprintf(buf, "%02X␣GLO␣R(%d).0␣->␣D", x, (x & 0x0F));
435         run_test(tb, buf,
436                 (char*)&x, 1, 1,
437                 GetLowTest1::setup_8N_GLO,
438                 GetLowTest1::verify_8N_GLO,
439                 (void*)x);
440     }
441
442     ////////////////////////////////////////////////////////////////////////
443     // 90-9F (GHI)
444
445     struct GetHighTest1 {
446         static void setup_9N_GHI(Vtp tb, void *data) {
447             tb->VCPU(D_register) = 0xCC;
448
449             size_t x = (size_t)data & 0x0F;
450             if (x != 0) {
451                 tb->VCPU(R_registers[x]) = 0x2E74;
452             }
453         }
454
455         static void verify_9N_GHI(CVtp tb, void *data) {
456             size_t x = (size_t)data & 0x0F;
457             ASSERT(tb->VCPU(D_register) == (x ? 0x2E : 0x00));
458         }
459     };
460
461     for (unsigned char x = 0x90; x <= 0x9F; x++) {
462         sprintf(buf, "%02X␣GHI␣R(%d).1␣->␣D", x, (x & 0x0F));
463         run_test(tb, buf,
464                 (char*)&x, 1, 1,
465                 GetHighTest1::setup_9N_GHI,
466                 GetHighTest1::verify_9N_GHI,
467                 (void*)x);
468     }
469
470     ////////////////////////////////////////////////////////////////////////
471     // A0-AF (PLO)
```

```
472
473     struct PutLowTest1 {
474         static void setup_AN_PLO(Vtp tb, void *data) {
475             size_t x = (size_t)data & 0x0F;
476             tb->VCPU(D_register) = 0xCB;
477             tb->VCPU(R_registers[x]) = 0x0000;
478         }
479
480         static void verify_AN_PLO(CVtp tb, void *data) {
481             size_t x = (size_t)data & 0x0F;
482             ASSERT(tb->VCPU(R_registers[x]) == 0x00CB);
483         }
484     };
485
486     for (unsigned char x = 0xA0; x <= 0xAF; x++) {
487         sprintf(buf, "%02X␣PLO␣D␣->␣R(%d).0", x, (x & 0x0F));
488         run_test(tb, buf,
489                 (char*)&x, 1, 1,
490                 PutLowTest1::setup_AN_PLO,
491                 PutLowTest1::verify_AN_PLO,
492                 (void*)x);
493     }
494
495     ////////////////////////////////////////////////////////////////////////
496     // B0-BF (PHI)
497
498     struct PutHighTest1 {
499         static void setup_BN_PHI(Vtp tb, void *data) {
500             size_t x = (size_t)data & 0x0F;
501             tb->VCPU(D_register) = 0xCB;
502             tb->VCPU(R_registers[x]) = 0x0000;
503         }
504
505         static void verify_BN_PHI(CVtp tb, void *data) {
506             size_t x = (size_t)data & 0x0F;
507             ASSERT(tb->VCPU(R_registers[x]) == (x ? 0xCB00 : 0xCB01));
508         }
509     };
510
511     for (unsigned char x = 0xB0; x <= 0xBF; x++) {
512         sprintf(buf, "%02X␣PHI␣D␣->␣R(%d).1", x, (x & 0x0F));
513         run_test(tb, buf,
514                 (char*)&x, 1, 1,
515                 PutHighTest1::setup_BN_PHI,
516                 PutHighTest1::verify_BN_PHI,
517                 (void*)x);
518     }
519
520     ////////////////////////////////////////////////////////////////////////
521     // C0-CF (LONG BRANCH)
522
523     ////////////////////// Branches \\\\\\\\\\\\\\\\\\\\
524     // C0: unconditional long branch
525     char lbr_test_prog1[] = "\xC0\x34\x56";
526
527     struct LBRTest {
528         static void verify_taken(CVtp tb, void *data) {
529             ASSERT(tb->VCPU(R_registers[0]) == 0x3456);
530         }
531
532         static void verify_not_taken(CVtp tb, void *data) {
533             ASSERT(tb->VCPU(R_registers[0]) == 0x03);
```

```
534         }
535     };
536
537     run_basic_test(tb, "C0 LBR NOCOND", PROG(lbr_test_prog1), 1, LBRTest::verify_taken, NULL);
538
539     // C1: LBQ (Q == 1)
540     // C9: LBNQ (Q == 0)
541     char lbq_test_prog1[] = "\xC1\x34\x56";
542     char lbq_test_prog2[] = "\xC9\x34\x56";
543
544     struct LBQTest {
545         static void setup_C1C9_LBQ(Vtp tb, void *data) { tb->VCPU(Q) = 1; }
546         static void setup_C1C9_LBNQ(Vtp tb, void *data) { tb->VCPU(Q) = 0; }
547     };
548
549     run_test(tb, "C1 LBQ Q==0", PROG(lbq_test_prog1), 1,
550             LBQTest::setup_C1C9_LBNQ, LBRTest::verify_not_taken, NULL);
551     run_test(tb, "C1 LBQ Q==1", PROG(lbq_test_prog1), 1,
552             LBQTest::setup_C1C9_LBQ, LBRTest::verify_taken, NULL);
553     run_test(tb, "C9 LBNQ Q==0", PROG(lbq_test_prog2), 1,
554             LBQTest::setup_C1C9_LBNQ, LBRTest::verify_taken, NULL);
555     run_test(tb, "C9 LBNQ Q==1", PROG(lbq_test_prog2), 1,
556             LBQTest::setup_C1C9_LBQ, LBRTest::verify_not_taken, NULL);
557
558     // C2: LBZ (D == 0)
559     // CA: LBNZ (D != 0)
560     char lbz_test_prog1[] = "\xC2\x34\x56";
561     char lbz_test_prog2[] = "\xCA\x34\x56";
562
563     struct LBZTest {
564         static void setup_C2CA_LBZ(Vtp tb, void *data) { tb->VCPU(D_register) = 0; }
565         static void setup_C2CA_LBNZ(Vtp tb, void *data) { tb->VCPU(D_register) = 0xFE; }
566     };
567
568     run_test(tb, "C2 LBZ D==0", PROG(lbz_test_prog1), 1,
569             LBZTest::setup_C2CA_LBZ, LBRTest::verify_taken, NULL);
570     run_test(tb, "C2 LBZ D==1", PROG(lbz_test_prog1), 1,
571             LBZTest::setup_C2CA_LBNZ, LBRTest::verify_not_taken, NULL);
572     run_test(tb, "CA LBNZ D==0", PROG(lbz_test_prog2), 1,
573             LBZTest::setup_C2CA_LBZ, LBRTest::verify_not_taken, NULL);
574     run_test(tb, "CA LBNZ D==1", PROG(lbz_test_prog2), 1,
575             LBZTest::setup_C2CA_LBNZ, LBRTest::verify_taken, NULL);
576
577     // C3: LBDF (DF == 1)
578     // CB: LBNF (DF == 0)
579     char lbdf_test_prog1[] = "\xC3\x34\x56";
580     char lbdf_test_prog2[] = "\xCB\x34\x56";
581
582     struct LBDFTest {
583         static void setup_C3CB_LBDF(Vtp tb, void *data) { tb->VCPU(DF) = 1; }
584         static void setup_C3CB_LBNF(Vtp tb, void *data) { tb->VCPU(DF) = 0; }
585     };
586
587     run_test(tb, "C3 LBDF D==0", PROG(lbdf_test_prog1), 1,
588             LBDFTest::setup_C3CB_LBDF, LBRTest::verify_taken, NULL);
589     run_test(tb, "C3 LBDF D==1", PROG(lbdf_test_prog1), 1,
590             LBDFTest::setup_C3CB_LBNF, LBRTest::verify_not_taken, NULL);
591     run_test(tb, "CB LBNF D==0", PROG(lbdf_test_prog2), 1,
592             LBDFTest::setup_C3CB_LBNF, LBRTest::verify_taken, NULL);
593     run_test(tb, "CB LBNF D==1", PROG(lbdf_test_prog2), 1,
594             LBDFTest::setup_C3CB_LBDF, LBRTest::verify_not_taken, NULL);
595
```

```
596     // C4: NOP
597     char nop_test_prog1[] = "\xC4\xC4\xC4";
598     Vprocessor old_state;
599
600     struct NOPTest {
601         static void setup_C4_NOP(Vtp tb, void *data) {
602             memcpy(data, tb, sizeof(Vprocessor));
603         }
604
605         static void verify_C4_NOP(CVtp tb, void *data) {
606             Vprocessor *old = (Vprocessor *)data;
607             old->VCPU(R_registers[0]) += 3 + 1; // + 1 since FETCH will increase PC
608
609             ASSERT(memcmp(old->processor__DOT__m__DOT__mem,
610                           tb->processor__DOT__m__DOT__mem,
611                           sizeof(tb->processor__DOT__m__DOT__mem)) == 0);
612
613             ASSERT(memcmp(old->VCPU(R_registers),
614                           tb->VCPU(R_registers),
615                           sizeof(tb->VCPU(R_registers))) == 0);
616
617             ASSERT(memcmp(old->VCPU(B_registers),
618                           tb->VCPU(B_registers),
619                           sizeof(tb->VCPU(B_registers))) == 0);
620
621             ASSERT(old->VCPU(mode) == tb->VCPU(mode));
622             ASSERT(old->VCPU(state) == tb->VCPU(state));
623             ASSERT(old->VCPU(alu_operand) == tb->VCPU(alu_operand));
624             ASSERT(old->VCPU(alu_result) == tb->VCPU(alu_result));
625             ASSERT(old->VCPU(alu_carry) == tb->VCPU(alu_carry));
626             ASSERT(old->VCPU(Q) == tb->VCPU(Q));
627             ASSERT(old->VCPU(DF) == tb->VCPU(DF));
628             ASSERT(old->VCPU(IE) == tb->VCPU(IE));
629             ASSERT(old->VCPU(N_register) == tb->VCPU(N_register));
630             ASSERT(old->VCPU(P_register) == tb->VCPU(P_register));
631             ASSERT(old->VCPU(X_register) == tb->VCPU(X_register));
632             ASSERT(old->VCPU(I_register) == tb->VCPU(I_register));
633             ASSERT(old->VCPU(D_register) == tb->VCPU(D_register));
634         }
635     };
636
637     run_test(tb, "C4␣NOP", NBRANCH_PROG(nop_test_prog1),
638             NOPTest::setup_C4_NOP, NOPTest::verify_C4_NOP, &old_state);
639
640     ////////////////////// Long skips \\\\\\\\\\\\\\\\\\\\
641     // C5: LNQ (Q == 0)
642     // CD: LSQ (Q == 1)
643     struct LSKPTest {
644         static void verify_taken(CVtp tb, void *) {
645             ASSERT(tb->VCPU(R_registers[0]) == 0x03);
646         }
647
648         static void verify_not_taken(CVtp tb, void *) {
649             ASSERT(tb->VCPU(R_registers[0]) == 0x01);
650         }
651     };
652
653     char lnq_test_prog1 = 0xC5;
654     char lsq_test_prog1 = 0xCD;
655
656     struct LNQTest {
657         static void setup_C5CD_LNQ(Vtp tb, void *) { tb->VCPU(Q) = 0; }
```

```
658         static void setup_C5CD_LSQ(Vtp tb, void *) { tb->VCPU(Q) = 1; }
659     };
660 
661     run_test(tb, "C5␣LNQ␣Q==0", &lnq_test_prog1, 1, 1,
662             LNQTest::setup_C5CD_LNQ, LSKPTest::verify_taken, NULL);
663     run_test(tb, "C5␣LNQ␣Q==1", &lnq_test_prog1, 1, 1,
664             LNQTest::setup_C5CD_LSQ, LSKPTest::verify_not_taken, NULL);
665     run_test(tb, "CD␣LSQ␣Q==0", &lsq_test_prog1, 1, 1,
666             LNQTest::setup_C5CD_LNQ, LSKPTest::verify_not_taken, NULL);
667     run_test(tb, "CD␣LSQ␣Q==1", &lsq_test_prog1, 1, 1,
668             LNQTest::setup_C5CD_LSQ, LSKPTest::verify_taken, NULL);
669 
670     // C6: LSNZ (D != 0)
671     // CE: LSZ (D == 0)
672     char lsnz_test_prog1 = 0xC6;
673     char lsz_test_prog1 = 0xCE;
674 
675     struct LSNZTest {
676         static void setup_C6CE_LSNZ(Vtp tb, void *) { tb->VCPU(D_register) = 0x65; }
677         static void setup_C6CE_LSZ(Vtp tb, void *) { tb->VCPU(D_register) = 0; }
678     };
679 
680     run_test(tb, "C6␣LSNZ␣D!=0", &lsnz_test_prog1, 1, 1,
681             LSNZTest::setup_C6CE_LSNZ, LSKPTest::verify_taken, NULL);
682     run_test(tb, "C6␣LSNZ␣D==0", &lsnz_test_prog1, 1, 1,
683             LSNZTest::setup_C6CE_LSZ, LSKPTest::verify_not_taken, NULL);
684     run_test(tb, "CE␣LSZ␣D!=0", &lsz_test_prog1, 1, 1,
685             LSNZTest::setup_C6CE_LSNZ, LSKPTest::verify_not_taken, NULL);
686     run_test(tb, "CE␣LSZ␣D==0", &lsz_test_prog1, 1, 1,
687             LSNZTest::setup_C6CE_LSZ, LSKPTest::verify_taken, NULL);
688 
689     // C7: LSNF (DF == 0)
690     // CF: LSDF (DF == 1)
691     char lsnf_test_prog1 = 0xC7;
692     char lsdf_test_prog1 = 0xCF;
693 
694     struct LSNFTest {
695         static void setup_C7CF_LSNF(Vtp tb, void *) { tb->VCPU(DF) = 0; }
696         static void setup_C7CF_LSDF(Vtp tb, void *) { tb->VCPU(DF) = 1; }
697     };
698 
699     run_test(tb, "C7␣LSNF␣DF==0", &lsnf_test_prog1, 1, 1,
700             LSNFTest::setup_C7CF_LSNF, LSKPTest::verify_taken, NULL);
701     run_test(tb, "C7␣LSNF␣DF==1", &lsnf_test_prog1, 1, 1,
702             LSNFTest::setup_C7CF_LSDF, LSKPTest::verify_not_taken, NULL);
703     run_test(tb, "CF␣LSDF␣DF==0", &lsdf_test_prog1, 1, 1,
704             LSNFTest::setup_C7CF_LSNF, LSKPTest::verify_not_taken, NULL);
705     run_test(tb, "CF␣LSDF␣DF==1", &lsdf_test_prog1, 1, 1,
706             LSNFTest::setup_C7CF_LSDF, LSKPTest::verify_taken, NULL);
707 
708     // C8: NLBR (uncond. long skip)
709     char nlbr_test_prog1 = 0xC8;
710 
711     run_basic_test(tb, "C8␣NLBR␣uncond.␣long␣skip", &nlbr_test_prog1, 1, 1,
712             LSKPTest::verify_taken, NULL);
713 
714     // CC: LSIE (IE == 1)
715     char lsie_test_prog1 = 0xCC;
716 
717     struct LSIETest {
718         static void setup_CC_POS(Vtp tb, void *) { tb->VCPU(IE) = 1; }
719         static void setup_CC_NEG(Vtp tb, void *) { tb->VCPU(IE) = 0; }
```

```
720     };
721
722     run_test(tb, "CC LSIE IE==0", &lsie_test_prog1, 1, 1,
723             LSIETest::setup_CC_NEG, LSKPTest::verify_not_taken, NULL);
724     run_test(tb, "CC LSIE IE==1", &lsie_test_prog1, 1, 1,
725             LSIETest::setup_CC_POS, LSKPTest::verify_taken, NULL);
726
727     ////////////////////////////////////////////////////////////////////////
728     // D0-DF (SET P TO N)
729
730     struct SetPTest1 {
731         static void verify_DN_SEP(CVtp tb, void *data) {
732             ASSERT(tb->VCPU(P_register) == (size_t)data);
733         }
734     };
735
736     for (unsigned char x = 0xD0; x <= 0xDF; x++) {
737         sprintf(buf, "%02X SEP", x);
738         run_basic_test(tb, buf, (char*)&x, 1, 1, SetPTest1::verify_DN_SEP, (void*)(x & 0x0F));
739     }
740
741     ////////////////////////////////////////////////////////////////////////
742     // E0-EF (SET X TO N)
743
744     struct SetXTest1 {
745         static void verify_EN_SEX(CVtp tb, void *data) {
746             ASSERT(tb->VCPU(X_register) == (size_t)data);
747         }
748     };
749
750     for (unsigned char x = 0xE0; x <= 0xEF; x++) {
751         sprintf(buf, "%02X SEX", x);
752         run_basic_test(tb, buf, (char*)&x, 1, 1, SetXTest1::verify_EN_SEX, (void*)(x & 0x0F));
753     }
754
755     ////////////////////////////////////////////////////////////////////////
756     // F0-FF (ARITH)
757
758     // F0: load via X
759     // TODO
760
761     // F1-FF: OR via X
762     char arith_via_x_test_prog1[] = "\xF1\x54";
763
764     struct ArithViaXTest {
765         static void setup_FN_arith(Vtp tb, void *data) {
766             tb->VCPU(D_register) = 0xC1;
767
768             size_t x = (size_t)data & 0x0F;
769             if (x != 0) {
770                 tb->VCPU(R_registers[x]) = 0x01;
771             }
772         }
773
774         static void verify_IMM(CVtp tb, void *data) {
775             size_t x = (size_t)data;
776             if (x > 0xF8 && x != 0xFE) {
777                 ASSERT(tb->VCPU(R_registers[0]) == 0x03);
778             }
779         }
780
781         static void verify_OR(CVtp tb, void *data) {
```

```
782             ASSERT(tb->VCPU(D_register) == 0xD5);
783             verify_IMM(tb, data);
784         }
785
786         static void verify_AND(CVtp tb, void *data) {
787             ASSERT(tb->VCPU(D_register) == 0x40);
788             verify_IMM(tb, data);
789         }
790
791         static void verify_XOR(CVtp tb, void *data) {
792             ASSERT(tb->VCPU(D_register) == 0x95);
793             verify_IMM(tb, data);
794         }
795
796         static void verify_ADD(CVtp tb, void *data) {
797             ASSERT(tb->VCPU(D_register) == 0x15);
798             ASSERT(tb->VCPU(DF) == 1);
799             verify_IMM(tb, data);
800         }
801
802         static void verify_SUB(CVtp tb, void *data) {
803             ASSERT(tb->VCPU(D_register) == 0x93);
804             ASSERT(tb->VCPU(DF) == 0);
805             verify_IMM(tb, data);
806         }
807
808         static void verify_SHR(CVtp tb, void *data) {
809             ASSERT(tb->VCPU(D_register) == 0x60);
810             ASSERT(tb->VCPU(DF) == 1);
811         }
812
813         static void verify_SHL(CVtp tb, void *data) {
814             ASSERT(tb->VCPU(D_register) == 0x82);
815             ASSERT(tb->VCPU(DF) == 1);
816         }
817
818         static void verify_mem_SUB(CVtp tb, void *data) {
819             ASSERT(tb->VCPU(D_register) == 0x6D);
820             ASSERT(tb->VCPU(DF) == 1);
821             verify_IMM(tb, data);
822         }
823
824         // TODO: F8 (LDI)
825     };
826
827     const char *arith_via_x_names[16] = {
828         NULL, "OR", "AND", "XOR", "ADD", "SD", "SHR", "SM",
829         NULL, "ORI", "ANI", "XRI", "ADI", "SDI", "SHL", "SMI"
830     };
831
832     for (unsigned char x = 0xF1; ; x++) {
833         if (x == 0xF8) continue;
834
835         arith_via_x_test_prog1[0] = x;
836         void (*verifier)(CVtp, void*) = NULL;
837
838         switch (x & 0x07) {
839             case 0x01:
840                 verifier = ArithViaXTest::verify_OR;
841                 break;
842             case 0x02:
843                 verifier = ArithViaXTest::verify_AND;
```

```
                break;
            case 0x03:
                verifier = ArithViaXTest::verify_XOR;
                break;
            case 0x04:
                verifier = ArithViaXTest::verify_ADD;
                break;
            case 0x05:
                verifier = ArithViaXTest::verify_SUB;
                break;
            case 0x06:
                if (x == 0xF6) verifier = ArithViaXTest::verify_SHR;
                else verifier = ArithViaXTest::verify_SHL;
                break;
            case 0x07:
                verifier = ArithViaXTest::verify_mem_SUB;
                break;
        }

        sprintf(buf, "%02X␣%s", x, arith_via_x_names[x & 0x0F]);
        run_test(tb, buf, PROG(arith_via_x_test_prog1), 1,
                ArithViaXTest::setup_FN_arith, verifier, NULL);

        if (x == 0xFF) break;
    }

    return 0;
}
```

opstest.cpp

```
#!/bin/bash
verilator -Wno-WIDTH -Wno-CASEINCOMPLETE --cc ../hw/processor.sv
cd obj_dir
make -f Vprocessor.mk
make -f Vprocessor.mk ../test.o verilated.o
make -f Vprocessor.mk ../counter.o verilated.o
make -f Vprocessor.mk ../opstest.o verilated.o
g++ -g -Wall ../test.o Vprocessor__ALL*.o verilated.o -o ../test
g++ -g -Wall ../counter.o Vprocessor__ALL*.o verilated.o -o ../counter
g++ -g -Wall -std=gnu++0x ../opstest.o Vprocessor__ALL*.o verilated.o -o ../opstest
cd ..
#./test
```

test.sh

10.4 Hardware test programs

```
    PLO 1 ; Initialise R1 to 0
    LDN 1 ; Load op code for PLO 1 to D
    INC 1 ; Increment R1
    STR 1 ; Store D to 1
    IDL
    END
```

test_ldn.asm

```
; Loads via X, which will load memory location 1 as R(0) is 1 after reading LDX
; Stores D to memory location 0, D will be the opcode of STR 1, 0x51
;
; Observables: Memory location 0 will be 0x51 after execution

```

```
    PLO 1
    LDX
    STR 1
    IDL
    END
```

test_ldx.asm

```
#!/bin/bash

assembler="../A18/a18"
asmdir="./binaries"
listdir="./"

printferr() { printf "%b\n" "$*" >&2;}

main() {
    if [ ! -x "$assembler" ];then
        printferr "Error: A18 assembler is not available. Make sure to init git submodules and compi
        exit 1
    fi

    if [ $# -eq 0 ];then
        echo "Usage: $0 <assembly files to assemble>"
        exit
    fi

    local dir
    for dir in "$asmdir" "$listdir";do
        if [ ! -d "$dir" ];then
            printferr "Output directory '$dir' does not exist. Creating directory."
            mkdir -p "$dir"
        fi
    done
    unset dir

    local arg
    for arg in "$@";do
        if [ ! -f "$arg" ];then
            printferr "\e[31mFile '$arg' does not exist; skipping.\e[0m"
            continue
        fi
        local barg="${arg##*/}"
        echo -e "----------------- Assembling file: $arg -----------------\n"
        "$assembler" "$arg" -b "$asmdir/${barg%.*}.bin" -l "$listdir/${barg%.*}.prn" >/dev/null \
            || printferr "\e[31mFile assembled with errors reported. Check listings file for info.\e
    done
}

main "$@"
```

Script using the A18 cross-assembler