# PACS 6502

Akira Baruah akb2158
Chaiwen Chou cc3636
Philip Schiffrin pjs2186
Sean Liu sl3497

# Table of Contents



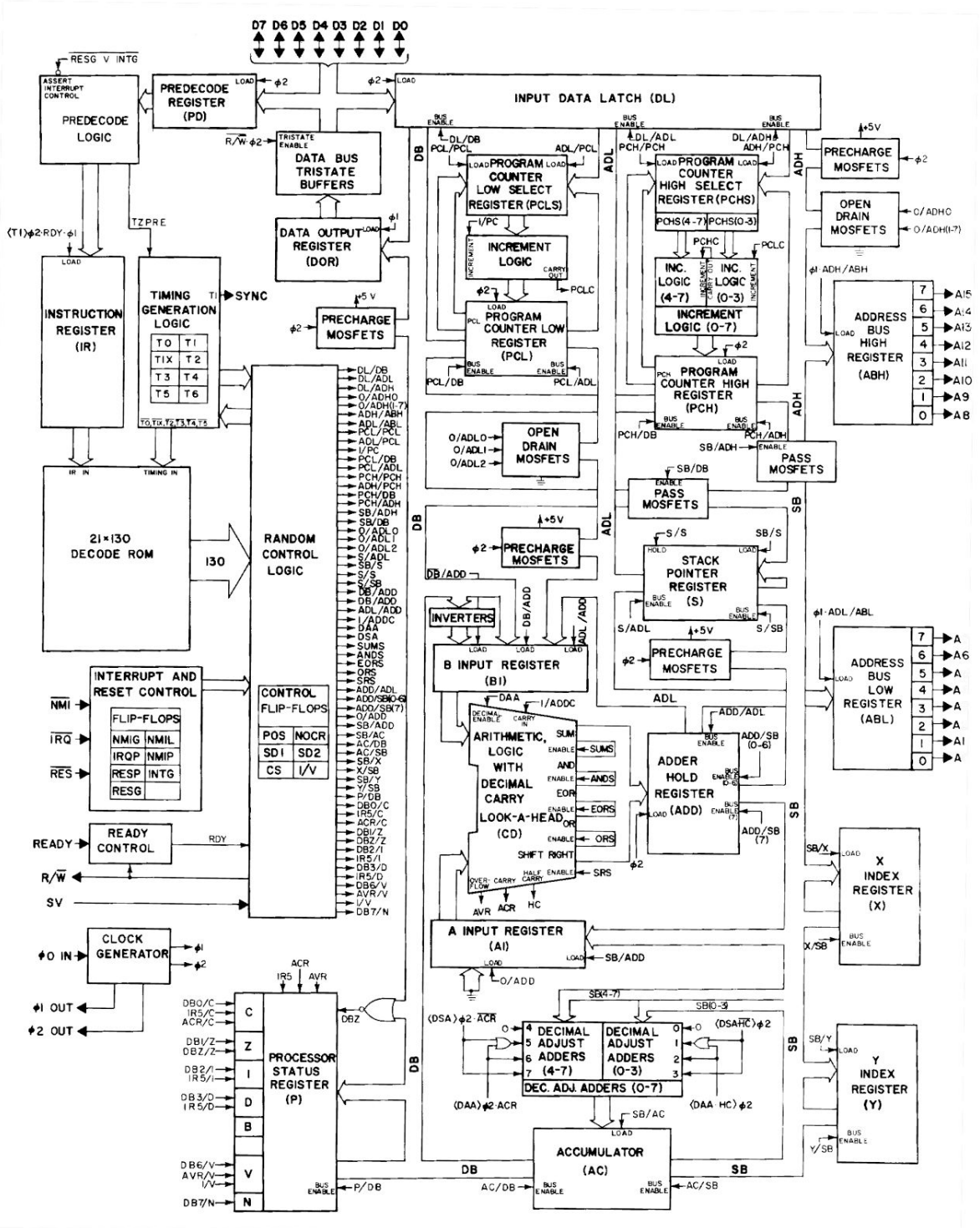
# Overview

Our project is the emulation of the NES 6502 microprocessor on Altera's Cyclone V FPGA. We were originally interested in pursuing this project because we wanted to emulate the Nintendo Entertainment System. However, upon further investigation, emulation of the entire system seemed like too broad of a project and we decided to pursue emulating the 2A03, which is the modified 6502 processor found inside the NES. The only major modification on the 2A03

is the removal of the binary coded decimal mode which was originally found on the 6502; otherwise, the processors have the same internal design.

The overall structure of the project was to implement the 6502 in systemverilog and communicate with software running on the hard processor system. This communication could be done with memory mapping via a user space program at “/dev/mem” or more properly with a kernel module device driver: we ended up using “/dev/mem” because we did not have time to get the kernel module to successfully work with our system. Our ultimate goal was to allow the user to write a compiled 6502 program into memory, run the processor, and read the memory during/after the lifetime of the program. One of the interesting features of the original 6502 is that, unlike modern processors, the user has read/write access to every addressable byte in memory. Thus an unprotected software interface for the user would double as an historically accurate emulation of the processor and a useful tool for debugging user programs. However, due to lack of time, we were not able to successfully write a full program into memory. The current state of the project is that the user can write compiled 6502 into the processor’s memory starting from address 0, but the read and execution of the program is still incomplete.

# About the 6502



# CPU

## Instructions

Instructions on the 6502 are split into different logical groups based on the type of their operation and their addressing modes. However, different resources divided the instructions differently: some group them strictly by common bits in the opcodes, while others group them logically based on their particular function. For instance, one will group together all store instructions given that they perform the same function with different values, while others separate store instructions given that they do not share a common bit pattern in their opcodes.

Instructions that explicitly reference memory locations have bit patterns of the form **aaabbbcc**. The **aaa** bits determine the opcode, the **bbb** bits determine the addressing mode, and the **cc** bits determine the mode.

## Addressing Modes

The 6502 allows the user to address memory in a complex fashion relative to the size of its registers - 8 bits - and the number of registers, which is less than 5. Instead of addressing only 256 bytes of memory, the address bus is 16 bits wide which allows for 65,536 memory addresses. Given this flexibility, the 6502 has seven different addressing modes ranging from immediate addressing, where no address is needed to perform the operation, to complex 6 or 7 cycle operations, where an offset contained in a certain register is added to a given address, and the values at that address are then fetched. This gives the programmer a lot of control in terms of how they want to program with regards to memory.

When **cc** = 01:

| **aaa** | **opcode** |
|---|---|
| 000 | ORA |
| 001 | AND |
| 010 | EOR |
| 011 | ADC |
| 100 | STA |
| 101 | LDA |
| 110 | CMP |
| 111 | SBC |

| **bbb** | **addressing mode** |
|---|---|
| 000 | (zero page, X) |
| 001 | zero page |
| 010 | # immediate |
| 011 | absolute |
| 100 | (zero page), Y |
| 101 | zero page, X |
| 110 | absolute, Y |
| 111 | absolute, X |

When **cc** = 10:

| **aaa** | **opcode** |
|---|---|
| 000 | ASL |

| | |
|---|---|
| 001 | ROL |
| 010 | LSR |
| 011 | ROR |
| 100 | STX |
| 101 | LDX |
| 110 | DEC |
| 111 | INC |

| **bbb** | **addressing mode** |
|---|---|
| 000 | # immediate |
| 001 | zero page |
| 010 | accumulator |
| 011 | absolute |
| 101 | zero page, X |
| 111 | absolute, X |

When **cc** = 00:

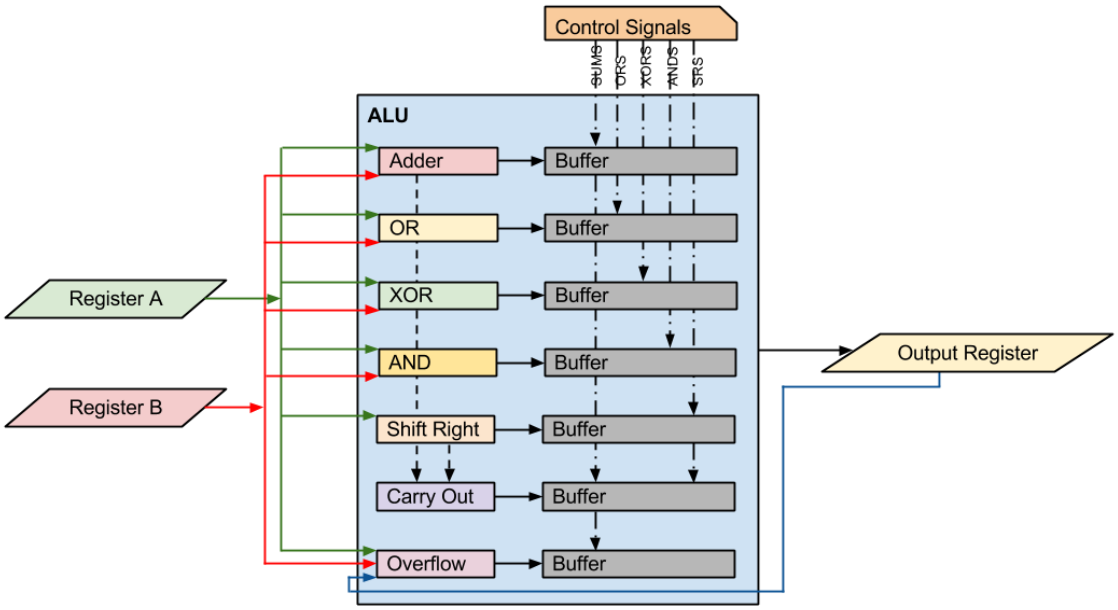
| **aaa** | **opcode** |
|---|---|
| 001 | BIT |
| 010 | JMP |
| 011 | JMP (abs) |
| 100 | STY |
| 101 | LDY |
| 110 | CPY |

| | |
|---|---|
| 111 | CPX |

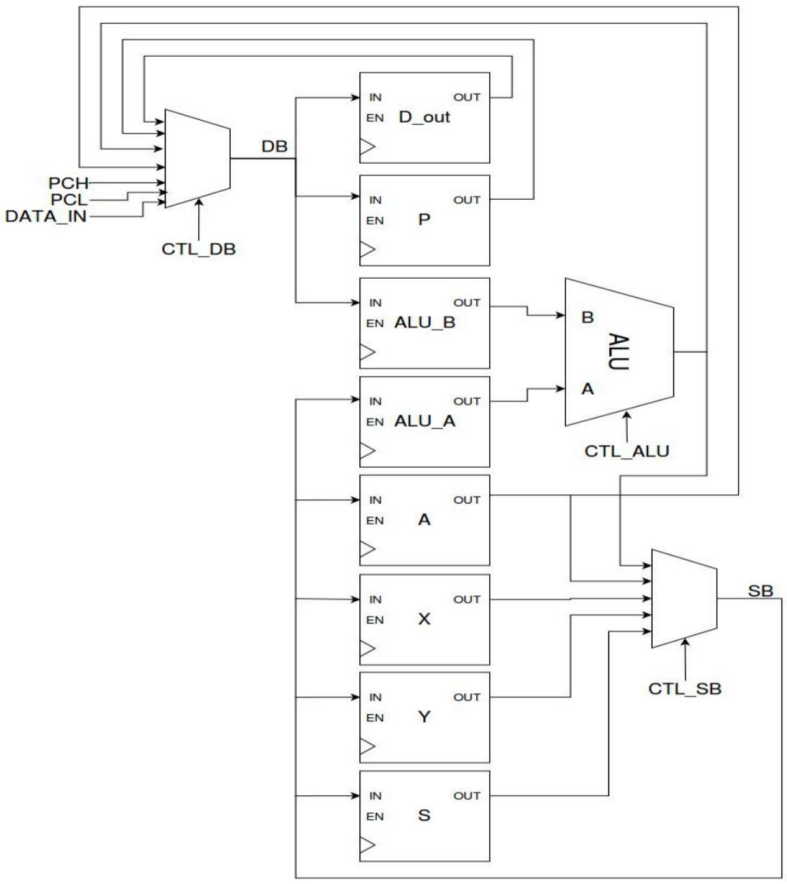
| **bbb** | **addressing mode** |
|---|---|
| 000 | # immediate |
| 001 | zero page |
| 011 | absolute |
| 101 | zero page, X |
| 111 | absolute, X |

## ALU

The 6502 ALU reads from two 8-bit input registers (A and B) and outputs its result on the Output register. What lies between them are the combinational logic blocks which perform the arithmetic and logical manipulation of the input data.



# Our Design



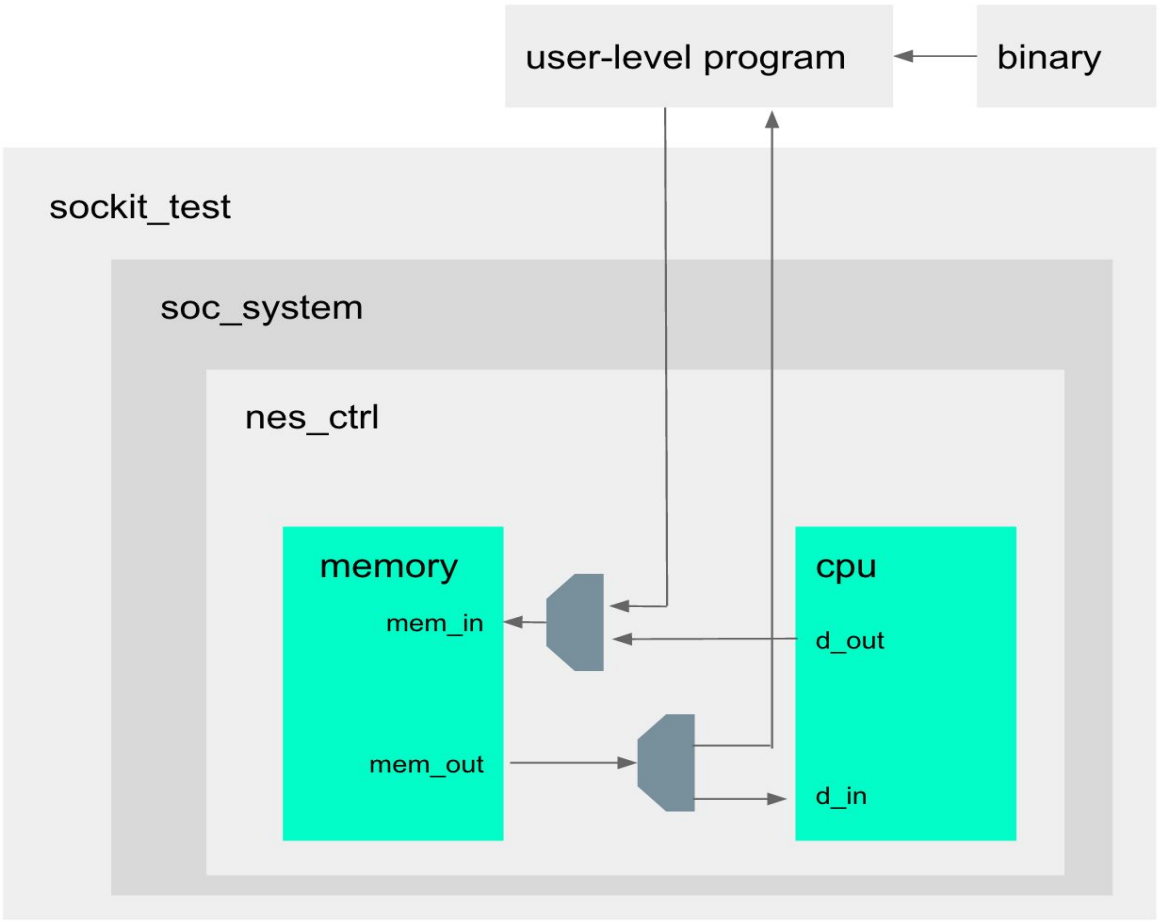

## Hardware

Our 6502 contains three main modules representing the CPU, ALU and memory. The top level module of the project also instantiates a few boilerplate modules adopted from an online tutorial which we followed in order to compile our RTL and program the FPGA.[1] Besides giving us a roadmap to follow when booting our Linux kernel onto the board and interfacing with the memory-mapped Avalon bus, given that the tutorial code is used for lighting up LEDs on the

[1] http://zhehaomao.com/

board, we were able to use this feature when debugging the hardware-software interface. It is exceedingly difficult to determine which of these two components has a problem when dealing with a hardware/software interface bug. Having access to visual cues on the board helped us on more than one occasion.

## Software



# Project Roadmap

## Objectives

- Initially set out to emulate the NES
- Implement the 6502 in SystemVerilog
- Synthesize the processor onto the FPGA
- Create software to interface with the processor
- Load programs into memory and read output of the processor in a user program

## Milestones

### Milestone 1

- Create simulations using Verilator for each submodule separately. Control logic partially complete.
- Verify that each submodule produces proper values for hard coded inputs

### Milestone 2

- Finish simulations for each submodule.
- Simulate timing between submodules... DEBUG!
- Begin synthesizing modules on the FPGA

### Milestone 3

- Finish synthesizing modules on the FPGA
- Write software interface to communicate between user and processor

### Final Presentation

- Processor successfully executes 6502 assembly
- Software interface allows user to place assembly code at proper memory address
- User can start and stop processor and view output

# Contributions

## Akira

- CPU design and implementation
- Testing suite
- Verilator integration

## Chaiwen

- ALU design
- Top-level module design and implementation

## Philip

- CPU design and implementation
- User-space program design and implementation

## Sean

- Memory module design
- Kernel module
- ALU implementation
- Top-level module design and implementation
- User-space program design

# References

The following resources proved to be invaluable in exploring the 6502 processor.

http://6502.org
https://github.com/Arlet/verilog-6502
http://www.llx.com/~nparker/a2/opcodes.html, http://www.6502.org/tutorials/6502opcodes.html
http://web.mit.edu/6.111/www/f2004/projects/dkm_report.pdf
http://nesdev.com/6502_cpu.txt
https://www.princeton.edu/~mae412/HANDOUTS/Datasheets/6502.pdf
http://rubbermallet.org/fake6502.c
http://users.telenet.be/kim1-6502/6502/hwman.html
https://zhehaomao.com/blog/fpga/2013/12/22/sockit-1.html

# Sample Test

```
$ make test_all
[pass] test_abs
[pass] test_absx
[pass] test_absy
[pass] test_adc
[fail] test_and
[fail] test_cmp
[fail] test_eor
[pass] test_indx
[pass] test_indy
[pass] test_ldx
[fail] test_logic
[pass] test_ora
[pass] test_sbc
[pass] test_stabs
[pass] test_sta
[pass] test_zp
[pass] test_zpx
[pass] test_zpy
[pass] test_stx
[pass] test_sty
```

# Sample 6502 Assembly Tests

```
LDA $0707
ADC $0304
LDA $aaff, X
ADC #$04
LDA $aaff, Y
ADC #$04
LDA #$02
ADC #$02
ADC #$04
ADC #$04
ADC #$04
LDA #$55
AND #$50
```

```
AND #$01
LDA #$03
CMP #$01
LDA #$0F
EOR #$0F
LDA ($aa,X)
ADC ($bb,X)
ADC #$04
ADC #$01
LDA ($ff),Y
ADC ($bb),Y
ADC #$04
ADC #$01
LDX #$05
LDX #$03
LDA #$03
AND #$01
STA $0200
LDA #$0
ORA #$01
STA $0201
LDA #$03
EOR #$01
STA $0202
LDA #$0F
ORA #$F0
ORA #$00
LDA #$0F
SBC #$03
SBC #$01
LDA #$05
STA $07aa
LDA #$03
LDA $07aa
ADC #$02

LDA #$05
STA $00aa
LDA #$03
LDA $00aa
ADC #$02
LDX #$08
STX $0506
```

```
ADC $00aa,X
ADC #$03
LDY #$08
STY $0506
ADC $00aa,Y
ADC #$03
LDA $00aa
ADC #$01
ADC $00aa
ADC #$01
LDA $ee, X
ADC $bb, X
ADC #$01
LDA $aa, Y
ADC $cc, Y
ADC #$01
```

# Source Code

```
parameter
   RESET_CPU = 8'd0,
    START_CPU = 8'd1,
    PAUSE_CPU = 8'd2,
    WRITE_MEM = 8'd3
;

module delay_ctrl (
     input logic clk,
     input logic faster,
     output logic slower,

     input logic reset,

     input logic read,
     output logic [15:0] readdata,

     input logic write,
     input logic [15:0] writedata,
     input logic [15:0] address
);
```

```
logic cpu_ready;
logic sync;
logic cpu_write, mem_write, cpu_reset;
logic [15:0] cpu_addr, mem_addr;
logic [7:0] cpu_in, cpu_out, mem_in, mem_out;
logic [7:0] nes_op;                          // our own NES opcodes

logic [15:0] tmp_data;

assign nes_op = writedata[15:8];

initial begin
    tmp_data = 0;
    slower <= 0;
end

always_ff @(posedge clk) begin
    if (write) begin
      slower <= 1;
    end else begin
      slower <= 0;
    end

    /*if (read) begin
      slower <= 1;
           readdata <= tmp_data;
    end else begin
      slower <= 0;
           readdata <= 7;
    end

    readdata <= tmp_data;*/
end

cpu c (
   .clk (clk),
   .reset (cpu_reset),
   .ready (cpu_ready),
   .irq (0),
   .nmi (0),
   .d_in (cpu_in),
   .write (cpu_write),
```

```
    .sync (sync),
    .d_out (cpu_out),
    .addr (cpu_addr)
);

memory mem (
    .clk (clk),
    .addr (mem_addr),
    .write (mem_write),
    .in (mem_in),
    .out (mem_out)
);

assign readdata = {8'd0, mem_out};
assign cpu_in = mem_out;

always_comb begin

    if (interface_on) begin
        mem_write = write;
        mem_in = writedata[7:0];
        mem_addr = address;
    end else begin
       mem_write = cpu_write;
       mem_addr = cpu_addr;
       mem_in = cpu_out;
    end

end

logic interface_on = 1;
always_ff @(posedge clk)
begin
    case (nes_op)
       WRITE_MEM: interface_on <= 1;
       RESET_CPU: interface_on <= 1;
       START_CPU: interface_on <= 0;
    endcase
end

always_comb begin

    if (interface_on) begin
```

```
       cpu_ready = 0;
       cpu_reset = 1;
   end else begin
       cpu_reset = 0;
       cpu_ready = 1;
    end
end
endmodule


module sockit_test (
      input CLOCK_50,
      input [3:0] KEY,
      output [3:0] LED,

     output [12:0] hps_memory_mem_a,
      output [2:0]  hps_memory_mem_ba,
      output     hps_memory_mem_ck,
      output     hps_memory_mem_ck_n,
      output     hps_memory_mem_cke,
      output     hps_memory_mem_cs_n,
      output     hps_memory_mem_ras_n,
      output     hps_memory_mem_cas_n,
      output     hps_memory_mem_we_n,
      output     hps_memory_mem_reset_n,
      inout  [7:0] hps_memory_mem_dq,
      inout         hps_memory_mem_dqs,
      inout         hps_memory_mem_dqs_n,
      output     hps_memory_mem_odt,
      output        hps_memory_mem_dm,
      input      hps_memory_oct_rzqin
);

// internal signals
wire [3:0] key_os;
wire [3:0] delay;
wire main_clk = CLOCK_50;

assign delay[1] = 0;
assign delay[2] = 0;
//wire test_out;

oneshot os (
```

```
      .clk (main_clk), // port mappings
      .edge_sig (KEY),
      .level_sig (key_os)
);


soc_system soc (
      .delay_ctrl_slower (delay[0]),
      .delay_ctrl_faster (key_os[1]),
      .memory_mem_a         (hps_memory_mem_a),
      .memory_mem_ba        (hps_memory_mem_ba),
      .memory_mem_ck        (hps_memory_mem_ck),
      .memory_mem_ck_n      (hps_memory_mem_ck_n),
      .memory_mem_cke       (hps_memory_mem_cke),
      .memory_mem_cs_n      (hps_memory_mem_cs_n),
      .memory_mem_ras_n     (hps_memory_mem_ras_n),
      .memory_mem_cas_n     (hps_memory_mem_cas_n),
      .memory_mem_we_n      (hps_memory_mem_we_n),
      .memory_mem_reset_n   (hps_memory_mem_reset_n),
      .memory_mem_dq        (hps_memory_mem_dq),
      .memory_mem_dqs       (hps_memory_mem_dqs),
      .memory_mem_dqs_n     (hps_memory_mem_dqs_n),
      .memory_mem_odt       (hps_memory_mem_odt),
      .memory_mem_dm        (hps_memory_mem_dm),
      .memory_oct_rzqin     (hps_memory_oct_rzqin),

      .clk_clk (main_clk),
      .reset_reset_n (!key_os[3])
);

blinker b (
      .clk (main_clk),
      .delay (delay),
      .led (LED),
      .reset (key_os[3]),
      .pause (key_os[2])
);
endmodule

parameter
  ALU_ADD = 0,
  ALU_AND = 1,
  ALU_OR = 2,
```

```
  ALU_EOR = 3,
  ALU_SR = 4,
  ALU_SUB = 5,
  ALU_CMP = 6;


/*
 * Opcodes {aaa}
 */

parameter
  ORA = 3'b000,
  AND = 3'b001,
  EOR = 3'b010,
  ADC = 3'b011,
  STA = 3'b100,
  LDA = 3'b101,
  CMP = 3'b110,
  SBC = 3'b111,

  ASL = 3'b000,
  ROL = 3'b001,
  LSR = 3'b010,
  ROR = 3'b011,
  STX = 3'b100,
  LDX = 3'b101,
  DEC = 3'b110,
  INC = 3'b111,

  BRK = 3'b000,
  BIT = 3'b001,
  JMP = 3'b010,
  STY = 3'b100,
  LDY = 3'b101,
  CPY = 3'b110,
  CPX = 3'b111;


module cpu (
           input      clk,
           input      reset,
           input      ready,
           input      irq,
```

```
        input      nmi,
        input [7:0]   d_in,
        output     write,
        output     sync,
        output [7:0]  d_out,
        output [15:0] addr
        );

/*
  * Instruction Fields
  */

logic [2:0]             aaa;
logic [2:0]             bbb;
logic [1:0]             cc;
logic [4:0]             opcode;


assign {aaa, bbb, cc} = IR;
assign opcode = {aaa, cc};
assign t1op = {d_in[7:5], d_in[1:0]};

/*
  * Arith control signal
  */

logic                   arith;
always_comb begin
  case (aaa)
  LDA: arith = 0;

  ORA: arith = 1;
  AND: arith = 1;
  EOR: arith = 1;
  ADC: arith = 1;
  SBC: arith = 1;
  CMP: arith = 1;
  default: arith = 0;
  endcase
  if (reset)
  arith = 0;
end
```

```
/*
   * Store control signal
   */


logic store;
always_comb begin
   case (aaa)
   STA: store = 1;
   STY: store = 1;
   STX: store = 1;
   default: store = 0;
   endcase
   if (reset)
   store = 0;
end

/*
   * Registers
   */

logic [7:0] A, // accumulator
             X, // X index
             Y, // Y index
             D_OUT, // data output
             IR, // instruction register
             P, // processor status
             SP, // stack pointer
             ADL, // address low
             ADH; // address high
logic [15:0] PC; // program counter

/*
   * Instruction Register
   */

always_ff @ (posedge clk)
   begin
   if (state == DECODE)
         IR <= d_in;

   if (reset)
         IR <= 0;
```

```
  end

enum {DST_A, DST_X, DST_Y} dst;
always_comb
  begin
  casex (IR)
       8'b101xxx01: dst = DST_A;
       8'b101xxx10: dst = DST_X;
       8'b101xxx00: dst = DST_Y;
       default: dst = DST_A;
  endcase
  end

/*
  * Accumulator
  */
// ORA, AND, EOR, ADC, SBC

always_ff @ (posedge clk)
  begin
  if (reset)
       A <= 0;
  else if (dst == DST_A)
       case (state)
       FETCH: A <= arith ? alu_out : d_in;
       default: A <= A;
       endcase;
  end

/*
  * X Index Register
  */

always_ff @ (posedge clk)
  begin
  if (reset)
       X <= 0;
  else if (dst == DST_X)
       case (state)
       FETCH: X <= arith ? alu_out : d_in;
       default: X <= X;
       endcase;
  end
```

```
/*
   * Y Index Register
   */

always_ff @ (posedge clk)
   begin
   if (reset)
         Y <= 0;
   else if (dst == DST_Y)
         case (state)
         FETCH: Y <= arith ? alu_out : d_in;
         default: Y <= Y;
         endcase;
   end

/*
   * Processor Status Register
   */

always_ff @ (posedge clk)
   begin
   if (reset)
         P <= 0;
   else
         case (state)
         ABSX1,
         ABSY1,
         INDY2,
         FETCH: P <= {sign, over, 3'b100, P[2], zero, cout};
         endcase;
   end

/*
   * Program Counter
   */

always_ff @ (posedge clk)
   begin
   case (state)
         ABS2,

         INDX1,
```

```
        INDX2,
        INDX3,
        INDX4,

        INDY1,
        INDY2,
        INDY3,
        INDY4,

        ABSY2,
        ABSY3,

        ABSX2,
        ABSX3,

        ZPX1,
        ZPX2,

        ZP1: PC <= PC;
        default: PC <= PC + 1;
  endcase
  if (reset)
        PC <= 0;
  end

/*
  * Address Low Register
  */

always_ff @ (posedge clk)
  begin
  case (state)
        ABS1: ADL <= d_in;

        INDX3: ADL <= d_in;

        INDY2: ADL <= alu_out;

        ZP1: ADL <= d_in;

        ABSX1: ADL <= alu_out;

        ABSY1: ADL <= alu_out;
```

```
        default: ADL <= ADL;
  endcase;
  if (reset)
        ADL <= 0;
  end

/*
  * Address High Register
  */

always_ff @ (posedge clk)
  begin
  case (state)
        ABS2: ADH <= d_in;

        ABSX2: ADH <= alu_out;

        INDY3: ADH <= alu_out;

        default: ADH <= ADH;
  endcase;
  if (reset)
        ADH <= 0;

  end

logic [7:0] BAL;
always_ff @ (posedge clk)
  begin
  case (state)
        INDX1: BAL <= alu_out;
        INDX2: BAL <= alu_out;
        INDX3: BAL <= alu_out;

        INDY1: BAL <= alu_out;

        ZPX1: BAL <= alu_out;
  endcase
  if (reset)
        BAL <= 0;
  end
```

```
logic [7:0] IAL;
always_ff @ (posedge clk)
  begin
  case (state)

        INDY1: IAL <= alu_out;
  endcase
  if (reset)
        IAL <= 0;

  end
/*
  * Address Output
  */

always_comb
  begin
  case (state)

        ABS2: addr = {d_in, ADL};

        INDX2: addr = {8'b0, BAL};
        INDX3: addr = {8'b0, BAL};
        INDX4: addr = {d_in, ADL};

        INDY1: addr = {8'b0, d_in}; // IAL
        INDY2: addr = {8'b0, BAL}; // IAL + 1
        INDY3: addr = {d_in, ADL}; // BAH, BAL + Y
        INDY4: addr = {ADH, ADL}; // BAH + C, BAL + Y

        ABSX2: addr = {d_in, ADL};
        ABSX3: addr = {ADH, ADL};

        ABSY2: addr = {d_in, ADL};
        ABSY3: addr = {ADH, ADL};

        ZP1: addr = {8'b0, d_in};

        ZPX1: addr = {8'b0, d_in};
        ZPX2: addr = {8'b0, BAL};

        default: addr = PC;
```

```
   endcase;

   if (reset)
        addr = 0;
   end


/*
   * Controller FSM
   */

enum {
   DECODE, // T0
   FETCH, // TX (final state of instruction)

   ABS1,
   ABS2,

   ABSX1,
   ABSX2,
   ABSX3,

   ABSY1,
   ABSY2,
   ABSY3,

   INDX1,
   INDX2,
   INDX3,
   INDX4,

   INDY1,
   INDY2,
   INDY3,
   INDY4,


   ZPX1,
   ZPX2,

   ZP1

   } state;
```

```
initial state = FETCH;

always_ff @ (posedge clk)
  begin
  if (ready) begin
  case (state)
       FETCH: state <= DECODE;
       DECODE: begin
       casex (d_in)
            8'bxxx01101,
            8'bxxx01110,
            8'bxxx01100: state <= ABS1; // Absolute
            8'bxxx00101: state <= ZP1; // Zero Page
            8'bxxx11101: state <= ABSX1; // Absolute X
            8'bxxx11001: state <= ABSY1; // Absolute Y
            8'bxxx00001: state <= INDX1; // Indirect, X
            8'bxxx10001: state <= INDY1; // Indirect, Y
            8'bxxx10101: state <= ZPX1; // Zero Page X

            default: state <= FETCH; // Immediate
       endcase
       end

       INDX1: state <= INDX2;
       INDX2: state <= INDX3;
       INDX3: state <= INDX4;
       INDX4: state <= FETCH;

       INDY1: state <= INDY2;
       INDY2: state <= INDY3;
       INDY3: state <= P[0] ? INDY4 : FETCH;
       INDY4: state <= FETCH;

       ABS1: state <= ABS2;
       ABS2: state <= FETCH;

       ABSX1: state <= ABSX2;
       ABSX2: state <= P[0] ? ABSX3 : FETCH;
       ABSX3: state <= FETCH;


       ABSY1: state <= ABSY2;
```

```
            ABSY2: state <= P[0] ? ABSY3 : FETCH;
            ABSY3: state <= FETCH;

            ZP1:  state <= FETCH;

            ZPX1: state <= ZPX2;
            ZPX2: state <= FETCH;

            default: state <= FETCH;
      endcase;
      end
      end

`ifdef DEBUG
   always_ff @ (posedge clk)
     begin
     $display("addr:%x d_in:%x d_out:%x write:%x A:%x X:%x Y:%x a:%x
b:%x: out:%x P:%x BAL:%x ADL:%x ADH:%x",
                addr, d_in, d_out, write, A, X, Y, alu_a, alu_b,
alu_out, P, BAL, ADL, ADH);
     end
`endif

   /*
     * alu_a, alu_b control
     */

   always_comb
     begin
     case (state)

            INDX1: alu_a = X;
            INDX2: alu_a = BAL;

            INDY1: alu_a = 1;
            INDY2: alu_a = Y;
            INDY3: alu_a = 0;

            ABSX1: alu_a = X;
            ABSX3: alu_a = ADH;

            ABSY1: alu_a = Y;
            ABSY3: alu_a = ADH;
```

```
        ZPX1:  alu_a = X;

        FETCH: alu_a = arith ? A : d_in;

        default: alu_a = 0;
  endcase;

  if (reset)
        alu_a = 0;
  end

always_comb
  begin
  case (state)
        INDX1: alu_b = d_in; // ADL
        INDX2: alu_b = 1;

        INDY1: alu_b = d_in; // BAL
        INDY2: alu_b = d_in; // ADL
        INDY3: alu_b = d_in; // BAH


        ABSX1: alu_b = d_in; // ADL
        ABSX3: alu_b = P[0];

        ABSY1: alu_b = d_in; // ADL
        ABSY3: alu_b = P[0];

        ZPX1: alu_b = d_in;

        FETCH: alu_b = d_in;
        default: alu_b = d_in;
  endcase;
  if (reset)
  alu_b = 0;

  end


/*
  * d_out
  */
```

```
always_comb
  begin
  case (state)
       ZP1: begin
            if (store) begin
            case (aaa)
                 STA: d_out = A;
                 STX: d_out = X;
                 STY: d_out = Y;
            endcase
            end
       end
       ABS2: begin
            if (store) begin
            case (aaa)
                 STA: d_out = A;
                 STX: d_out = X;
                 STY: d_out = Y;
            endcase
            end
       end
       default: d_out = 0;
  endcase
  if (reset)
       d_out = 0;
  end



/*
  * write
  */
always_comb
  begin
  case (state)
       ZP1: write = store ? 1 : 0;
       ABS2: write = store ? 1 : 0;
       default: write = 0;
  endcase
  if (reset)
       write = 0;
  end
```

```
/*
   * ALU carry in
   */

assign cin = reset ? 0 : P[0];


/*
   * ALU
   */

logic [7:0] alu_a, alu_b, alu_out;
logic [4:0] alu_mode;
logic        cin, cout, over, zero, sign;
alu ALU(
        .alu_a(alu_a),
        .alu_b(alu_b),
        .mode(alu_mode),
        .carry_in(cin),
        .alu_out(alu_out),
        .carry_out(cout),
        .overflow(over),
        .zero(over),
        .sign(sign)
        );

always_comb
  begin
  casex (IR)
        8'b000xxx01: alu_mode = ALU_OR;
        8'b001xxx01: alu_mode = ALU_AND;
        8'b010xxx01: alu_mode = ALU_EOR;
        8'b011xxx01: alu_mode = ALU_ADD;
        8'b110xxx01: alu_mode = ALU_CMP;
        8'b111xxx01: alu_mode = ALU_SUB;
        8'b010xxx10: alu_mode = ALU_SR;

        default: alu_mode = ALU_ADD;
  endcase
  end
```

```
   /*
     * SYNC Signal
     */

   assign sync = (state == DECODE);

endmodule; // cpu

module alu (input [7:0] alu_a,
           input [7:0] alu_b,
                 input [4:0] mode,
                 input carry_in,
                 output [7:0] alu_out,
                 output carry_out,
                 output overflow,
                 output zero,
                 output sign);

    logic [8:0] tmp_out; //9 bit add for easy overflow/carry checks

    // for right shift, alu_b will be zero
   always_comb
     begin
     case (mode)
          ALU_CMP: alu_out = alu_a;
          default: alu_out = tmp_out[7:0];
     endcase
     end
    assign   overflow = alu_a[7] ^ alu_b[7] ^ tmp_out[7];
    assign    carry_out = (mode == ALU_SUB) ? ~carry : carry;
    logic carry;
    assign  carry = tmp_out[8];
    assign  sign = alu_out[7];
    assign    zero = tmp_out == 0;
    logic borrow;
    assign borrow = ~carry_in;

    always_comb begin
      case (mode)
           ALU_ADD: begin tmp_out = alu_a + alu_b + carry_in; end
        ALU_SUB: begin tmp_out = alu_a - alu_b - borrow; end
           ALU_AND: begin tmp_out = alu_a & alu_b; end
           ALU_OR : tmp_out = alu_a | alu_b;
```

```
            ALU_EOR: tmp_out = alu_a ^ alu_b;
            ALU_SR : begin tmp_out = {alu_a[0], carry_in,
alu_a[7:1]}; end //ASL,
            ALU_CMP: begin tmp_out = alu_a - alu_b; end

            default begin tmp_out = alu_a; end
       endcase

    end
endmodule

module memory(
   input logic [15:0] addr,
   input logic [7:0] in,
   input logic write,
   input logic clk,
   output logic [7:0] out);

logic [7:0] mem [65535:0];   // 65536-size array of 8-bit elements

always_ff @(posedge clk) begin
   if (write)
     mem[addr] <= in;
   else
     out <= mem[addr];
end

/* write.c */
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>

#define PAGE_SIZE 4096
#define LWHPS2FPGA_BRIDGE_BASE 0xff200000
#define NES_OFFSET 0x0
#define MEMSIZE 65536
```

```
typedef struct {
  unsigned char nes_op;
  unsigned char nes_in;
  unsigned char nes_out;
  unsigned short address;
} nes_args;

volatile unsigned char *nes_mem;
void *bridge_map;
int nes_fd;

void print_state(nes_args *nes)
{
     printf("current state: \n");
     printf( "nes_op: %x\n", nes->nes_op);
     printf( "nes_in: %x\n", nes->nes_in);
     printf("nes_out: %x\n", nes->nes_out);
     printf("address: %x\n", nes->address);
}

int main(int argc, char *argv[])
{
    int mem_fd;
    int ret = EXIT_FAILURE;
    off_t nes_base = LWHPS2FPGA_BRIDGE_BASE;
   char memory[MEMSIZE];
   memset((char *)memory, 0, sizeof(memory));

    if (argc < 2) {
       fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
       exit(EXIT_FAILURE);
    }

     /* Read assembled binary into simulated memory */
     char *filename = argv[1];
     FILE *binary = fopen(filename, "r");
     if (binary == NULL) {
     perror(filename);
     return 2;
     }
     size_t len = fread(memory, 1, MEMSIZE, binary);
```

```
 printf("userspace NES program started, read %d bytes.\n\n", len);

 /* open the memory device file */
 // char *mem_file = "/sys/bus/platform/devices/nes/nes";
char *mem_file = "/dev/mem";
 mem_fd = open(mem_file, O_RDWR|O_SYNC);
 if (mem_fd < 0) {
   perror("open");
   exit(EXIT_FAILURE);
 }

 printf("open done: %s\n", mem_file);

 /* map the LWHPS2FPGA bridge into process memory */
 bridge_map = mmap(NULL, PAGE_SIZE, PROT_WRITE, MAP_SHARED,
              mem_fd, nes_base);
 if (bridge_map == MAP_FAILED) {
   perror("mmap");
   goto cleanup;
 }

 printf("mmap done\n");

 printf("loading program into memory\n");

  /* get the delay_ctrl peripheral's base address */
 nes_mem = (unsigned char *) (bridge_map + NES_OFFSET);
   printf("passed nes_mem\n");

int x = 0;
while (x < len) {
   printf("%d: writing %x to memory\n", x, memory[x]);
   nes_mem[2 * x] = memory[x];
   nes_mem[1] = (char)3; //CPU_WRITE
   x++;
 }


 printf("munmap\n");
 if (munmap(bridge_map, PAGE_SIZE) < 0) {
   perror("munmap");
   goto cleanup;
 }
```

```
    ret = 0;

cleanup:
    fclose(binary);
    return ret;
}

/* read.c */
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>

#define PAGE_SIZE 4096
#define LWHPS2FPGA_BRIDGE_BASE 0xff200000
#define NES_OFFSET 0x0
#define MEMSIZE 65536

typedef struct {
  unsigned char nes_op;
  unsigned char nes_in;
  unsigned char nes_out;
  unsigned short address;
} nes_args;

volatile unsigned char *nes_mem;
void *bridge_map;
int nes_fd;

void print_state(nes_args *nes)
{
     printf("current state: \n");
     printf( "nes_op: %x\n", nes->nes_op);
     printf( "nes_in: %x\n", nes->nes_in);
     printf("nes_out: %x\n", nes->nes_out);
     printf("address: %x\n", nes->address);
}
```

```
int main(int argc, char *argv[])
{
    int mem_fd;
    int ret = EXIT_FAILURE;
    off_t nes_base = LWHPS2FPGA_BRIDGE_BASE;


    /* open the memory device file */
    // char *mem_file = "/sys/bus/platform/devices/nes/nes";
   char *mem_file = "/dev/mem";
    mem_fd = open(mem_file, O_RDWR|O_SYNC);
    if (mem_fd < 0) {
      perror("open");
      exit(EXIT_FAILURE);
    }

    printf("open done: %s\n", mem_file);

    /* map the LWHPS2FPGA bridge into process memory */
    bridge_map = mmap(NULL, PAGE_SIZE, PROT_WRITE, MAP_SHARED,
                 mem_fd, nes_base);
    if (bridge_map == MAP_FAILED) {
      perror("mmap");
      goto cleanup;
    }

    printf("mmap done\n");


     /* get the delay_ctrl peripheral's base address */
    nes_mem = (unsigned char *) (bridge_map + NES_OFFSET);
      printf("passed nes_mem\n");
   int x = 0;

      while (x < PAGE_SIZE) {
      printf("read %x\n", nes_mem[2 * x]);
      x++;
    }

    printf("munmap\n");
    if (munmap(bridge_map, PAGE_SIZE) < 0) {
      perror("munmap");
```

```
      goto cleanup;
    }

    ret = 0;

cleanup:
    return ret;
}

/* start.c */
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>

#define PAGE_SIZE 4096
#define LWHPS2FPGA_BRIDGE_BASE 0xff200000
#define NES_OFFSET 0x0
#define MEMSIZE 65536

typedef struct {
  unsigned char nes_op;
  unsigned char nes_in;
  unsigned char nes_out;
  unsigned short address;
} nes_args;

volatile unsigned char *nes_mem;
void *bridge_map;
int nes_fd;

void print_state(nes_args *nes)
{
     printf("current state: \n");
     printf( "nes_op: %x\n", nes->nes_op);
     printf( "nes_in: %x\n", nes->nes_in);
     printf("nes_out: %x\n", nes->nes_out);
     printf("address: %x\n", nes->address);
```

```
}

int main(int argc, char *argv[])
{
    int mem_fd;
    int ret = EXIT_FAILURE;
    off_t nes_base = LWHPS2FPGA_BRIDGE_BASE;
   char memory[MEMSIZE];
   memset((char *)memory, 0, sizeof(memory));

    /* open the memory device file */
    // char *mem_file = "/sys/bus/platform/devices/nes/nes";
   char *mem_file = "/dev/mem";
    mem_fd = open(mem_file, O_RDWR|O_SYNC);
    if (mem_fd < 0) {
      perror("open");
      exit(EXIT_FAILURE);
    }

    printf("open done: %s\n", mem_file);

    /* map the LWHPS2FPGA bridge into process memory */
    bridge_map = mmap(NULL, PAGE_SIZE, PROT_WRITE, MAP_SHARED,
                  mem_fd, nes_base);
    if (bridge_map == MAP_FAILED) {
      perror("mmap");
      goto cleanup;
    }

    printf("mmap done\n");

    printf("loading program into memory\n");

     /* get the delay_ctrl peripheral's base address */
    nes_mem = (unsigned char *) (bridge_map + NES_OFFSET);
      printf("passed nes_mem\n");

      nes_mem[1] = (char)1; //CPU_START


    printf("munmap\n");
    if (munmap(bridge_map, PAGE_SIZE) < 0) {
      perror("munmap");
```

```
        goto cleanup;
    }

    ret = 0;

cleanup:
    return ret;
}
```