Embedded System Design
Summer 2005, NCTU **Course Project**

The Electronic chess game

By

Sheng-Kuo Lu (9395503)
Chun Huang (9395552)
Jang-Yuan Kao (9395556)

Abstract

We have designed a chess game that includes move validation, CHECK checking, pawn promotion, and undo. It also has a logging system which shows and can saves what to play and moves.

Table of Contents

1. Overview and Design

The overall design is to create an electronic check game. User can press the bottom on the keyboard in order to control which one should be moved on the screen, the output signal from the keyboard will be decoded by FPGA, the movement shall displayed on the screen at the same time.

The architecture of synthesis FPGA included main controller / color controller / shift register / board RAM / piece ROM … etc.

2. What we intended to do

What we want to do is to implement the chess games should have move validation & CHECK checking, pawn promotion, and undo. It also has a logging system which shows and can saves what to play and moves.

3. What we actually did

What we actually did was the chess games has move validation & CHECK checking, pawn promotion.

4. Design Components

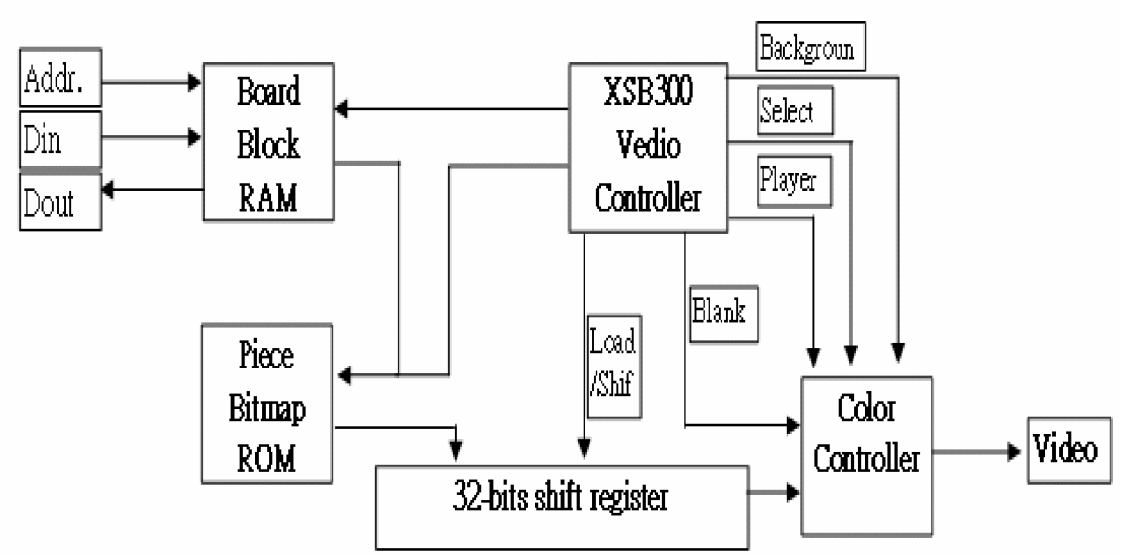


Diagram 1: Overall block diagram of the final project

4.1 Hardware Description

The block diagram above shows the components available in the final project. Of those available, we used the monitor as the output video peripheral.

4.2 VGA controller

This peripheral performs important functions. First, it controls the VGA by generating the necessary timing signals (horizontal sync, vertical sync, and blanking, which blanks the video output during horizontal and vertical refresh), and memory addresses for each pixel, and eventually formats. Second, The glue logic in the main module, opb_xsb300.vhd, which generates the signals and the off-chip drivers for these signals are in pad_io.vhd. It arbitrates access to this memory between the video controller and the processor. In each cycle, the processor, video, or both may want access to the memory. Since the video absolutely needs the memory when it asks for it the memory controller gives priority to the video system.

5. Who did what

Sheng-Kuo Lu (9395503)
Original design, keyboard control improvement, fonts, error debugging, coding of checking rule.

Chun Huang (9395552)
Graphical display, fonts, writing of final project report.

Jang-Yuan Kao (9395556)
Graphical display, displaying design layouts, coding of checking rule.

6. VHDL Code

```
----------------------------------------------------------------------------
--
-- Chess Board VGA controller for the Digilent Spartan 3 starter board
--
-- Uses an OPB interface, e.g., for use with the Microblaze soft core
--
-- Sheng-Kuo Lu
-- sklu.iic93g@nctu.edu.tw
--
-- Update from the sample Text Mode VGA controller
-- Stephen A. Edwards
-- sedwards@cs.columbia.edu
-- below link for further information about chess game.
-- http://hk.geocities.com/goodchessclub/main.html
----------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity opb_s3board_vga is

  generic (
    C_OPB_AWIDTH : integer                      := 32;
    C_OPB_DWIDTH : integer                      := 32;
    C_BASEADDR     : std_logic_vector(31 downto 0) := X"FEFF1000";
```

```
    C_HIGHADDR      : std_logic_vector(31 downto 0) := X"FEFF1FFF");

  port (
    OPB_Clk             : in std_logic;
    OPB_Rst             : in std_logic;

    -- OPB signals
    OPB_ABus            : in std_logic_vector (31 downto 0);
    OPB_BE              : in std_logic_vector (3 downto 0);
    OPB_DBus            : in std_logic_vector (31 downto 0);
    OPB_RNW               : in std_logic;
    OPB_select          : in std_logic;
    OPB_seqAddr         : in std_logic;

    VGA_DBus            : out std_logic_vector (31 downto 0);
    VGA_errAck          : out std_logic;
    VGA_retry           : out std_logic;
    VGA_toutSup         : out std_logic;
    VGA_xferAck         : out std_logic;

    Pixel_Clock_2x : in std_logic;

    -- Video signals
    VIDOUT_CLK          : out std_logic;
    VIDOUT_RED          : out std_logic;
    VIDOUT_GREEN        : out std_logic;
    VIDOUT_BLUE         : out std_logic;
    VIDOUT_HSYNC : out std_logic;
    VIDOUT_VSYNC : out std_logic);

end opb_s3board_vga;

architecture Behavioral of opb_s3board_vga is

  constant BASEADDR : std_logic_vector(31 downto 0) := X"FEFF1000";

  -- Video parameters

  constant HTOTAL : integer := 800;
  constant HSYNC : integer := 96;
  constant HBACK_PORCH : integer := 48;
  constant HACTIVE : integer := 640;
  constant HFRONT_PORCH : integer := 16;

  constant VTOTAL : integer := 525;
  constant VSYNC : integer := 2;
  constant VBACK_PORCH : integer := 33;
  constant VACTIVE : integer := 480;
  constant VFRONT_PORCH : integer := 10;

  -- 2048 X 9 dual-ported Xilinx block RAM
  component RAMB16_S9_S9
    port (
      DOA     : out std_logic_vector (7 downto 0);
      DOPA0   : out std_logic;
      ADDRA : in std_logic_vector (10 downto 0);
      CLKA    : in std_logic;
      DIA     : in std_logic_vector (7 downto 0);
      DIPA0   : in std_logic;
      ENA     : in std_logic;
      SSRA    : in std_logic;
      WEA     : in std_logic;
      DOB     : out std_logic_vector (7 downto 0);
      DOPB0   : out std_logic;
      ADDRB : in std_logic_vector (10 downto 0);
      CLKB    : in std_logic;
      DIB     : in std_logic_vector (7 downto 0);
      DIPB0   : in std_logic;
      ENB     : in std_logic;
      SSRB    : in std_logic;
```

```
      WEB     : in std_logic);
end component;

-- 512 X 36 dual-ported Xilinx block RAM
component RAMB16_S36_S36
   port (
      DOA     : out std_logic_vector (31 downto 0);
      DOPA    : out std_logic_vector(3 downto 0);
      ADDRA : in std_logic_vector (8 downto 0);
      CLKA    : in std_logic;
      DIA     : in std_logic_vector (31 downto 0);
      DIPA    : in std_logic_vector(3 downto 0);
      ENA     : in std_logic;
      SSRA    : in std_logic;
      WEA     : in std_logic;
      DOB     : out std_logic_vector (31 downto 0);
      DOPB    : out std_logic_vector(3 downto 0);
      ADDRB : in std_logic_vector (8 downto 0);
      CLKB    : in std_logic;
      DIB     : in std_logic_vector (31 downto 0);
      DIPB    : in std_logic_vector(3 downto 0);
      ENB     : in std_logic;
      SSRB    : in std_logic;
      WEB     : in std_logic);
end component;

-- Signals latched from the OPB
signal ABus : std_logic_vector (31 downto 0);
signal DBus : std_logic_vector (31 downto 0);
signal RNW    : std_logic;
signal select_delayed : std_logic;

-- Latched output signals for the OPB
signal DBus_out : std_logic_vector (31 downto 0);

-- Signals for the OPB-mapped RAM
signal ChipSelect : std_logic;                    -- Address decode
signal MemCycle1, MemCycle2 : std_logic;    -- State bits
signal RamPageAddress : std_logic;
signal RamSelect : std_logic_vector (1 downto 0);
signal RST, WE : std_logic_vector(1 downto 0);
signal DOUT0 : std_logic_vector(7 downto 0);
signal DOUT1 : std_logic_vector(31 downto 0);
signal ReadData : std_logic_vector(31 downto 0);

-- Signals for the video controller
signal Pixel_Clock : std_logic;            -- 25 MHz clock divided from 50 MHz
signal LoadNShift : std_logic;             -- Shift register control
signal PieceData : std_logic_vector(31 downto 0);     -- Input to shift register
signal ShiftData : std_logic_vector(31 downto 0);    -- Shift register data
signal VideoData : std_logic;              -- Serial out ANDed with blanking

signal Hcount : std_logic_vector(9 downto 0);    -- Horizontal position (0-800)
signal Vcount : std_logic_vector(9 downto 0);    -- Vertical position (0-524)
signal HBLANK_N, VBLANK_N : std_logic;     -- Blanking signals

signal PieceLoad, LoadAddr : std_logic;    -- Font/Character RAM read triggers
signal PieceAddr : std_logic_vector(8 downto 0);
signal BlockAddr : std_logic_vector(10 downto 0);
signal BlockColumn : std_logic_vector(9 downto 0);
signal BlockRow : std_logic_vector(9 downto 0);
signal Column : std_logic_vector(4 downto 0); -- 0-19
signal Row : std_logic_vector(3 downto 0);      -- 0-14
signal EndOfLine, EndOfField : std_logic;
signal Player, Cursor : std_logic;
signal PlayerData, CursorData : std_logic;
signal Background, BackgroundData : std_logic_vector(1 downto 0);

signal DOUTB0 : std_logic_vector(7 downto 0);
signal DOUTB1 : std_logic_vector(31 downto 0);
```

```
signal DOP0A, DOP0B : std_logic;
signal DOP1A, DOP1B : std_logic_vector(3 downto 0);

attribute INIT_00 : string;
attribute INIT_01 : string;
attribute INIT_02 : string;
attribute INIT_03 : string;
attribute INIT_04 : string;
attribute INIT_05 : string;
attribute INIT_06 : string;
attribute INIT_07 : string;
attribute INIT_08 : string;
attribute INIT_09 : string;
attribute INIT_0a : string;
attribute INIT_0b : string;
attribute INIT_0c : string;
attribute INIT_0d : string;
attribute INIT_0e : string;
attribute INIT_0f : string;
attribute INIT_10 : string;
attribute INIT_11 : string;
attribute INIT_12 : string;
attribute INIT_13 : string;
attribute INIT_14 : string;
attribute INIT_15 : string;
attribute INIT_16 : string;
attribute INIT_17 : string;
attribute INIT_18 : string;
attribute INIT_19 : string;
attribute INIT_1a : string;
attribute INIT_1b : string;
attribute INIT_1c : string;
attribute INIT_1d : string;
attribute INIT_1e : string;
attribute INIT_1f : string;
attribute INIT_20 : string;
attribute INIT_21 : string;
attribute INIT_22 : string;
attribute INIT_23 : string;
attribute INIT_24 : string;
attribute INIT_25 : string;
attribute INIT_26 : string;
attribute INIT_27 : string;
attribute INIT_28 : string;
attribute INIT_29 : string;
attribute INIT_2a : string;
attribute INIT_2b : string;
attribute INIT_2c : string;
attribute INIT_2d : string;
attribute INIT_2e : string;
attribute INIT_2f : string;
attribute INIT_30 : string;
attribute INIT_31 : string;
attribute INIT_32 : string;
attribute INIT_33 : string;
attribute INIT_34 : string;
attribute INIT_35 : string;
attribute INIT_36 : string;
attribute INIT_37 : string;
attribute INIT_38 : string;
attribute INIT_39 : string;
attribute INIT_3a : string;
attribute INIT_3b : string;
attribute INIT_3c : string;
attribute INIT_3d : string;
attribute INIT_3e : string;
attribute INIT_3f : string;

-- Initial chess piece data
attribute INIT_00 of Board_Block : label is
```

```
  "0000000000000000000000000000000000000000000000000000000000000007";
attribute INIT_01 of Board_Block : label is
  "0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_02 of Board_Block : label is
  "0000050403010203040500000000000000000000000000000000000000000000";
attribute INIT_03 of Board_Block : label is
  "4000000000000000000000000000006060606060606060000000000000000000";
attribute INIT_04 of Board_Block : label is
  "0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_05 of Board_Block : label is
  "0000000000000000000000000000000000000000000000000000000000000000";
attribute INIT_06 of Board_Block : label is
  "0000000000000000000026262626262626260000000000000000000000000000";
attribute INIT_07 of Board_Block : label is
  "0000000000000000000000000000000000000000000025242321222324250000";


-- Standard 8x16 font taken from the Linux console font file "lat0-16.psfu"

attribute INIT_00 of Piece_Bitmap : label is
  "0000000000000000000000000000000000000000000000000000000000000000";   -- empty piece
attribute INIT_01 of Piece_Bitmap : label is
  "0000000000000000000000000000000000000000000000000000000000000000";   -- empty piece
attribute INIT_02 of Piece_Bitmap : label is
  "0000000000000000000000000000000000000000000000000000000000000000";   -- empty piece
attribute INIT_03 of Piece_Bitmap : label is
  "0000000000000000000000000000000000000000000000000000000000000000";   -- empty piece
attribute INIT_04 of Piece_Bitmap : label is
  "0000800000008000000080000003E0000000800000008000000000000000000";   -- the KING
attribute INIT_05 of Piece_Bitmap : label is
  "0BFD5FE80BFD5FE803FB6FE805E6B3D0061EBC3001F2A7C0000220000001C000";   -- the KING
attribute INIT_06 of Piece_Bitmap : label is
  "004FF9000060030000FFFF80008FF88001A082C0037EBF6006FEBFA005FEBFD0";   -- the KING
attribute INIT_07 of Piece_Bitmap : label is
  "00000000000000000000000000FF800007FFF0000FFFF800087F08000700700";          -- the KING
attribute INIT_08 of Piece_Bitmap : label is
  "3C6186381861861800F38E0000F3EE000061C60000038000000100000000000";   -- the QUEEN
attribute INIT_09 of Piece_Bitmap : label is
  "07FFFFE007FFFEE00679FEF00F71CE300C73CE301E73CE381863CE183C638E38";   -- the QUEEN
attribute INIT_0a of Piece_Bitmap : label is
  "009FF80000C0038001FFFF800180018001FFFF8003FFFF8007FFFF8007FFFFE0";   -- the QUEEN
attribute INIT_0b of Piece_Bitmap : label is
  "00000000000000000000000000000000FFFE0001FFFF00018FF00001E00600";          -- the QUEEN
attribute INIT_0c of Piece_Bitmap : label is
  "0001C0000003E0000003E0000003E0000001C000000000000000000000000000";-- the BUSHOP
attribute INIT_0d of Piece_Bitmap : label is
  "007F7E00007F7F00007C1F00007F7F00003F7E00001FFE00001FFC000007F000";       -- the BUSHOP
attribute INIT_0e of Piece_Bitmap : label is
  "001FFC000017F40000080800000FF800000FF80000180C00003FFE00003FFE00";       -- the BUSHOP
attribute INIT_0f of Piece_Bitmap : label is
  "0000000000000000040000 1007FE3FF00FFF7FF80FFFFFF80006B0000007F000";       -- the BUSHOP
attribute INIT_10 of Piece_Bitmap : label is
  "007FF800007FE0000077000000C3000000020000000000000000000000000000"; -- the KNIGHT
attribute INIT_11 of Piece_Bitmap : label is
  "03FF7F4003FFBE4003FFBEC003FFBEC001BFFC8001DFF30000FFE60000E78C00";          -- the KNIGHT
attribute INIT_12 of Piece_Bitmap : label is
  "0183FF980F83FF980EC1FF901FE1FFB01BF8FF300FFEFF300FFFFF3007FFFF20";-- the KNIGHT
attribute INIT_13 of Piece_Bitmap : label is
  "0000000000000000001FFFF8001FFFF8001FFF98001FFF98000FFF98000FFF98";       -- the KNIGHT
attribute INIT_14 of Piece_Bitmap : label is
  "01FFFF0001FFFF0001E7CF0001E7CF0000000000000000000000000000000000";       -- the ROOK
attribute INIT_15 of Piece_Bitmap : label is
  "003FF800003FF800003FF800003FF80000600C0000FFFE000180030001FFFF00";       -- the ROOK
attribute INIT_16 of Piece_Bitmap : label is
  "01FFFF000180030000FFFE0000600C00003FF800003FF800003FF800003FF800";       -- the ROOK
attribute INIT_17 of Piece_Bitmap : label is
  "00000000000000000000000007FFFFC007FFFFC007FFFFC0060000C001FFFF00";       -- the ROOK
attribute INIT_18 of Piece_Bitmap : label is
  "0007F0000003F0000003E0000001C000000000000000000000000000000000000"; -- the PAWN
attribute INIT_19 of Piece_Bitmap : label is
```

```
  "000FF800000FFC00001FFC00001FFC00001FFC00000FF8000007F0000003E000";     -- the PAWN
attribute INIT_1a of Piece_Bitmap : label is
  "00FFFF8000FFFF8000FFFF80007FFF80007FFF00003FFE00001FFC000007F000";     -- the PAWN
attribute INIT_1b of Piece_Bitmap : label is
  "00000000000000000000000000000000000000000000FFFF8000FFFF8000FFFF80";         -- the PAWN
attribute INIT_1c of Piece_Bitmap : label is
  "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";          -- full foreground
attribute INIT_1d of Piece_Bitmap : label is
  "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";          -- full foreground
attribute INIT_1e of Piece_Bitmap : label is
  "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";          -- full foreground
attribute INIT_1f of Piece_Bitmap : label is
  "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";          -- full foreground
attribute INIT_20 of Piece_Bitmap : label is
  "F000000FF000000FF000000FF000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF";         -- selected empty piece
attribute INIT_21 of Piece_Bitmap : label is
  "F000000FF000000FF000000FF000000FF000000FF000000FF000000F";       -- selected empty piece
attribute INIT_22 of Piece_Bitmap : label is
  "F000000FF000000FF000000FF000000FF000000FF000000FF000000F";       -- selected empty piece
attribute INIT_23 of Piece_Bitmap : label is
  "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF000000FF000000FF000000FF000000F";         -- selected empty piece
attribute INIT_24 of Piece_Bitmap : label is
  "800380018001000180010001800100018007C00180010001800100010FFFFFFF";          -- selected KING
attribute INIT_25 of Piece_Bitmap : label is
  "880100219000001190028019900 6C001980C6031843C784185E44701800C6001";          -- selected KING
attribute INIT_26 of Piece_Bitmap : label is
  "809FF20180E0060180BFFB018180030181C00701833FF981860100C184010061";       -- selected KING
attribute INIT_27 of Piece_Bitmap : label is
  "FFFFFFFF8000000180000001800000001800FE00180F01E018180030181E00F01";       -- selected KING
attribute INIT_28 of Piece_Bitmap : label is
  "9230C3138C30C30D8048C4818049248180312301800 0C00180000001FFFFFFFF";       -- selected QUEEN
attribute INIT_29 of Piece_Bitmap : label is
  "82E211D182A619518325293987292D39862925198429250D8C31C30D9230C313";       -- selected QUEEN
attribute INIT_2a of Piece_Bitmap : label is
  "80700181804FFE8180C00081809FFE418180006181E001E1813FFF21836291A1";       -- selected QUEEN
attribute INIT_2b of Piece_Bitmap : label is
  "FFFFFFFF8000000180000018007FC018078038180C000C180F007C1804FFC81";       -- selected QUEEN
attribute INIT_2c of Piece_Bitmap : label is
  "800EE001800380018006C001800440018006C00180038001800000001FFFFFFFF";       -- selected BUSHOP
attribute INIT_2d of Piece_Bitmap : label is
  "8040040180C10601808102018087C2018041040180410401800200C0180383801";  -- selected BUSHOP
attribute INIT_2e of Piece_Bitmap : label is
  "800FE00180383801803838018017D00180101001801830018 03FF80180200C01";-- selected BUSHOP
attribute INIT_2f of Piece_Bitmap : label is
  "FFFFFFFF8000000180000001880000218FFC7FE19006C0119FF39FF1800D6001";       -- selected BUSHOP
attribute INIT_30 of Piece_Bitmap : label is
  "80D08C01805CF801805DE0018075000180C3000180020001800000001FFFFFFFF";       -- selected KNIGHT
attribute INIT_31 of Piece_Bitmap : label is
  "840080618600C061820040C1810000818140008181600181 81A0030180800601";  -- selected KNIGHT
attribute INIT_32 of Piece_Bitmap : label is
  "80040019818600198F8100198D41001998618011 9A308031880E80318C038031";-- selected KNIGHT
attribute INIT_33 of Piece_Bitmap : label is
  "FFFFFFFF8000000180000001801FFFF9801000198010001980100019800800 19";          -- selected KNIGHT
attribute INIT_34 of Piece_Bitmap : label is
  "8100010181000101813C79018124490181E7CF018000000180000001FFFFFFFF";       -- selected ROOK
attribute INIT_35 of Piece_Bitmap : label is
  "802008018020080180200801802008018020080 1807FFC0180C0060181FFFF01";          -- selected ROOK
attribute INIT_36 of Piece_Bitmap : label is
  "8100010181000101 81FFFF0180C00601807FFC018020080180200801802008 01";          -- selected ROOK
attribute INIT_37 of Piece_Bitmap : label is
  "FFFFFFFF8000000180000001800000187FFFFC184000041840000418 7FFFFC1";       -- selected ROOK
attribute INIT_38 of Piece_Bitmap : label is
  "800630018006300180036001800180018000000180000001800000001FFFFFFFF";-- selected PAWN
attribute INIT_39 of Piece_Bitmap : label is
  "800C180180180C018010040180100401800 80C01800C18018006300180022001";-- selected PAWN
attribute INIT_3a of Piece_Bitmap : label is
  "80800081808000818 0C0018180400101802003018030060 1801C1C0180063001";  -- selected PAWN
attribute INIT_3b of Piece_Bitmap : label is
  "FFFFFFFF8000000180000001800000018000000180FFFF818080008180800081";          -- selected PAWN
attribute INIT_3c of Piece_Bitmap : label is
```

```
    "CDBFFFFFCDBFFFFFDDBF3FFFDDBF3FFF889F3FFFFFFFFFFFFFFFFFFFFFFFFF";        -- win
  attribute INIT_3d of Piece_Bitmap : label is
    "E87F3F79E87F3E3BEC7E3C33ECBF3F43CCBFFFFFCDBFFFFFCDBFFFFFCDBFFFFF";        -- win
  attribute INIT_3e of Piece_Bitmap : label is
    "E67F3F79E67F3F79E67F3F79E67F3F79E27F3F79EA7F3F79EA7F3F79E87F3F79";  -- win
  attribute INIT_3f of Piece_Bitmap : label is
    "FFFFFFFFFFFFFFFFF6FC1C30F6FF3E79F6FF3F79F6FF3F79F6FF3F79F67F3F79";        -- win

begin    -- Behavioral

  ---------------------------------------------------------------------------
  --
  -- Instances of the block RAMs
  -- Each is 2K bytes, so 4K total
  --
  ---------------------------------------------------------------------------

  -- First 2K is used for Pieces       (only 300 bytes in use)
  -- Remaining 2K holds the piece bitmap    (1 block RAMs)
  --

  -- Port A is used for communication with the OPB
  -- Port B is for video

  Board_Block : RAMB16_S9_S9
    port map (
      DOA     => DOUT0,
      DOPA0   => DOP0A,
      ADDRA => ABus(10 downto 0),
      CLKA    => OPB_Clk,
      DIA     => DBus(7 downto 0),
      DIPA0   => '1',
      ENA     => '1',
      SSRA    => RST(0),
      WEA     => WE(0),
      DOB     => DOUTB0,
      DOPB0   => DOP0B,
      ADDRB => BlockAddr(10 downto 0),
      CLKB    => Pixel_Clock,
      DIB     => X"00",
      DIPB0   => '1',
      ENB     => '1',
      SSRB    => '0',
      WEB     => '0');

    Piece_Bitmap : RAMB16_S36_S36
    port map (
      DOA     => DOUT1,
      DOPA    => DOP1A,
      ADDRA => ABus(8 downto 0),
      CLKA    => OPB_Clk,
      DIA     => DBus(31 downto 0),
      DIPA    => "1111",
      ENA     => '1',
      SSRA    => RST(1),
      WEA     => WE(1),
      DOB     => DOUTB1,
      DOPB    => DOP1B,
      ADDRB => PieceAddr(8 downto 0),
      CLKB    => Pixel_Clock,
      DIB     => X"00000000",
      DIPB    => "1111",
      ENB     => '1',
      SSRB    => '0',
      WEB     => '0');

  ---------------------------------------------------------------------------
  --
  -- OPB-RAM controller
  --
```

```
------------------------------------------------------------------------------

-- Unused OPB control signals
VGA_errAck <= '0';
VGA_retry <= '0';
VGA_toutSup <= '0';

-- Latch the relevant OPB signals from the OPB, since they arrive late
LatchOPB: process (OPB_Clk, OPB_Rst)
begin
   if OPB_Rst = '1' then
      Abus <= ( others => '0' );
      DBus <= ( others => '0' );
      RNW <= '1';
      select_delayed <= '0';
   elsif OPB_Clk'event and OPB_Clk = '1' then
      ABus <= OPB_ABus;
      DBus <= OPB_DBus;
      RNW <= OPB_RNW;
      select_delayed <= OPB_Select;
   end if;
end process LatchOPB;

-- Address bits 31 downto 12 is our chip select
-- 11 is the RAM page select
-- 10 downto 0 is the RAM byte select

ChipSelect <=
   '1' when select_delayed = '1' and
        (ABus(31 downto 12) = BASEADDR(31 downto 12)) and
        MemCycle1 = '0' and MemCycle2 = '0' else
   '0';

RamPageAddress <= ABus(11);

RamSelect <=
   "01" when RamPageAddress = '0' else
   "10" when RamPageAddress = '1' else
   "00";

MemCycleFSM : process(OPB_Clk, OPB_Rst)
begin
   if OPB_Rst = '1' then
      MemCycle1 <= '0';
      MemCycle2 <= '0';
   elsif OPB_Clk'event and OPB_Clk = '1' then
      MemCycle2 <= MemCycle1;
      MemCycle1 <= ChipSelect;
   end if;
end process MemCycleFSM;

VGA_xferAck <= MemCycle2;

WE <= RamSelect when ChipSelect = '1' and RNW = '0' and OPB_Rst = '0' else "00";

RST <= not RamSelect when ChipSelect = '1' and RNW = '1' and OPB_Rst = '0' else "11";

ReadData <= (DOUT0 & DOUT0 & DOUT0 & DOUT0) or DOUT1 when MemCycle1 = '1' else X"00000000";

-- DBus(31 downto 24) is the byte for addresses ending in 0

GenDOut: process (OPB_Clk, OPB_Rst)
begin
   if OPB_Rst = '1' then
      DBus_out <= ( others => '0');
   elsif OPB_Clk'event and OPB_Clk = '1' then
      DBus_out <= ReadData;
   end if;
end process GenDOut;
```

```
VGA_DBus <= DBus_out;

------------------------------------------------------------------------
--
-- Video controller
--
------------------------------------------------------------------------

-- Pixel clock divider

Pixel_clk_divider : process (Pixel_Clock_2x, OPB_Rst)
begin
   if OPB_Rst = '1' then
      Pixel_Clock <= '0';
   elsif Pixel_Clock_2x'event and Pixel_Clock_2x = '1' then
      Pixel_Clock <= not Pixel_Clock;
   end if;
end process Pixel_clk_divider;

-- Horizontal and vertical counters

HCounter : process (Pixel_Clock, OPB_Rst)
begin
   if OPB_Rst = '1' then
      Hcount <= (others => '0');
   elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if EndOfLine = '1' then
         Hcount <= (others => '0');
      else
         Hcount <= Hcount + 1;
      end if;
   end if;
end process HCounter;

EndOfLine <= '1' when Hcount = HTOTAL - 1 else '0';

VCounter: process (Pixel_Clock, OPB_Rst)
begin
   if OPB_Rst = '1' then
      Vcount <= (others => '0');
   elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if EndOfLine = '1' then
         if EndOfField = '1' then
            Vcount <= (others => '0');
         else
            Vcount <= Vcount + 1;
         end if;
      end if;
   end if;
end process VCounter;

EndOfField <= '1' when Vcount = VTOTAL - 1 else '0';

-- State machines to generate HSYNC, VSYNC, HBLANK, and VBLANK

HSyncGen : process (Pixel_Clock, OPB_Rst)
begin
   if OPB_Rst = '1' then
      VIDOUT_HSYNC <= '0';
   elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if EndOfLine = '1' then
         VIDOUT_HSYNC <= '1';
      elsif Hcount = HSYNC - 1 then
         VIDOUT_HSYNC <= '0';
      end if;
   end if;
end process HSyncGen;

HBlankGen : process (Pixel_Clock, OPB_Rst)
begin
```

```
   if OPB_Rst = '1' then
      HBLANK_N <= '0';
   elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if Hcount = HSYNC + HBACK_PORCH - 1 then
         HBLANK_N <= '1';
      elsif Hcount = HSYNC + HBACK_PORCH + HACTIVE - 1 then
         HBLANK_N <= '0';
      end if;
   end if;
end process HBlankGen;

VSyncGen : process (Pixel_Clock, OPB_Rst)
begin
   if OPB_Rst = '1' then
      VIDOUT_VSYNC <= '1';
   elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if EndOfLine ='1' then
         if EndOfField = '1' then
            VIDOUT_VSYNC <= '1';
         elsif VCount = VSYNC - 1 then
            VIDOUT_VSYNC <= '0';
         end if;
      end if;
   end if;
end process VSyncGen;

VBlankGen : process (Pixel_Clock, OPB_Rst)
begin
   if OPB_Rst = '1' then
      VBLANK_N <= '0';
   elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if EndOfLine ='1' then
         if Vcount = VSYNC + VBACK_PORCH - 1 then
            VBLANK_N <= '1';
         elsif VCount = VSYNC + VBACK_PORCH + VACTIVE - 1 then
            VBLANK_N <= '0';
         end if;
      end if;
   end if;
end process VBlankGen;

-- RAM read triggers and shift register control

LoadNShift <= '1' when Hcount(4 downto 0) = X"0F" else '0';

-- Correction of 4 needed to calculate the character address before the
-- character is displayed
BlockColumn <= Hcount - HSYNC - HBACK_PORCH + 32;
Column <= BlockColumn(9 downto 5);    -- / 32
BlockRow <= Vcount - VSYNC - VBACK_PORCH;
Row <= BlockRow(8 downto 5);    -- / 32

-- Column + Row * 20
BlockAddr <= Column +
                  ("000" & Row(3 downto 0) & "0000") +
                  ("00000" & Row(3 downto 0) & "00");

-- Most significant bit of character ignored

PieceAddr(8 downto 5) <= DOUTB0(3 downto 0);
PieceAddr(4 downto 0) <= BlockRow(4 downto 0);

PieceData <= DOUTB1;
PlayerData <= DOUTB0(5);
CursorData <= DOUTB0(4);

Board_background: process (Column, Row)
begin
   if (Column > 5 and Column < 14) and (Row > 3 and Row < 12) then
      if Column(0) = Row(0) then
```

```
            BackgroundData <= "01";
         else
            BackgroundData <= "10";
         end if;
      else
         BackgroundData <= "00";
      end if;
end process Board_background;

-- Shift register

ShiftRegister: process (Pixel_Clock, OPB_Rst)
begin
   if OPB_Rst = '1' then
      ShiftData <= X"00000000";
      Player <= '0';
      Cursor <= '0';
      Background <= "00";
   elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if LoadNShift = '1' then
         ShiftData <= PieceData;
         Player <= PlayerData;
         Cursor <= CursorData;
         Background <= BackgroundData;
      else
         ShiftData <= ShiftData(30 downto 0) & ShiftData(31);
         Player <= Player;
         Cursor <= Cursor;
         Background <= Background;
      end if;
   end if;
end process ShiftRegister;

VideoData <= ShiftData(31);

-- Video signals going to the "video DAC"

VideoOut: process (Pixel_Clock, OPB_Rst, Background, Player, Cursor)
begin
   if OPB_Rst = '1' then
      VIDOUT_RED   <= '0';
      VIDOUT_BLUE  <= '0';
      VIDOUT_GREEN <= '0';
   elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if HBLANK_N = '1' and VBLANK_N = '1' then
         if VideoData = '1' then
            if Player = '0' and Cursor = '0' then
               VIDOUT_RED   <= '1';
               VIDOUT_GREEN <= '0';
               VIDOUT_BLUE  <= '0';
            elsif Player = '0' and Cursor = '1' then
               VIDOUT_RED   <= '0';
               VIDOUT_GREEN <= '1';
               VIDOUT_BLUE  <= '1';
            elsif Player = '1' and Cursor = '1' then
               VIDOUT_RED   <= '1';
               VIDOUT_GREEN <= '1';
               VIDOUT_BLUE  <= '0';
            else
               VIDOUT_RED   <= '0';
               VIDOUT_GREEN <= '0';
               VIDOUT_BLUE  <= '1';
            end if;
         else
            if Cursor = '1' then
               VIDOUT_RED   <= '0';
               VIDOUT_GREEN <= '1';
               VIDOUT_BLUE  <= '0';
            else
               if Background(1) = '1' then
```

```
                VIDOUT_RED   <= '0';
                VIDOUT_GREEN <= '0';
                VIDOUT_BLUE  <= '0';
              elsif Background(0) = '1' then
                VIDOUT_RED   <= '1';
                VIDOUT_GREEN <= '1';
                VIDOUT_BLUE  <= '1';
              else
                if Hcount(0) = '1' xor Vcount(0) = '1' then
                  VIDOUT_RED   <= '1';
                  VIDOUT_GREEN <= '1';
                  VIDOUT_BLUE  <= '1';
                else
                  VIDOUT_RED   <= '0';
                  VIDOUT_GREEN <= '0';
                  VIDOUT_BLUE  <= '0';
                end if;
              end if;
            end if;
          end if;
        else
          VIDOUT_RED   <= '0';
          VIDOUT_GREEN <= '0';
          VIDOUT_BLUE  <= '0';
        end if;
      end if;
    end process VideoOut;

    VIDOUT_CLK <= Pixel_Clock;

end Behavioral;
```

7. C Code

```
/*
 *    * Copyright (c) 2004 Xilinx, Inc.    All rights reserved.
 *
 * Xilinx, Inc.
 * XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS A
 * COURTESY TO YOU.    BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS
 * ONE POSSIBLE      IMPLEMENTATION OF THIS FEATURE, APPLICATION OR
 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION
 * IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE
 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION
 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO
 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO
 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE
 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 * chess game code by Sheng-Kuo Lu
 * sklu.iic93g@nctu.edu.tw
 */

// Located in: microblaze_0/include/xparameters.h
#include "xparameters.h"

#include "xutil.h"
#include "xintc_l.h"
#include "xuartlite_l.h"
#include "xgpio_l.h"

#define PIECE_OFFSET 0xA00

/* Address of a particular chess piece on the screen (rows are 20) */
#define BOARD(c,r) \
    (((unsigned char *)(XPAR_VGA_BASEADDR))[(c) + ((r) << 4) + ((r) << 2)])
#define CHESS(c,r) (BOARD((c+6),(r+4)))

/* Start of bitmap memory */
#define PIECE ((unsigned int *)XPAR_VGA_BASEADDR + PIECE_OFFSET)

#define EMPTY 0x00
#define KING 0x01
#define QUEEN 0x02
#define BISHOP 0x03
#define KNIGHT 0x04
#define ROOK 0x05
#define PAWN 0x06
#define SELECT_OFFSET 0x08
#define WINFLAG 0x0f
#define CURSOR 0x10
#define ABS(v) (((v) < 0) ? (0-(v)) : (v))
#define GET_TYPE(c) ((c) & 0x0f)
#define GET_PPL(c) (((c) & 0x20) >> 5)
#define GET_HINT(c) ((GET_TYPE(c)) > 7)

/* Must be a power of two */
#define SCANCODE_BUFFER_SIZE 16
unsigned char scancode_buffer[SCANCODE_BUFFER_SIZE]; // act as a circular queue
int scancode_buffer_front = 0;
int scancode_buffer_rear = 0;

int row = 0, col = 0; //cursor location
unsigned char ps = 0; //piece select flag
unsigned char player = 0; //player flag
unsigned char selected; //the selected piece register
int rsel = 0, csel = 0; //the selected piece location
unsigned char gameover = 0; //game over flag
int prison[2] = {0, 0}; //the captured prison
```

```
int lastmove[5] = {0, 0, 0, 0, 0};
int moveking[2] = {0 ,0};
int movelrook[2] = {0, 0};
int moverrook[2] = {0, 0};

unsigned char get_character() //queue delete operation
{
   unsigned char result;
   result = scancode_buffer[scancode_buffer_front];
   scancode_buffer_front = (scancode_buffer_front + 1) & (SCANCODE_BUFFER_SIZE - 1);
   return result;
}

int keyboard_interrupt_count = 0;
unsigned int scan_code = 0;
unsigned int keyboard_bit = 11;
unsigned int pbit=1, cbit=0, start=0;

/* PS/2 Keyboard interrupt service routine */
void ps2_int_handler(void *baseaddr_p)
{
   keyboard_interrupt_count++;

/* We're reading a scancode and the keyboard clock fell: sample the data */
   cbit = XGpio_mReadReg(XPAR_PS2IO_BASEADDR, XGPIO_DATA_OFFSET);
   if ((pbit == 1) && (cbit == 0)) start = 1;
   if (start == 1) {
      scan_code = (scan_code >> 1) | (cbit << 9);
      if (--keyboard_bit == 0) {
         /* keycode is ready */
         if (scancode_buffer_front != ((scancode_buffer_rear + 1) & (SCANCODE_BUFFER_SIZE - 1))) {
            /* buffer is not full; add the character */
            scancode_buffer[scancode_buffer_rear] = scan_code;
            scancode_buffer_rear = (scancode_buffer_rear + 1) & (SCANCODE_BUFFER_SIZE - 1);
         }
         keyboard_bit = 11;
         start = 0;
      }
   }
   pbit = cbit;

   /* Acknowledge the interrupt */
   XGpio_mWriteReg(XPAR_PS2IO_BASEADDR, XGPIO_ISR_OFFSET, 1);
}

void check_hint() //check if any place can move before piece select
{
   int i, j;
   for (i=0;i<8;i++) {
      for (j=0;j<8;j++) {
         if (GET_TYPE(CHESS(i, j)) >= SELECT_OFFSET) {
         ps = 1;
         return;
         }
      }
   }
}

void clear_hint() //clear all move hint after a move done
{
   unsigned char hint;
   int i, j;
   for (i=0;i<8;i++) {
      for (j=0;j<8;j++) {
         hint = CHESS(i, j);
         if (GET_TYPE(hint) >= SELECT_OFFSET) {
         CHESS(i, j) = hint - SELECT_OFFSET;
         }
      }
   }
```

```
}

int find_king() //find and move cursor to the king
{
   int i, j;
   unsigned char finder;
   for (i=0; i<8; i++) {
      for (j=0; j<8; j++) {
         finder = CHESS(j, i);
         if ((GET_TYPE(finder) == KING) && (GET_PPL(finder) == player)) {
            row = i;
            col = j;
            return 1;
         }
      }
   }
   return 0; //no king found means game over
}

void check_bishop(int step) //checking bishop-like moveing direction
{
   unsigned char move;
   int i;
   for (i=1; i<=step; i++) {
      if (((rsel-i) < 0) || ((csel-i) < 0)) break;
      move = CHESS(csel-i, rsel-i);
      if (GET_TYPE(move) == EMPTY) {
         CHESS(csel-i, rsel-i) = (player << 5) | SELECT_OFFSET;
      } else {
         if (GET_PPL(move) != player) CHESS(csel-i, rsel-i) = move + SELECT_OFFSET;
         break;
      }
   }
   for (i=1; i<=step; i++) {
      if (((rsel-i) < 0) || ((csel+i) > 7)) break;
      move = CHESS(csel+i, rsel-i);
      if (GET_TYPE(move) == EMPTY) {
         CHESS(csel+i, rsel-i) = (player << 5) | SELECT_OFFSET;
      } else {
         if (GET_PPL(move) != player) CHESS(csel+i, rsel-i) = move + SELECT_OFFSET;
         break;
      }
   }
   for (i=1; i<=step; i++) {
      if (((rsel+i) > 7) || ((csel-i) < 0)) break;
      move = CHESS(csel-i, rsel+i);
      if (GET_TYPE(move) == EMPTY) {
         CHESS(csel-i, rsel+i) = (player << 5) | SELECT_OFFSET;
      } else {
         if (GET_PPL(move) != player) CHESS(csel-i, rsel+i) = move + SELECT_OFFSET;
         break;
      }
   }
   for (i=1; i<=step; i++) {
      if (((rsel+i) > 7) || ((csel+i) > 7)) break;
      move = CHESS(csel+i, rsel+i);
      if (GET_TYPE(move) == EMPTY) {
         CHESS(csel+i, rsel+i) = (player << 5) | SELECT_OFFSET;
      } else {
         if (GET_PPL(move) != player) CHESS(csel+i, rsel+i) = move + SELECT_OFFSET;
         break;
      }
   }
}

void check_rook(int step) //checking rook-like moving direction
{
   unsigned char move;
   int i;
   for (i=1; i<=step; i++) {
```

```
      if ((rsel-i) < 0) break;
      move = CHESS(csel, rsel-i);
      if (GET_TYPE(move) == EMPTY) {
         CHESS(csel, rsel-i) = (player << 5) | SELECT_OFFSET;
      } else {
         if (GET_PPL(move) != player) CHESS(csel, rsel-i) = move + SELECT_OFFSET;
         break;
      }
   }
   for (i=1; i<=step; i++) {
      if ((csel-i) < 0) break;
      move = CHESS(csel-i, rsel);
      if (GET_TYPE(move) == EMPTY) {
         CHESS(csel-i, rsel) = (player << 5) | SELECT_OFFSET;
      } else {
         if (GET_PPL(move) != player) CHESS(csel-i, rsel) = move + SELECT_OFFSET;
         break;
      }
   }
   for (i=1; i<=step; i++) {
      if ((csel+i) > 7) break;
      move = CHESS(csel+i, rsel);
      if (GET_TYPE(move) == EMPTY) {
         CHESS(csel+i, rsel) = (player << 5) | SELECT_OFFSET;
      } else {
         if (GET_PPL(move) != player) CHESS(csel+i, rsel) = move + SELECT_OFFSET;
         break;
      }
   }
   for (i=1; i<=step; i++) {
      if ((rsel+i) > 7) break;
      move = CHESS(csel, rsel+i);
      if (GET_TYPE(move) == EMPTY) {
         CHESS(csel, rsel+i) = (player << 5) | SELECT_OFFSET;
      } else {
         if (GET_PPL(move) != player) CHESS(csel, rsel+i) = move + SELECT_OFFSET;
         break;
      }
   }
}

void piece_select() //all move checking when a piece selected
{
   unsigned char move;
   int i;

   selected = CHESS(col, row);
   if (GET_PPL(selected) == player) {
      rsel = row;
      csel = col;
      switch (GET_TYPE(selected)) {
      case KING:
         check_bishop(1);
         check_rook(1);
         if (moveking[player] != 1) {
            if (movelrook[player] != 2) {
               for (i=1; i<4; i++) {
                  if ((GET_TYPE(CHESS(csel-i, rsel)) & 0x07) != EMPTY) break;
               }
               if (i == 4) CHESS(csel-3, rsel) = (player << 5) | SELECT_OFFSET;
            }
            if (moverrook[player] != 2) {
               for (i=1; i<3; i++) {
                  if ((GET_TYPE(CHESS(csel+i, rsel)) & 0x07) != EMPTY) break;
               }
               if (i == 3) CHESS(csel+2, rsel) = (player << 5) | SELECT_OFFSET;
            }
         }
         break;
      case QUEEN:
```

```
    check_bishop(7);
    check_rook(7);
    break;
case BISHOP:
    check_bishop(7);
    break;
case KNIGHT:
    if (((csel-2) >= 0) && ((rsel-1) >= 0)) {
        move = CHESS(csel-2, rsel-1);
        if (GET_TYPE(move) == EMPTY) {
            CHESS(csel-2, rsel-1) = (player << 5) | SELECT_OFFSET;
        } else if (GET_PPL(move) != player) {
            CHESS(csel-2, rsel-1) = move + SELECT_OFFSET;
        }
    }
    if (((csel-2) >= 0) && ((rsel+1) <= 7)) {
        move = CHESS(csel-2, rsel+1);
        if (GET_TYPE(move) == EMPTY) {
            CHESS(csel-2, rsel+1) = (player << 5) | SELECT_OFFSET;
        } else if (GET_PPL(move) != player) {
            CHESS(csel-2, rsel+1) = move + SELECT_OFFSET;
        }
    }
    if (((csel+2) <= 7) && ((rsel-1) >= 0)) {
        move = CHESS(csel+2, rsel-1);
        if (GET_TYPE(move) == EMPTY) {
            CHESS(csel+2, rsel-1) = (player << 5) | SELECT_OFFSET;
        } else if (GET_PPL(move) != player) {
            CHESS(csel+2, rsel-1) = move + SELECT_OFFSET;
        }
    }
    if (((csel+2) <= 7) && ((rsel+1) <= 7)) {
        move = CHESS(csel+2, rsel+1);
        if (GET_TYPE(move) == EMPTY) {
            CHESS(csel+2, rsel+1) = (player << 5) | SELECT_OFFSET;
        } else if (GET_PPL(move) != player) {
            CHESS(csel+2, rsel+1) = move + SELECT_OFFSET;
        }
    }
    if (((csel-1) >= 0) && ((rsel-2) >= 0)) {
        move = CHESS(csel-1, rsel-2);
        if (GET_TYPE(move) == EMPTY) {
            CHESS(csel-1, rsel-2) = (player << 5) | SELECT_OFFSET;
        } else if (GET_PPL(move) != player) {
            CHESS(csel-1, rsel-2) = move + SELECT_OFFSET;
        }
    }
    if (((csel-1) >= 0) && ((rsel+2) <= 7)) {
        move = CHESS(csel-1, rsel+2);
        if (GET_TYPE(move) == EMPTY) {
            CHESS(csel-1, rsel+2) = (player << 5) | SELECT_OFFSET;
        } else if (GET_PPL(move) != player) {
            CHESS(csel-1, rsel+2) = move + SELECT_OFFSET;
        }
    }
    if (((csel+1) <= 7) && ((rsel-2) >= 0)) {
        move = CHESS(csel+1, rsel-2);
        if (GET_TYPE(move) == EMPTY) {
            CHESS(csel+1, rsel-2) = (player << 5) | SELECT_OFFSET;
        } else if (GET_PPL(move) != player) {
            CHESS(csel+1, rsel-2) = move + SELECT_OFFSET;
        }
    }
    if (((csel+1) <= 7) && ((rsel+2) <= 7)) {
        move = CHESS(csel+1, rsel+2);
        if (GET_TYPE(move) == EMPTY) {
            CHESS(csel+1, rsel+2) = (player << 5) | SELECT_OFFSET;
        } else if (GET_PPL(move) != player) {
            CHESS(csel+1, rsel+2) = move + SELECT_OFFSET;
        }
```

```
        }
        break;
      case ROOK:
        check_rook(7);
        if ((csel == 0) && (movelrook[player] != 2)) movelrook[player] = 1;
        if ((csel == 7) && (moverrook[player] != 2)) moverrook[player] = 1;
        break;
      case PAWN:
        switch (player) {
        case 0:
          if (GET_TYPE(CHESS(csel, rsel+1)) == EMPTY) {
            CHESS(csel, rsel+1) = (player << 5) | SELECT_OFFSET;
            if ((rsel == 1) && (GET_TYPE(CHESS(csel, rsel+2)) == EMPTY)) CHESS(csel, rsel+2) = (player << 5) |
SELECT_OFFSET;
          }
          move = CHESS(csel+1, rsel+1);
          if ((GET_PPL(move) != player) && (GET_TYPE(move) != EMPTY)) CHESS(csel+1, rsel+1) = move +
SELECT_OFFSET;
          move = CHESS(csel-1, rsel+1);
          if ((GET_PPL(move) != player) && (GET_TYPE(move) != EMPTY)) CHESS(csel-1, rsel+1) = move +
SELECT_OFFSET;
          if ((lastmove[0] == PAWN) && ((lastmove[1] - lastmove[3]) == 2) && (lastmove[3] == rsel) &&
              (ABS(lastmove[4] - csel) == 1))
            CHESS(lastmove[4], lastmove[3]+1) = (player << 5) | SELECT_OFFSET;
          break;
        case 1:
          if (GET_TYPE(CHESS(csel, rsel-1)) == EMPTY) {
            CHESS(csel, rsel-1) = (player << 5) | SELECT_OFFSET;
            if ((rsel == 6) && (GET_TYPE(CHESS(csel, rsel-2)) == EMPTY)) CHESS(csel, rsel-2) = (player << 5) |
SELECT_OFFSET;
          }
          move = CHESS(csel+1, rsel-1);
          if ((GET_PPL(move) != player) && (GET_TYPE(move) != EMPTY)) CHESS(csel+1, rsel-1) = move +
SELECT_OFFSET;
          move = CHESS(csel-1, rsel-1);
          if ((GET_PPL(move) != player) && (GET_TYPE(move) != EMPTY)) CHESS(csel-1, rsel-1) = move +
SELECT_OFFSET;
          if ((lastmove[0] == PAWN) && ((lastmove[3] - lastmove[1]) == 2) && (lastmove[3] == rsel) &&
              (ABS(lastmove[4] - csel) == 1))
            CHESS(lastmove[4], lastmove[3]-1) = (player << 5) | SELECT_OFFSET;
          break;
        default:
        }
        break;
      default:
      }
      check_hint();
    }
}

void piece_move() //operations when a piece move
{
    unsigned char check;
    check = CHESS(col, row);
    if (GET_HINT(check)) {
      if (GET_TYPE(check) != SELECT_OFFSET) { //capture occur
        switch (player) {
        case 0:
          BOARD(19-(prison[0]++), 0) = check - SELECT_OFFSET;
          break;
        case 1:
          BOARD(prison[1]++, 14) = check - SELECT_OFFSET;
          break;
        default:
        }
      }
      CHESS(csel, rsel) = EMPTY;
      if ((lastmove[0] == PAWN) && (GET_TYPE(selected)== PAWN) && (col == lastmove[4]) &&
          (ABS(lastmove[3] - lastmove[1]) == 2) && (ABS(row - lastmove[3]) == 1) && (col != csel)) {
        switch (player) {
```

```
            case 0:
               check = CHESS(col, row-1);
               BOARD(19-(prison[0]++), 0) = check;
               CHESS(col, row-1) = EMPTY;
               break;
            case 1:
               check = CHESS(col, row+1);
               BOARD(prison[1]++, 14) = check;
               CHESS(col, row+1) = EMPTY;
               break;
            default:
            }
         }
         if ((row == ((player + 7) & 7)) && (GET_TYPE(selected) == PAWN)) {
            CHESS(col, row) = selected - 4; //pawn promotion
         } else {
            CHESS(col, row) = selected;
         }
         if ((GET_TYPE(selected) == KING) && (ABS(col - csel) > 1)) {
            if (col == 1) {
               CHESS(2, row) = CHESS(0, row);
               CHESS(0, row) = EMPTY;
            } else {
               CHESS(5, row) = CHESS(7, row);
               CHESS(7, row) = EMPTY;
            }
         }
         lastmove[0] = GET_TYPE(selected); //piece type
         lastmove[1] = rsel;                    //start row
         lastmove[2] = csel;                    //start column
         lastmove[3] = row;                      //end row
         lastmove[4] = col;                      //end column
         if (GET_TYPE(selected) == KING) {
            moveking[player] = 1;
         } else if (GET_TYPE(selected) == ROOK) {
            if (movelrook[player] == 1) {
               movelrook[player] = 2;
            } else if (moverrook[player] == 1) {
               moverrook[player] = 2;
            }
         }
         player = player ^ 1;
         if (player == 0) {
            BOARD(0, 0) = 0x07;
            BOARD(19,14) = 0x00;
         } else {
            BOARD(0, 0) = 0x00;
            BOARD(19,14) = 0x27;
         }
         ps = 0;
         clear_hint();
         gameover = find_king() ^ 1;
      }
   }

   void game_reset() //initial the game system to restart
   {
      int i, j;
      for (i=0; i<20; i++) {
         for (j=0; j<15; j++) {
            BOARD(i, j) = 0x00;
         }
      }
      CHESS(0, 0) = ROOK;
      CHESS(1, 0) = KNIGHT;
      CHESS(2, 0) = BISHOP;
      CHESS(3, 0) = QUEEN;
      CHESS(4, 0) = KING;
      CHESS(5, 0) = BISHOP;
      CHESS(6, 0) = KNIGHT;
```

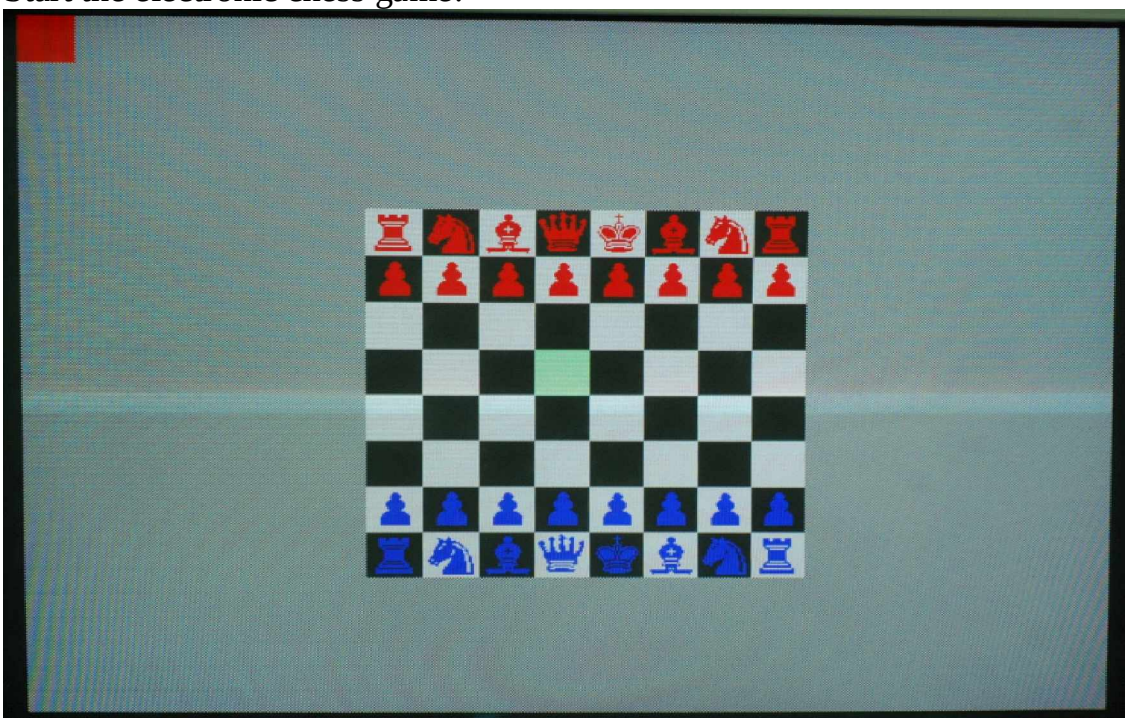
```
    CHESS(7, 0) = ROOK;
    for (i=0; i<8; i++)
      CHESS(i, 1) = PAWN;
    CHESS(0, 7) = ROOK | 0x20;
    CHESS(1, 7) = KNIGHT | 0x20;
    CHESS(2, 7) = BISHOP | 0x20;
    CHESS(3, 7) = QUEEN | 0x20;
    CHESS(4, 7) = KING | 0x20;
    CHESS(5, 7) = BISHOP | 0x20;
    CHESS(6, 7) = KNIGHT | 0x20;
    CHESS(7, 7) = ROOK | 0x20;
    for (i=0; i<8; i++)
      CHESS(i, 6) = PAWN | 0x20;
    gameover = 0;
    prison[0] = 0;
    prison[1] = 0;
    moveking[0] = 0;
    moveking[1] = 0;
    movelrook[0] = 0;
    movelrook[1] = 0;
    moverrook[0] = 0;
    moverrook[1] = 0;
    if (player == 0) {
      BOARD(0, 0) = 0x07;
      BOARD(19,14) = 0x00;
    } else {
      BOARD(0, 0) = 0x00;
      BOARD(19,14) = 0x27;
    }
}

int main (void) {
    int i, j, extended=0;
    unsigned char uc;
    char buf;

    /* Enable MicroBlaze interrupts */
    microblaze_enable_interrupts();
    /* Register the keyboard interrupt handler */
    XIntc_RegisterHandler(XPAR_OPB_INTC_0_BASEADDR, XPAR_OPB_INTC_0_SYSTEM_PS2C_INTR,
        (XInterruptHandler)ps2_int_handler, (void *)0);
    /* Start the interrupt controller */
    XIntc_mMasterEnable(XPAR_OPB_INTC_0_BASEADDR);
    /* Enable timer and keyboard interrupt requests in the interrupt controller */
    XIntc_mEnableIntr(XPAR_OPB_INTC_0_BASEADDR, XPAR_SYSTEM_PS2C_MASK);

  for (;;) {
    if (gameover) {
      for (i=4; i<12; i++) {
        BOARD(5, i) = ((player ^ 1) << 5) | WINFLAG;
        BOARD(14, i) = ((player ^ 1) << 5) | WINFLAG;
      }
      for (j=5; j<15; j++) {
        BOARD(j, 3) = ((player ^ 1) << 5) | WINFLAG;
        BOARD(j, 12) = ((player ^ 1) << 5) | WINFLAG;
      }
    }
    if (scancode_buffer_front != scancode_buffer_rear) {
      uc = get_character();
      if (extended == 1) {
        switch (uc) {
        case 0x75: //press up
          row = (row - 1) & 7;
          break;
        case 0x72: //press down
          row = (row + 1) & 7;
          break;
        case 0x6b: //press left
          col = (col - 1) & 7;
          break;
```

```
            case 0x74: //press right
               col = (col + 1) & 7;
               break;
            case 0x5a: //press enter
               if (ps == 0) {
                  if (gameover == 0) piece_select();
               } else {
                  if (gameover == 0) piece_move();
               }
               break;
            default:
            }
         } else {
            switch (uc) {
               case 0x04: //press F3
                  game_reset();
                  break;
               case 0x5a: //press enter
                  if (ps == 0) {
                     if (gameover == 0) piece_select();
                  } else {
                     if (gameover == 0) piece_move();
                  }
                  break;
               case 0x76: //press esc
                  ps = 0;
                  if (gameover == 0) clear_hint();
                  break;
               default:
               }
         }
         if (uc == 0xe0) {
            extended = 1;
         }
         if (uc == 0xf0) {
            scancode_buffer_front = scancode_buffer_rear;
            extended = 0;
         }
      }
      buf = CHESS(col, row);
      for (i = 0 ; i < 2 ; i++) {
         if (i == 0) {
            CHESS(col, row) = CURSOR | buf;
         } else {
            CHESS(col, row) = buf;
         }
         for (j = 0 ; j < 300000 ; j++ )    ; /* delay */
      }
   }
   return 0;
}
```

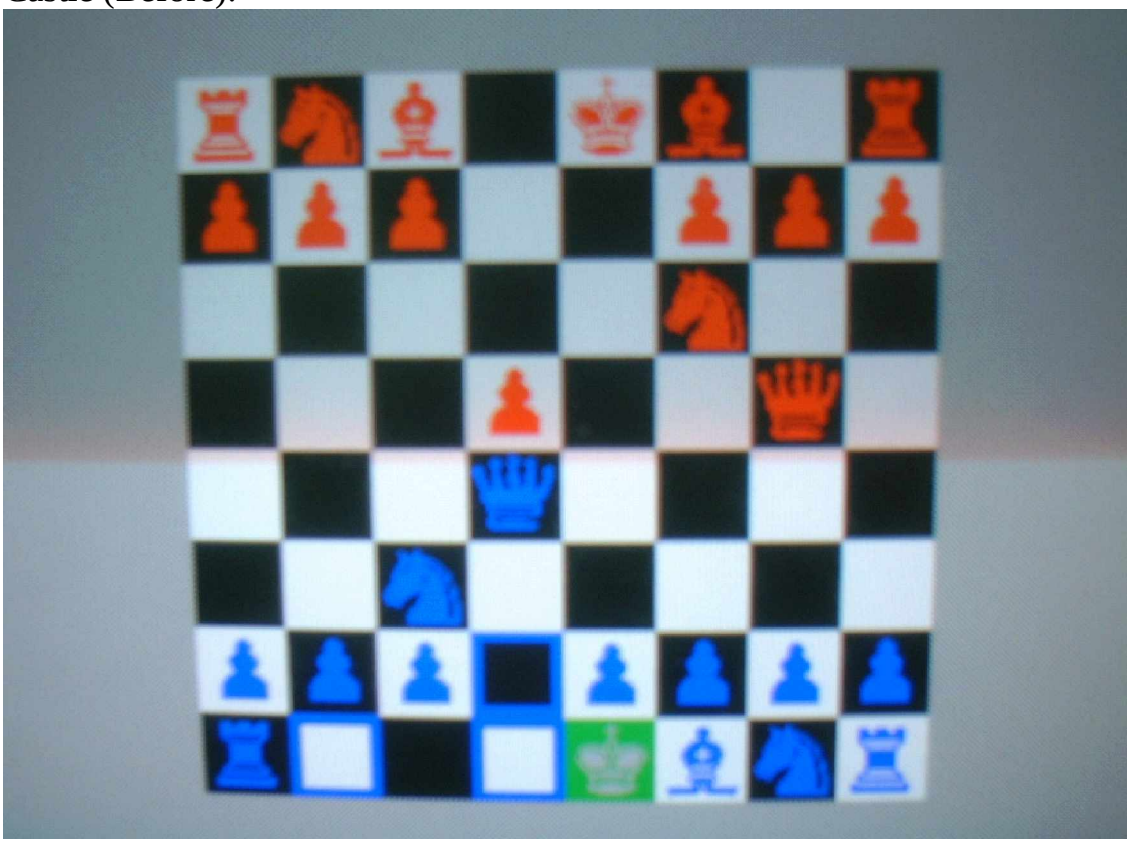
8. CONCLUSION

We have succeeded in designing and implementing a working chess program. The program allows both human and computer players and also offers a number of powerful features to manipulate the game. The computer movement portion presented the greatest challenge and yet proved to be by far the most interesting. The only deficiencies of the program lie in the strategy of the computers movement. The rest of the program was very straight forward to implement, with only a few temporary setbacks

Start the electronic chess game:



Castle (Before):



Castle (After):

CHECK checking: