Delta09 – An Interactive FPGA Circuit Simulator

Personal Report

Rubin Xu / rx201

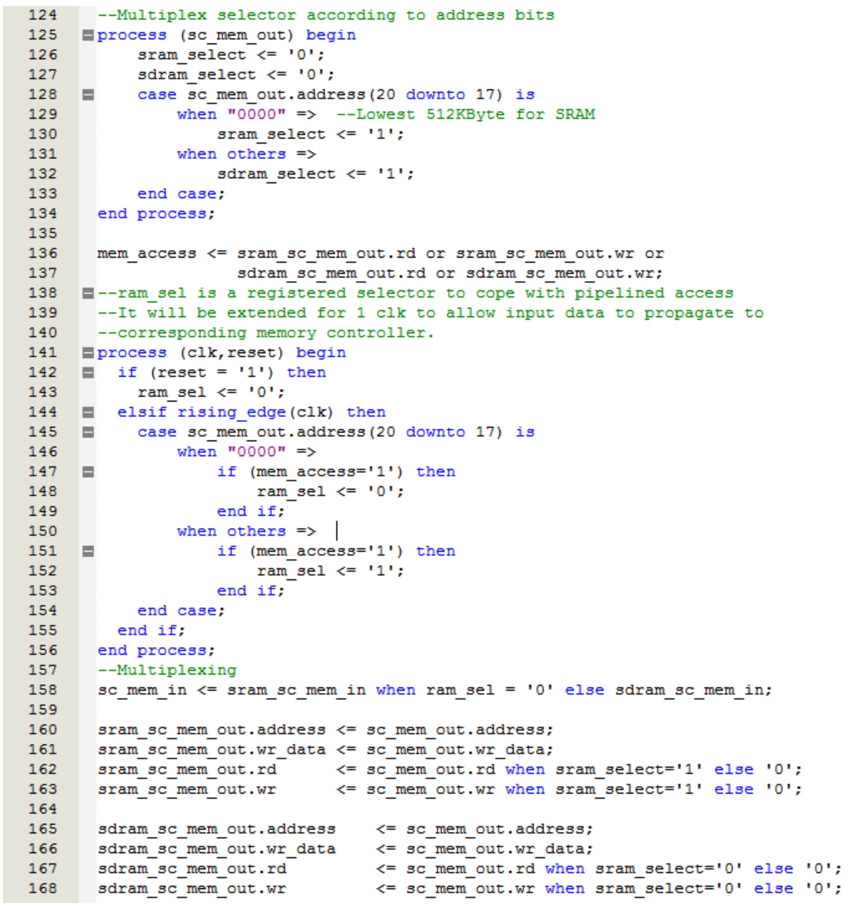
Contribution to Project

I was mainly in responsible for the hardware and communication part of the project. Soon after our group decided to use JoP – a soft processor that supports Java byte code natively, I focused on the internal architecture of JoP and investigated ways of incorporating resources on the DE2 board to JoP to suit out needs. Within the project timeframe I managed to integrate the following resources on DE2 into JoP architecture:

■ 8MByte SDRAM

The JoP is already able to access the 512Kbyte SRAM on DE2 as its main memory. But in order to give our simulator the flexibility of using more natural data structures instead of some complex memory-squeezing tricks, we decided to multiplex the 8MByte SDRAM into the existing memory space. As the JoP was not designed by SoPC builder, I cannot simply “plug in” the memory module, instead the SDRAM controller needed to be written in HDL and integrated into the existing JoP memory bus. After some search I managed to find an open source implementation of the SDRAM controller on DE2. However the bus interface of this SDRAM controller is “WishBone” and it is not directly compatible with the JoP’s own memory bus “SimpConn”. It took me a few hours before I successfully modified the SDRAM controller’s interface into JoP’s SimpConn and fixed a critical bug caused by pipelined access to the SDRAM. Because the SDRAM is inherently slower than the existing SRAM, it would improve the performance of JoP if the SRAM and SDRAM could be multiplexed together and SRAM occupies the lower memory space so most memory access will hit the SRAM. At this stage a problem emerged when I tried to multiplex the SRAM and SDRAM together. I tried to arrange the memory address space such that the SDRAM address 0 begins right after the SRAM’s largest address. This arrangement requires a subtraction to decode the correct SDRAM address from the uniform address and the subtraction became the critical path in timing. My solution was to overlap the address space of SRAM and SDRAM so no extra address decoding was required. The downside of this approach is that the lowest 512Kbyte of SDRAM is shadowed by SRAM so it is no longer visible and hence wasted.

Fig. 1 Multiplexer circuit for SRAM and SDRAM in JoP

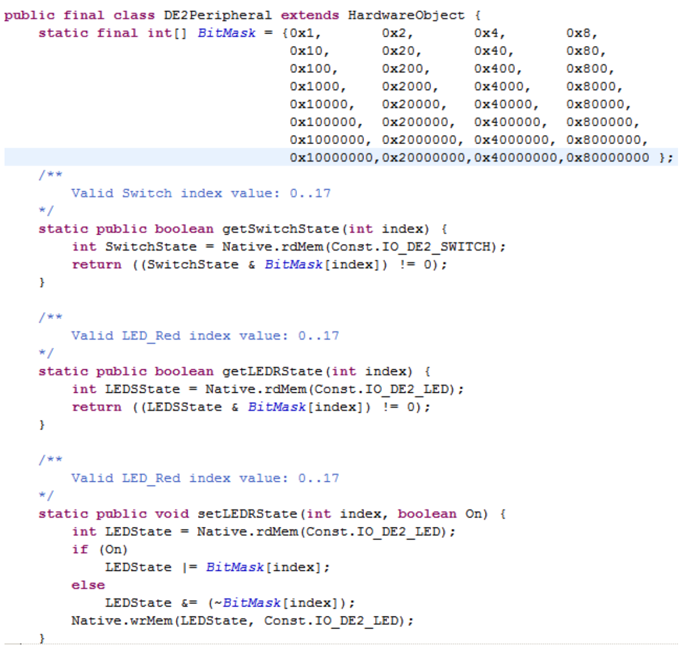


```
--Multiplex selector according to address bits
process (sc_mem_out) begin
    sram_select <= '0';
    sdram_select <= '0';
    case sc_mem_out.address(20 downto 17) is
        when "0000" =>  --Lowest 512KByte for SRAM
            sram_select <= '1';
        when others =>
            sdram_select <= '1';
    end case;
end process;

mem_access <= sram_sc_mem_out.rd or sram_sc_mem_out.wr or
              sdram_sc_mem_out.rd or sdram_sc_mem_out.wr;
--ram_sel is a registered selector to cope with pipelined access
--It will be extended for 1 clk to allow input data to propagate to
--corresponding memory controller.
process (clk,reset) begin
  if (reset = '1') then
    ram_sel <= '0';
  elsif rising_edge(clk) then
    case sc_mem_out.address(20 downto 17) is
        when "0000" =>
            if (mem_access='1') then
                ram_sel <= '0';
            end if;
        when others =>
            if (mem_access='1') then
                ram_sel <= '1';
            end if;
    end case;
  end if;
end process;
--Multiplexing
sc_mem_in <= sram_sc_mem_in when ram_sel = '0' else sdram_sc_mem_in;

sram_sc_mem_out.address <= sc_mem_out.address;
sram_sc_mem_out.wr_data <= sc_mem_out.wr_data;
sram_sc_mem_out.rd      <= sc_mem_out.rd when sram_select='1' else '0';
sram_sc_mem_out.wr      <= sc_mem_out.wr when sram_select='1' else '0';

sdram_sc_mem_out.address    <= sc_mem_out.address;
sdram_sc_mem_out.wr_data    <= sc_mem_out.wr_data;
sdram_sc_mem_out.rd         <= sc_mem_out.rd when sram_select='0' else '0';
sdram_sc_mem_out.wr         <= sc_mem_out.wr when sram_select='0' else '0';
```

■ LEDs and Switches

The VHDL codes for these peripherals were relatively easy after the architecture of JoP’s memory-mapped I/O devices was understood. By monitoring the address bus and linking the switches/LEDs pins to the data bus, switches/LEDs can be operated by simple memory access. Direct memory access is provided by one of the native methods from JoP's SDK. Below is a wrapper class I wrote for accessing switches and LEDs in Java.

Fig. 2 Wrapper class for Switch&LED access in JoP

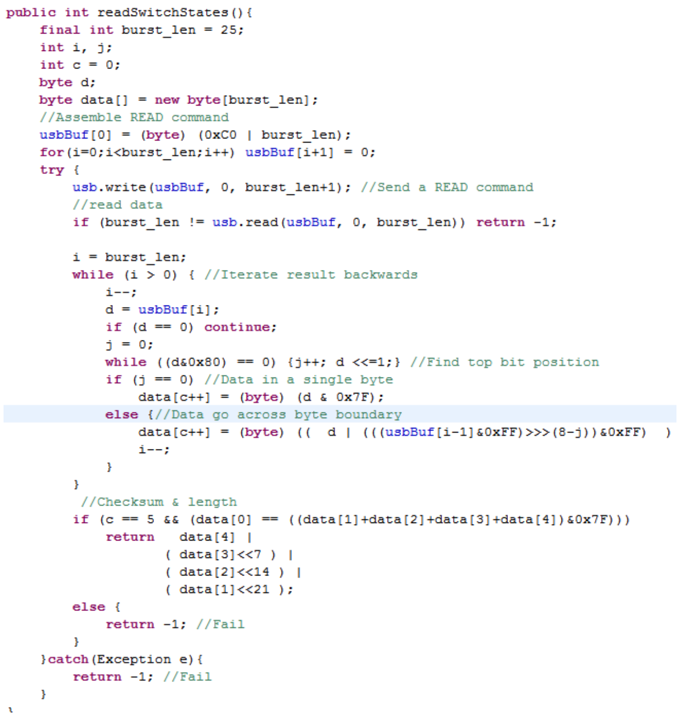


```java
public final class DE2Peripheral extends HardwareObject {
    static final int[] BitMask = {0x1,       0x2,       0x4,       0x8,
                                  0x10,      0x20,      0x40,      0x80,
                                  0x100,     0x200,     0x400,     0x800,
                                  0x1000,    0x2000,    0x4000,    0x8000,
                                  0x10000,   0x20000,   0x40000,   0x80000,
                                  0x100000,  0x200000,  0x400000,  0x800000,
                                  0x1000000, 0x2000000, 0x4000000, 0x8000000,
                                  0x10000000,0x20000000,0x40000000,0x80000000 };
    /**
        Valid Switch index value: 0..17
    */
    static public boolean getSwitchState(int index) {
        int SwitchState = Native.rdMem(Const.IO_DE2_SWITCH);
        return ((SwitchState & BitMask[index]) != 0);
    }

    /**
        Valid LED_Red index value: 0..17
    */
    static public boolean getLEDRState(int index) {
        int LEDSState = Native.rdMem(Const.IO_DE2_LED);
        return ((LEDSState & BitMask[index]) != 0);
    }

    /**
        Valid LED_Red index value: 0..17
    */
    static public void setLEDRState(int index, boolean On) {
        int LEDState = Native.rdMem(Const.IO_DE2_LED);
        if (On)
            LEDState |= BitMask[index];
        else
            LEDState &= (~BitMask[index]);
        Native.wrMem(LEDState, Const.IO_DE2_LED);
    }
```

■ Communication Link over USB-Blaster

I implemented a communication link over USB-Blaster for JoP on DE2. The USB-Blaster provides a bidirectional serial link to PC. What I did is designing a protocol for reliable packet transmission and implementing it as Java classes on both the JoP side and PC side. The hardware side of my design is basically a parallel-serial shift register circuit. In order to improve efficiency, I adapted the protocol and the hardware circuit to allow the hardware to recognize a packet and store it in its buffer for later processing. In terms of the PC side, the low-level USB communication is controlled by a DLL provided by the chip's vendor. This was made available to Java by me writing a JNI-based wrapper DLL & class. The following code shows part of the PC side high-level packet recovery algorithm. The low-level class *usb* is able to poll a number of bytes from the USB-Blaster. For the sake of efficiency *burst_len* of bytes is polled at a time. According to my hardware design the packet sent by JoP is distributed as collections of 8-bits within the data retrieved. However unwanted zeroes are introduced between adjacent collections due to the speed mismatch between USB-Blaster and JoP. My algorithm tries to recover data bytes from it and performs a simple checksum on the packet.

Fig. 3 PC Side packet data recovery algorithm



```java
public int readSwitchStates(){
    final int burst_len = 25;
    int i, j;
    int c = 0;
    byte d;
    byte data[] = new byte[burst_len];
    //Assemble READ command
    usbBuf[0] = (byte) (0xC0 | burst_len);
    for(i=0;i<burst_len;i++) usbBuf[i+1] = 0;
    try {
        usb.write(usbBuf, 0, burst_len+1); //Send a READ command
        //read data
        if (burst_len != usb.read(usbBuf, 0, burst_len)) return -1;

        i = burst_len;
        while (i > 0) { //Iterate result backwards
            i--;
            d = usbBuf[i];
            if (d == 0) continue;
            j = 0;
            while ((d&0x80) == 0) {j++; d <<=1;} //Find top bit position
            if (j == 0) //Data in a single byte
                data[c++] = (byte) (d & 0x7F);
            else {//Data go across byte boundary
                data[c++] = (byte) ((  d | (((usbBuf[i-1]&0xFF)>>>(8-j))&0xFF)   )
                i--;
            }
        }
         //Checksum & length
        if (c == 5 && (data[0] == ((data[1]+data[2]+data[3]+data[4])&0x7F)))
            return   data[4] |
                   ( data[3]<<7 ) |
                   ( data[2]<<14 ) |
                   ( data[1]<<21 );
        else {
            return -1; //Fail
        }
    }catch(Exception e){
        return -1; //Fail
    }
```

■ Other contributions to the project

Apart from the above aspects, I also contributed to the unit testing of the communication channel, translated the interface into Chinese and Japanese, located an open source project for rendering SVG icons of circuit components and wrote part of the help file.

Contribution of other group members

- **Robert Duncan**

Robert did an excellent job as project manager. He set up the Google code repository, produced professional documents throughout the project timeframe as well as organized group meetings and coding sessions. Not only did he run the management work efficiently he also took up a considerable amount of coding tasks himself. He wrote the verilog exporting facilities, provided UI translation architectures, integrate my communication layer into the rest of the project as well as many other UI tasks. Without him we couldn’t have achieved what we achieve today.

- **Justus Matthiesen**

Justus was in charge of the circuit simulation algorithm and data structures. He did a great job of implementing the simulator and integrating his data structure with Chris’s UI and Robert’s verilog exporter. He was also very friendly and active and willing to discuss and contribute during our meetings. Thanks to Justus’s recommendation the JGraph library made the UI coding much more convenient.

- **Christopher Wilson**

Chris was the driving force behind our UI design. He dealt with circuit components and wire connections on top of JGraph. He achieved a lot within the short timeframe of the project. He also worked closely with Justus to integrate Justus’s data structure with his JGraph hierarchy.

- **David Weston**

David was designing the UI together with Chris. He was implementing the general UI including toolbars and menus. The collapsible menu for circuit component library was written by him from scratch and it looks very cool. David was also willing to ask for help when he encountered problems.