Embedded Systems – CSEE 4840 – Spring 2005

Real Time Programmable Video Processor with Predictable Timing

Final Project Report

Julio A. Rios jar2153
Sharmila Gupta sg2337
Emil Nuñez ern2101

Tuesday, May 10, 2005

School of Engineering and Applied Science
Columbia University, New York

TABLE OF CONTENTS

1. OVERVIEW

The main purpose of our system is to implement the NTSC standard for a VGA text display like the video processor we used in lab 2 (opb_xsb300e_vga.vhd, Edwards 2005). However, our system performs this task through software running on a programmable processor rather than purely on custom designed hardware. The implication of implementing this application in software is that it is not specific to video signal generation, but could be reprogrammed to perform a variety of other real-time tasks. Our target application, however, is NTSC.

Due to the precise timing required by NTSC, the most important aspect of the project is to provide a processor with predictable timing, hence the name "Real Time Programmable Video Processor with Predictable Timing." The way predictable timing is accomplished in our programmable processor is by the implementation of a special "wait" instruction which allows the stalling of execution for a specified number of cycles.

Our design is implemented on the XESS XSB-300E FPGA and it uses the on-board Video DAC. The Microblaze processor was not used at all. Furthermore, the free Xilinx ISE WebPACK 7.1i and ModelSim Xilinx Edition-III 6.0a were used to synthesize and simulate the design during our implementation and debugging phase. We based our design on the open-source VHDL code for the miniMIPS processor available at opencores.org. miniMIPS is an implementation of a 5 stage pipelined RISC processor.

The system is implemented as a VHDL circuit consisting of an enhanced version of the miniMIPS processor and a video controller module that provides a very primitive, but essential interface between the processor and the Video DAC. The video generation is done by executing MIPS assembly code on the processor.

This document explains the details of our design and implementation. The Appendix includes the assembly code and the VHDL modules that encompass our entire system, along with test benches we used during the implementation.

2. DESIGN

The basic approach that was followed to design the application was to emulate the behavior of the hardware-only implementation of Stephen Edwards' video controller, which is briefly summarized below. We started by describing the behavior in a more software oriented way with pseudo code. Then we started thinking of how to implement it in assembly code, which led to the development of the new "wait" instruction with the help of Professor Edwards. To support the "wait" instruction in the MIPS architecture we developed the concept of a special wait register. In addition, we developed an addressing scheme to allow us to issue video signals by writing to a set of predefined memory addresses. The addressing scheme was also enhanced to simplify access to a space in memory holding the text characters to display. Finally, we also used the addressing scheme to condense issuing of a byte of data to directly to the video controller in one instruction.

2.1 NTSC Implementation through the VGA

The video DAC provides the following ports to the VGA port:

VSYNC	vertical synchronization signal
HSYNC	horizontal synchronization signal
BLANK	blanking signal
RED(9 downto 0)	pixel red value
GREEN(9 downto 0)	pixel green value
BLUE(9 downto 0)	pixel blue value

Since our application is only for a text display, each pixel is either on, with value '0', or off, with value '1'. Therefore, the RED, GREEN, and BLUE bits are either all on ("1111111111"), meaning a white pixel, or all off ("0000000000"), meaning a black pixel. Any other pair of colors can be used by changing the constants to each of the three color signals, but we chose to stick to the simple terminal colors for simplicity and clarity.

We also kept the regular 640 x 480 pixel resolution size at 60Hz, as is the case in the video controller we were trying to emulate. Likewise, we kept the character resolution of 80 x 30 using the standard 8 x 16 Linux console font. Each byte of data output to the video controller displays one line of the font that represents the character that is currently being processed. The figure below exemplifies what is meant by lines of a font, A in this example. The video controller is to handle output of each individual pixel sequentially once a byte is loaded.

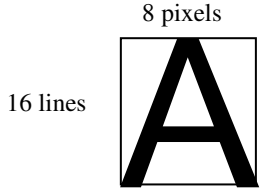


Figure 1. Sample font mapping.

The following diagrams are taken from the presentation of the hardware video controller, available at http://www1.cs.columbia.edu/~sedwards/classes/2005/4840/video-controller.pdf. The synchronization and blanking signals are active low. The BLANK signal is unasserted when both HACTIVE and VACTIVE are asserted. The BACK_PORCH and FRONT_PORCH regions are those where no signals are asserted, but are still necessary for correct synchronization.

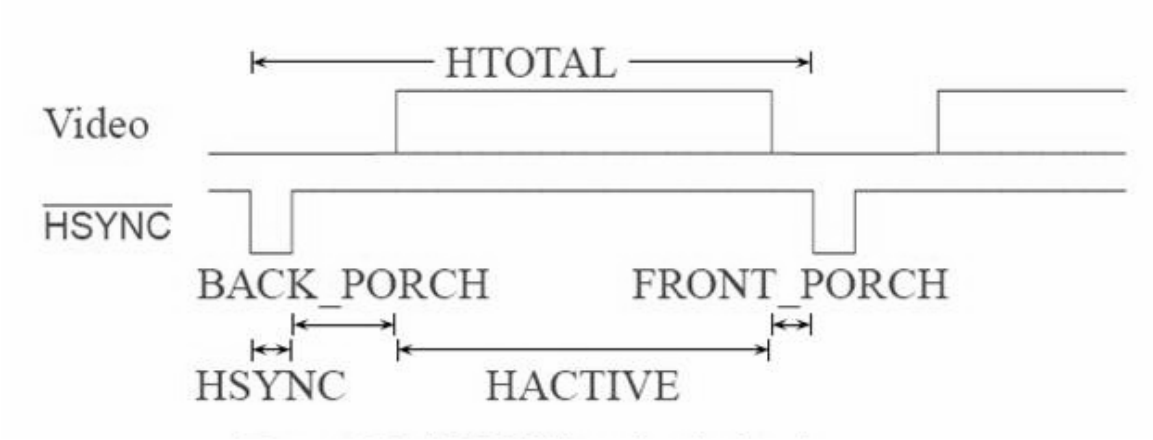


Figure 2. NTSC horizontal timing.

HSYNC	96 pixels
BACK_PORCH	48
HACTIVE	640
FRONT_PORCH	16
HTOTAL	800

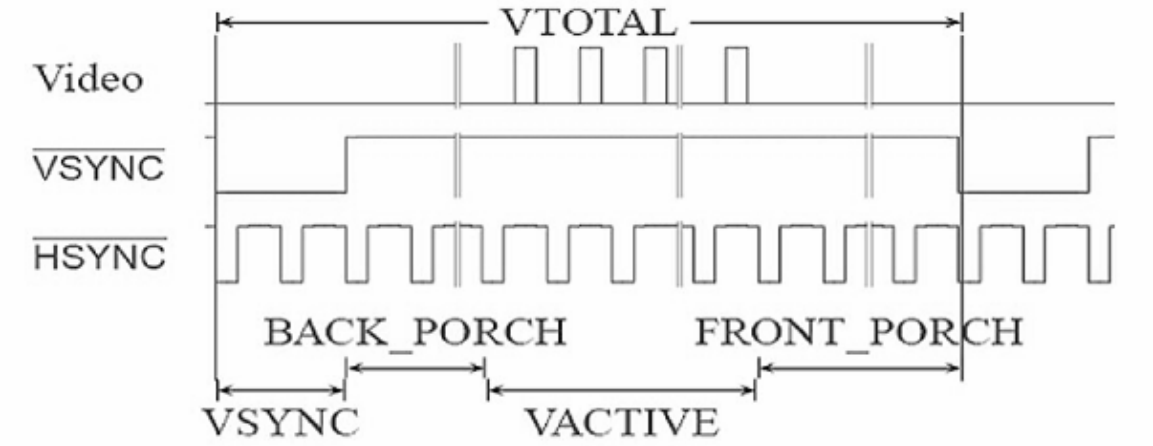


Figure 3. NTSC vertical timing.

VSYNC	2 lines
BACK_PORCH	33
VACTIVE	480
FRONT_PORCH	10
VTOTAL	525

Thus, if we consider all the clock cycles as comprising the screen we get the following "screen".

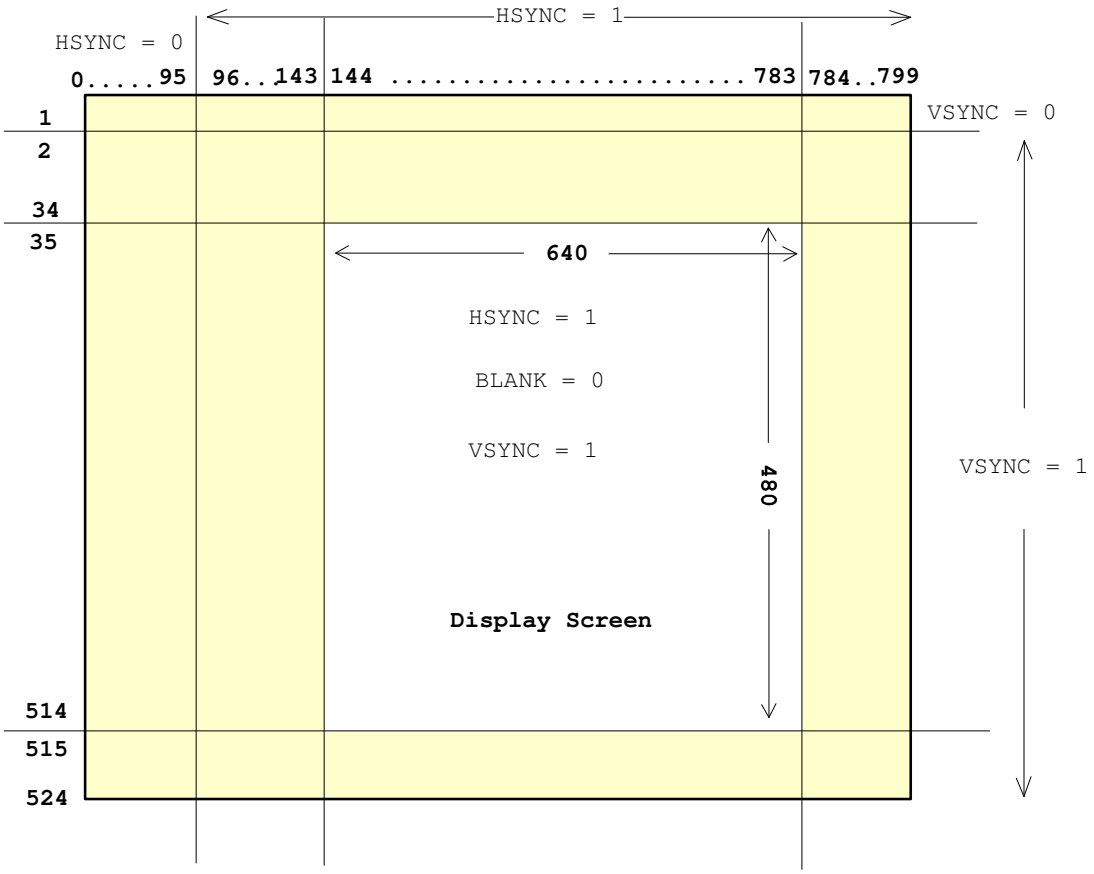


Figure 4. Clock cycles view.

In Figure 4, we see that the viewable portion is only displayed for a portion of the time, in the section that is white. Each field takes 420K cycles (800 x 525 = 420,000). At 60 Hz, the screen is refreshed 60 times per second. Therefore, we need a clock frequency of 25.2 MHz (420K * 60Hz). As a side note, our design was very large so we had to turn off some performance optimizations to get it to fit on the FPGA by setting the option –r in the map program section in fast_runtime.opt (see Appendix). With the help of Marcio Buss, we made a change to the clock generator to get this frequency (see clkgen.v in Appendix for the updated version of the clock generator).

2.2 Pseudo Code

Translating the behavior of the video controller, we arrived a the following pseudo code description.

```
char screen[80*30];              // special 1.5KB memory
byte font[96*16];                // special 2.4KB memory
bool HSYNC, VSYNC, BLANK;        // video signals
sr LOAD;             // special shift register that actually outputs
bytes to display
while (1) {
   assert HSYNC for 96 cycles every 800 cycles
   assert VSYNC for 2000 cycles every 420,000 cycles
   for(int row=0; row<30; row++) {
         for(int line=0; line<16; line++) {
               assert BLANK for entire next loop
               for(int col=0; col<80; col++) {
                     char c = screen[row*80+col];
                     byte d = font[c*16+line];
                     LOAD = d [every 8 cycles]
               } all signals deasserted for 16 cycles
         }} all signals except HSYNC deasserted for 8000 cycles
   }
```

2.3 Wait Instruction and Wait Register

Our solution involves the addition of a "wait" instruction to the MIPS assembly language. This new instruction is designed to halt the execution of a program by a specified number of clock cycles to provide precise real-time capability.

The main function of the "wait" instruction is to set a timer that counts down to zero. To accomplish this task we use a hybrid of a register and a down-counter. Initially, the register is initialized to zero. Issuing the first "wait" instruction will cause a value to be written to the register. Every subsequent clock cycle will decrement the value in the register until it hits zero again. If another "wait" instruction is issued before the register reaches zero, execution stops until it does. Then, execution continues as normal.

Basically, on a "wait" instruction, there is a check to see if the register is zero. If it is, continue. If it is not, wait until it gets to zero, and then continue.

The "wait" instruction has the form:

```
wait [wait-register], [value]
```

To demonstrate the use of the "wait" instruction, let us focus on the innermost loop of the pseudo code which loads the video controller's output shift register with a new byte of data every 8 cycles.

```
for(int col=0; col<80; col++) {
      char c = screen[row*80+col];
      byte d = font[c*16+line];
      LOAD = d [every 8 cycles]
}
```

The assembly code to implement this loop is the following.

```
                                ; assume i already holds row*80
                                ; assume line holds the right value
                                ; the first byte has been handled specially (BLANK)
            addi $4, $0, 0      ; col=0
COLS:       addi $3, $3, 1      ; i++
            lw   $8, CHARRAM($3)  ; c = current character
            wait $1, 8          ; wait 8
            sw $8, LOAD($2)     ; LOAD(c,line)
            addi $4, $4, 1      ; col++
            slti $5, $4, 80     ; if (col <80)
            bne $5, $0, COLS    ; loop again
```

In this example, the "wait" instruction sets the register to 8 and begins to count down. It takes seven clock cycles to loop back to the "wait" instruction again. At that point, the register holds the value one. Execution stalls for one cycle, and then continues. The full assembly code can be found in the Appendix.

Initially, we had thought that we would need a few wait registers to account for each of the video signals. However, all the delays are contingent upon each other and some signals get set simultaneously, such as VSYNC and HSYNC. Therefore, we were able to use the same wait register to implement the video generation application. In other real-time applications, however, that might conceivably depend on delays that are independent of each other might require more than one wait register. The addition of more of these registers is straightforward and merely requires the instantiation of more than one of these components. For simplicity, we only implemented one.

2.4 Addressing Schemes

Issuing signals to the video DAC through assembly code required designing an addressing scheme in the bus controller and a way to handle issuing the signals in a video controller. The address space we used (32 bits) was much larger than we needed to reference the actual memory we used. Therefore, it was simple to assign high addresses to issue signals to the video controller. Issuing a "store word" instruction to one of these locations issues the appropriate signal to the video controller. We simplified the assembly code by handling these signals as switches in the video controller. Basically, issuing a signal from the bus controller to the video controller for one clock cycle turns on the corresponding signal to the video DAC. Issuing the signal again later, after the required delay, turns it off, and vice versa. The video controller is described in detail later, in the implementation section.

For reasons of implementation, the Character RAM and the Font RAM were also given special addresses. This is discussed later, in the implementation section. In addition to these special addresses, we also implemented a special way to reference the Font RAM and issue a LOAD signal in one instruction. It is necessary to have the font byte ready in the same cycle as the load, but if we had to load the byte from a register after loading the register with the byte from memory, we would have spent more than 8 cycles in the inner loop. The solution is clever and it basically involves using all the parts of the "store word" instruction:

```
SW rt,offset(base)
```

In general, this instruction stores the contents of register rt into the memory location specified by [offset + value of register base]. For our special load instruction, `offset` is the special location that

flags for a LOAD to video, `base` holds the value of the current line (a value between 0 and 15) and `rt` holds the value of the character we want to index. Since `base` gets added to `offset`, we get the value of the line by referencing the least significant 4 bits of the address, and use two levels of indexing on the Font RAM to get the right byte to send to the video controller. Hence, the bus controller outputs directly from the Font RAM to the video controller in the same cycle that it issues the LOAD signal.

2.5 Overall System

The general schematic of the system is shown in Figure 5 and is explored further in the implementation section.

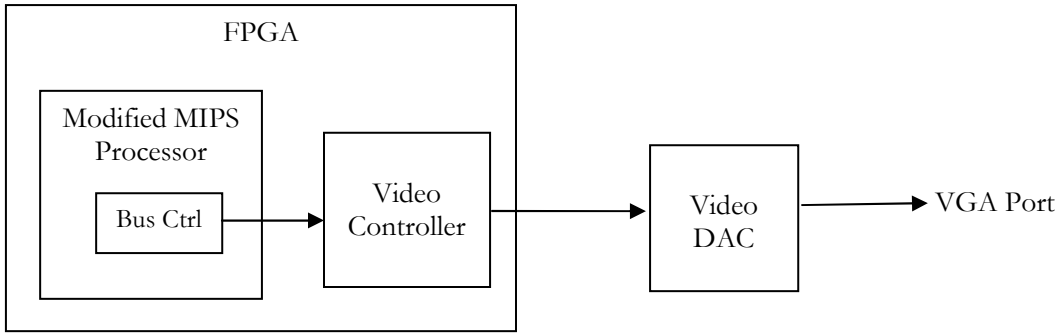


Figure 5. System Block Diagram

3. IMPLEMENTATION

3.1 Development

The final implementation evolved through several alterations that were made on the system design. We shall go through a brief overview of a few of the designs that were partially implemented before arriving at the final implementation.

The first challenge to get the design working was implementing a RAM, which the miniMIPS does not supply. Initially, the XSB-300E on-board SRAM and the Video DAC were used indirectly by the MIPS processor referenced as peripherals through the On-Chip Peripheral Bus (OPB) alongside the Microblaze. Three modifications were done on this design. First, SRAM was replaced with BRAMs internal to our system design, but external to the processor, making memory communication simpler and faster. Second, a protocol was developed for the MIPS processor to communicate directly with the RAM and the video controller eliminating the need for using the OPB Bus and therefore, disregarding the Microblaze as the bus arbiter.

The problem faced in implementing this design was that removing the Microblaze from the system synthesis was not as straight forward as one would like. Until that point, we had simply used supplied project folders to base our design. These folders included a Makefile which heretofore had been a virtual black box to us. Taking out the Microblaze required understanding the steps undertaken by the Makefile to arrive at a bit file to download on the board. More specifically, the notion of a MERGED_BITFILE was banished and we only deal with FPGA_BITFILE, or "system.bit", which is what we actually downloaded on the board.

The next design that was implemented had two separate controller modules: the RAM controller and the video controller. Each component acted as a bridge between the modified MIPS processor and the RAM and video DAC respectively. In this scenario, to reference the RAM, there was a need to follow a protocol defined in the miniMIPS bus control module that required more than one clock cycle for any memory access. Also, the stalling of execution using wait registers was designed as a separate module within the processor. The wait register, as a new entity inside the complicated pipelined architecture of the miniMIPS presented connection challenges that were difficult to overcome. Hence, what was needed was a redesign that required the least change to the already existing structure.

In our final implementation, the BRAMs reside inside the bus control unit of the MIPS processor (bus_ctrl.vhd). Basically, the old RAM communication protocol was scrapped for a more direct method of data access from the BRAMS. In addition, the new bus control unit also communicates directly with the video controller (outside the processor) by using special addressing. In this final design, the wait register was incorporated as a part of the register file itself (banc.vhd). Now, the wait register is written by issuing a "store word" instruction on register 31 (the last register). This was a better design since it used the preexisting components and did not require modifying the supplied gasm miniMIPS assembler.

Finally, we arrived at our final design:

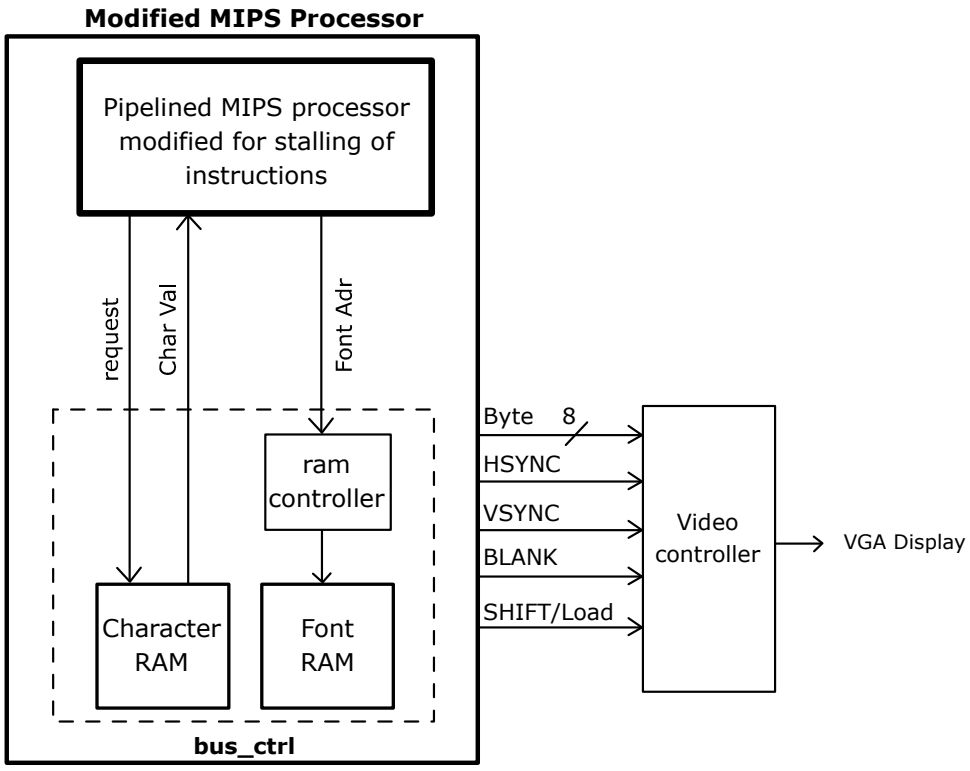


Figure 6. Final System Implementation for real time task of video generation.

3.2 Modified miniMIPS Processor

The figure below shows a simplified view of our processor, an enhanced version of the miniMIPS.

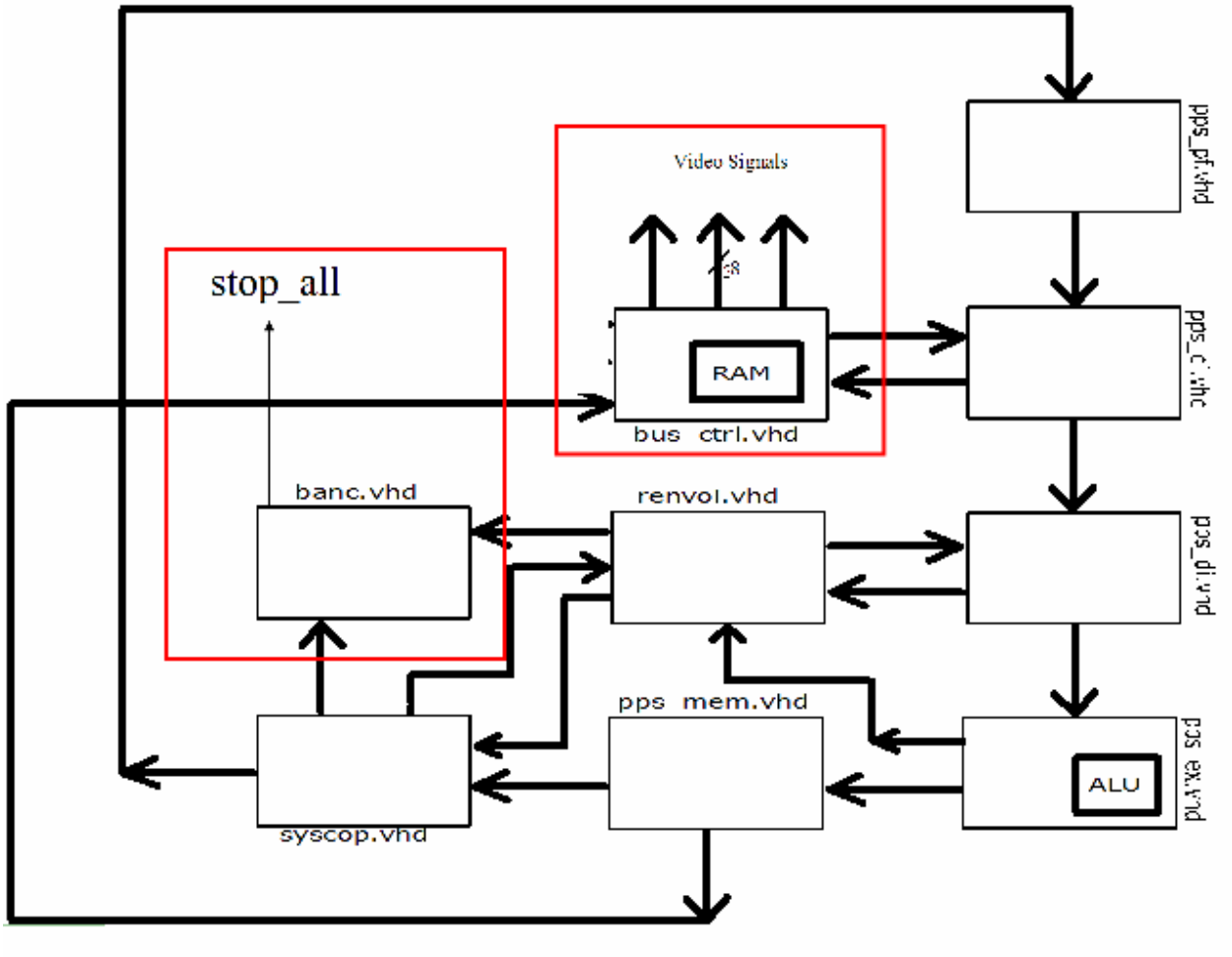


Figure 7. miniMIPS top level schematic

The miniMIPS is composed of five pipeline stages:

pps_pf	program counter stage
pps_ei	instruction extraction stage
pps_di	instruction decode stage
pps_ex	execute stage (includes ALU)
pps_mem	memory access stage

Additionally, there are other components which handle other aspects of the CPU:

banc	register file
bus_ctrl	bus controller (for RAM)
syscop	coprocessor system
renvoi	bypass unit

The only parts we changed are the bus controller and the register file. The register file includes a wait register (register 31) and issues a stop_all signal to stall the pipeline. The bus controller initially implemented a protocol to communicate with an external RAM. Now, it includes a RAM, implemented by BRAMS and outputs signals to the video controller. The bus controller also implements complex addressing schemes to allow the processor to implement video generation. In addition, the bus controller takes the stop_all signal which inhibits the sending of video signals.

3.3 Bus Controller

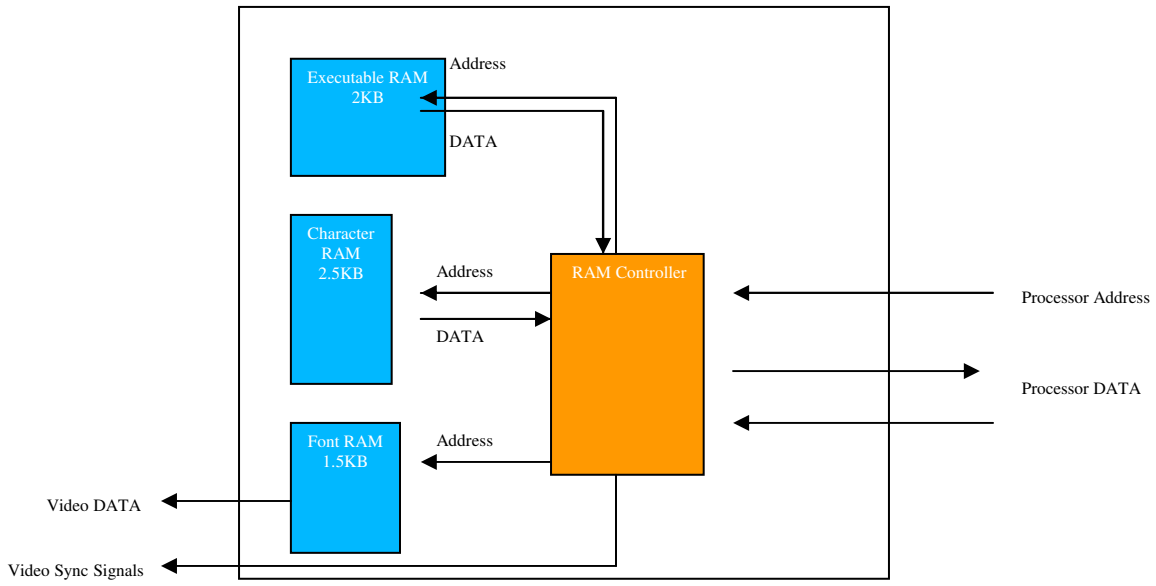


Figure 8. Bus controller schematic

All memory (instruction data, RAM data, Character RAM, and Font RAM) were implemented using BRAM's that are made available on the FPGA. All documentation can be found in the Application Note : Using Block SelectRAM+ Memory in Spartan-II FPGAs:
http://www.xilinx.com/bvdocs/appnotes/xapp173.pdf

3.3.1 Design Steps

The MiniMIPS microprocessor has no RAM associated with it, so we had to install a RAM controller that interfaced with the bus controller on the miniMIPS. The original plan was to use the on-board RAM, communicating with our RAM controller through the (OPB). We were under the impression that since we had done this before, this would be the easiest route to take. The challenge here was to make sure that the timing constraints of the RAM matched those expected by the miniMIPS. Also a lot of time was spent on learning how to connect the miniMIPS, making sure that it communicated correctly with the Microblaze and the OPB. We realized that this was becoming too tedious, and we were spending too much time on it.

To simplify our design, we decided to implement the RAM as a grouping of BRAMS. As shown in Table 1 of page 2 of the BRAM documentation, each BRAM contains 4KB of data, which can be configured to have a different number of address rows (depth) depending on the size of the data bus. Our XC2S300 device can use up to 16 blocks for BRAM.

In order to maximize amount of RAM space, we decided to use 8 blocks of 1K x 4 in parallel. In this way, we would emulate a 1KB address space, each with 32-bit data. In the BRAM, addressing each 32-bit data corresponds to 1 address space. Therefore, the byte-addressing that the miniMIPS assumes is incorrect. We take care of this issue by taking the address from the Instruction Decode stage and shifting it right by two bits. All other memory access does not need to be shifted, for Loads and Stores deliver absolute addresses to the RAM.

Since each address space is 32 bits wide, this made it difficult to access the Character and Font address spaces, since they should only be 8 bits wide. Furthermore, there was already a template for a Font RAM with font data already placed in the BRAMs in continuous order. If we used the 8 BRAMs we would have to parse the data and place it in 4 bit parallel blocks. We were also having problems with the timing between the RAM controller and the bus controller on the miniMIPS. The miniMIPS was waiting for an ack signal that the BRAM did not provide, so we had to make it ourselves. The miniMIPS itself had a drawback that could hinder our productivity. The bus controller assumes one RAM for both data and instructions. Therefore, in the pipeline, any loads or stores could not occur on the same clock cycle as an instruction fetch. Since instruction fetches occur EVERY clock cycle, a load or store instruction would cause a one-cycle-stall. We were trying to easily predict and cut down on timing, for instruction execution. In order to combat this, we decided to utilize the dual port BRAM, half for instructions, half for data storage. Since this required a total revamping of the functionality of the bus controller, we decided (instead of redesigning both the external RAM controller AND the bus controller) to redesign the bus controller on the miniMIPS to have direct access to the BRAMs, meaning, the BRAMs were written into the bus controller. Therefore, we came up with the following schema for our final design:

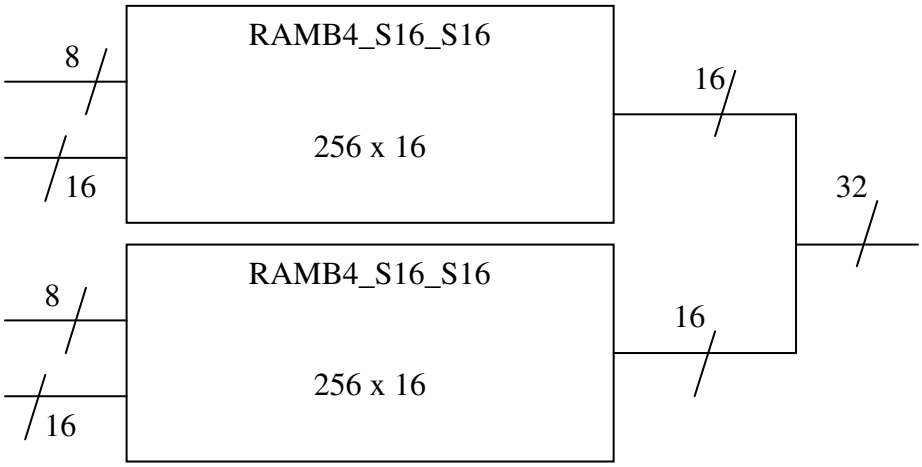


Figure 9. Parallel BRAMs for instructions and data.

As one can see, the instruction and data storage RAMs work in parallel, while the Character and Font RAMs use contiguous data. Therefore, the addressing schemes are a bit different. For the RAMB4S16_S16, if the address bus requests a pertinent memory location, both BRAM's are activated, and the output signals are concatenated, otherwise, both BRAM's output X"0000" (BRAM does this by itself when you assert the RESET signal). For the Character and Font RAMs

(both implemented using RAMB4_S8 components), all necessary data is in one address space of one block. Therefore, when a pertinent address comes in, only one of its BRAMs get their RESET signal set to LOW. All others have a high RESET, and output X"00".

Both the data storage RAM and Character RAM have to go back to the memory stage of the miniMIPS on a 32-bit bus. Therefore, the 8-bit output of the Character RAM gets padded with zero's, and is OR'd with the output of the data storage RAM. The instruction ROM (same as RAM except write-enable is tied to ground) outputs are directly connected to the Instruction Decode stage of the miniMIPS pipeline, while the Font ROM outputs are directly connected to our video controller. This was explained in detail in the Addressing Schemes section of Design.

3.3.2 Address Map

For ease of decoding addresses and programming, the following addresses were mapped to data in the RAMs. (Note: Since our RAM is substantially small, we notate with the lower 16 bits of address. The upper 16 bits are X"0000")

Instruction ROM	0000 - 007F	: 128 bytes
Data Storage RAM	0080 - 00FF	: 128 bytes
Character RAM	1000 - 19FF	: 2.5 Kbytes
Font ROM	2000 - 25FF	: 1.5 Kbytes

The Font and Character ROMs were specifically sized for their purposes as stated in the section about NTSC. The Character ROM holds the content of the 640x480 screen. Each 8-bit location in the character RAM refers to the ASCII value of the character at the corresponding point on the screen. Each character is listed in the Font ROM, ordered by ASCII value. Each character appears on the screen using an 8-bit by 16-row pixel matrix. Therefore, in the Font ROM, there is a new character every 16 bytes. We had to develop a way to index the RAM so we could get the correct byte value every clock cycle.

In order to write a byte to Video, we would have to:

1) Load from Character ROM (i.e. get ASCII Value)
2) Load from Font ROM using ASCII value and line value as an index
3) Write font data to video

Once again, to cut down on instruction count, steps 2 and 3 were combined in a clever manner, as was explained in the Design section.

3.3.3 "New" Write to Video Instruction

We devised a "new" instruction without having to change the instruction set architecture. This instruction would access the Font RAM and immediately send to video. This is why Figure 5 has the Font RAM directly connected to the Video input.

As stated earlier, the Font ROM is located in address space 2000-25FF. Since each character takes up 16 rows of 8-bits in the Font ROM, we can index the heading each character (i.e. the first byte that corresponds to a specific character), and simply traverse through the following 15 address spaces. To do this, we would need:

1) A valid address space to access the ROM (i.e. 2000 – 25FF). This enables the ROM
2) Use the ASCII value and multiply it by 16 (left shift 4 bits). This gives us the header (beginning) of the character we want
3) Use the line value as an offset to the character header

Since the SW (Store Word) instruction in miniMIPS provides the bus controller with 3 values, we could make use of this. Here is how the SW instruction works:

Syntax: SW R1, offset(R2)

This instruction instructs the processor to get data from from Register 2 (R2) and added to the offset. This is the target address and gets input to the bus controller address bus. The data is read from Register 1 (R1) and is sent to the data bus. The memory stage makes a RAM request and turns the r_w_to_ram (WRITE_ENABLE) signal on.

In normal operation, the RAM writes data from R1 and stores it into the calculated target address. However, we can "trick" the memory into doing whatever we want.

Assume : R1 holds the ASCII value previously loaded from Character RAM
Let's assume R1 = X"0000007F" (the highest ascii value)
R2 holds holds the current loop index value. This value can only be from 0 to 15, as those are the constraints of the loop.
Let's assume R2 = X"0000000F" (the highest loop value)

Note: 8192 is the decimal value for (hex) X"2000". Our assembler only accepts decimal offset values.

Instruction: SW R1, 8192(R2)

This sends the bus controller: X"0000007F" on the data bus, and X"0000200F" on the address bus. This is the highest address location that the Font ROM should receive. Therefore we can enable it (actually, set its RESET signal to LOW) when the addr(31 downto 4) = X"0000200" AND write_enable is on. The bus controller now knows that we have sent a special SW instruction. Firs, the Font ROM does not include the first 32 ASCII values, so ASCII value #32 (space bar) is mapped at address X"00002000". So the first thing we do is subtract X"20" from the data bus. Shift this value by 4 bits and we get X"000005F0", which is the header (beginning of character: "DEL"). Now we add this to the address bus. This gives us a new value of X"000025FF". We can now use this new "address" to locate the byte of the font we want. Our test case used the last byte of the last character, so therefore corresponds to the last available byte in the font ROM. The

write_enable signal of the font ROM is tied to ground, so everything is a read. The outputs are connected to the video controller.

3.4 Video Controller

The video controller is a bridge between our processor and the Video DAC, residing on the XSB Board, which awaits an NTSC video signal for the VGA Display.

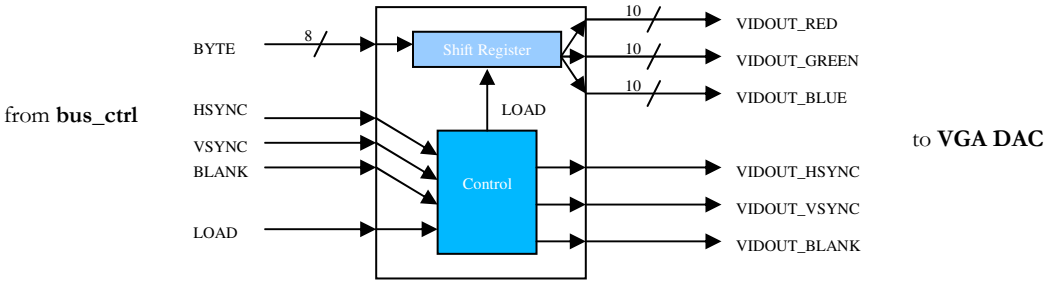


Figure 10. Video controller schematic

As shown in Figure 10, the video controller receives input signals from the bus controller (bus_ctrl.vhd). These are *HSYNC*, *VSYNC*, *BLANK*, *loadNshift* and the byte input holding the first 8 font pixels of the first character.

All these signals are initially asserted, (i.e. they are inactive). Every time a *HSYNC*, *VSYNC* or *BLANK* signal is received from the **bus_ctrl**, with every positive edge, the outputs, *Vid_Hsync*, *Vid_Vsync* and *Vid_Blank* are de-asserted (made active) and re- asserted when the next positive edge is triggered. This is illustrated in the timing diagram below. However *loadNshift* holds an initial value of '0' which implies that most significant data bit is continuously shifting out of the shift register unless asserted. At this point *byte_data* is loaded.

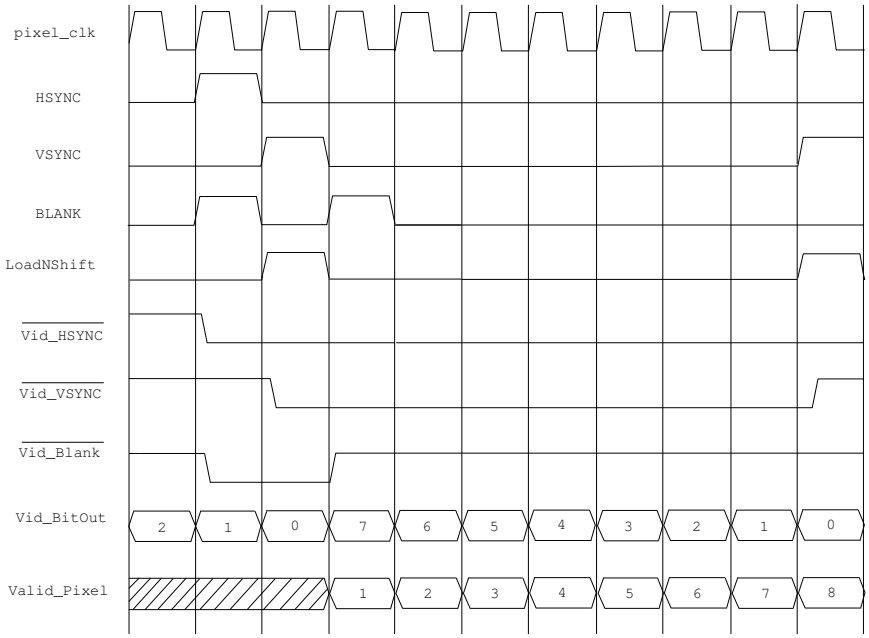


Figure 11. Video controller sample test bench

The signals follow the following Mealy state pattern as shown in the figure below.

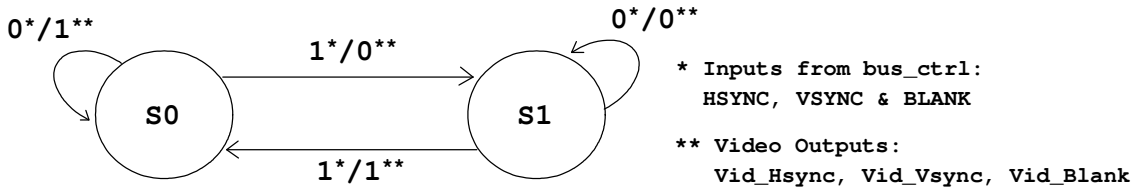


Figure 12. Mealy state machine for video controller.

These output signals are directly mapped from the pins in the FPGA in the system.ucf file to the VGA Connector as specified in the XSB-300E Documentation.

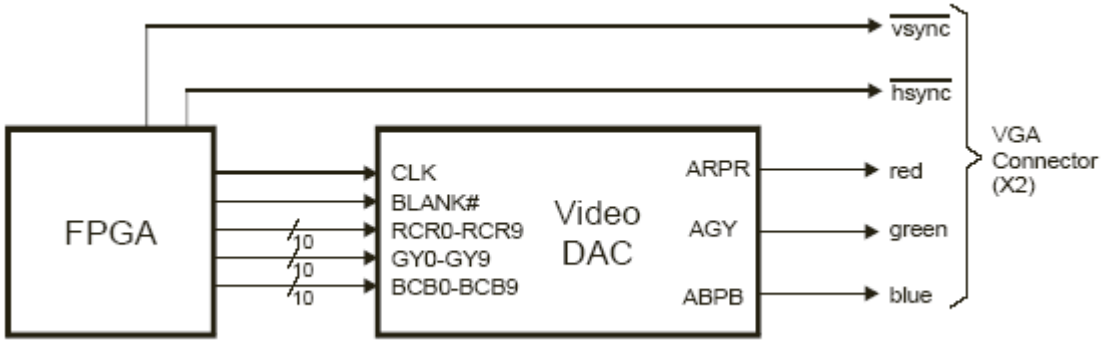


Figure 13. Video DA Converter connection to the FPGA on the XSB-300E board

3.5 Wait Register and Conclusions

The controller for the special purpose register is as follows. It is actually pretty simple, but there are many conditions due to the fact that the write was only issued for one cycle. This fact necessitates the use of an auxiliary latched signal called wr_val to store the next value to write.

```
     wr_ecr <= '1' when (cmd_ecr = '1' and
        adr_dest = 31) or  cs = '1' else
               '0';

    wr_dec <= '0' when wait_reg = X"000000" else
              '1';-- wait register controller
    process(clock,reset,wr_ecr, wr_dec)
    begin
        if clock = '1' and clock'event then
            if reset = '1' then
                wait_reg <= (others =>'0');
            elsif wr_ecr = '1' and wr_dec = '0' then
                if wr_src = '0' then
                    wait_reg <= donnee;
                else
                    wait_reg <= wr_val;
                end if;
            elsif wr_dec = '1' then
                wait_reg <= wait_reg - 1;
            end if;
        end if;
    end process;
```

The generation of the stop_all signal was very difficult to get the right and it always interfered with the correct behavior of the register itself. We never got this completely right, as can be seen by the commented sequential behavior.

```
    -- handle stop_all condition and setup new value to write
    process(reset,wr_ecr, wr_dec)
    begin
--        if clock = '1' and clock'event then
            if reset = '1' then
                cs <= '0';
                stop_all <= '0';
                wr_val  <= X"00000000";
            else
                case cs is
                    when '0' =>
                        wr_src <= '0';  -- store input
                        if wr_ecr = '1' and wr_dec = '1' then
                            wr_val <= donnee;
                            stop_all <= '1';
                            cs <= '1';
                        end if;
                    when '1' =>
                        wr_src <= '1';  -- store register
                        if wr_dec = '0' then
                            stop_all <= '0';
                            cs <= '0';
                        end if;
                    when others =>
                        stop_all <= '0';
                        cs <= '0';
                end case;
            end if;
--        end if;
    end process;
```

The wait register is seemingly the simplest component logically speaking, but it proved to be the hardest to implement and, eventually, the reason for why our implementation did not work. All the components, including the wait register worked exactly as they should independently; we simulated everything by itself several times using countless test benches that have since been lost in the chaos of the massive redesigns of the last few days (a few have been found and are in the Appendix).

We spent weeks trying to figure out how to synthesize our design and put it on the board. Of course it did not work at all, because with video generation it either works perfectly or it does not work at all. It was not until very late in the term that we were able to put everything together, which is when we realized that there was a fatal design flaw we had overlooked due to the use of a pipelined processor. When the wait register issues the stop_all signal, it halts execution at the **next** instruction.

This fact has a few implications for our implementation. If the next instruction is a "store word" to a video signal, it asserts the video output while it is waiting, which it should not; we were able to fix this problem by routing the stop_all signal to the bus controller causing it to not output anything to the video when stop_all is asserted. Another problem is that the previous instruction is lost so the new value to be written to the register is needs to be stored in an auxiliary latch. This seems like a simple enough problem to correct, but after about 20 redesigns of the wait register, the correct behavior could not be reached and after two days of no sleep we gave up and decided to present what we have since it was already two days late. This is really an interesting project and it would be great to get it to work. At least one of us will work on getting it to work after this term is over. It is definitely doable without the pressure of deadlines and exams that this past week has brought. At least we can not say we did not try.

4. PROJECT EXECUTION

4.1 Work Distribution

Job Description	*Team Member*
System Design	Julio
RAM Controller	Emil
Modified miniMIPS design	Julio
Video Controller	Sharmila
Modified register file	Julio
Compiling	Emil, Julio, Sharmila
Documentation	Emil, Julio, Sharmila

4.2 Advice for Future projects

One useful tip is that, to compile a project with the package, the package file must be listed first in the corresponding PAO file. It is also useful to try to understand the Makefile completely before trying to implement a custom design.

There are many things we could have done to cut down on the amount of time and stress that inevitably took for us to get through this project. Of course, everything is easier the second time around. We learned a great deal about VHDL semantics, implementation, and synthesis, but we could have gone about it in a more organized manner. We would encourage groups to spend as much time as possible on the initial design to avoid the time consuming and stressful redesigns that our project undertook. Inevitably, nothing is going to work perfectly the first time, but starting out with a complete and sound design will make debugging infinitely easier, rather than get to a point where you realize you are doing things completely wrong and going back to the drawing board several times or deciding you will just make do with what you have. Had we explored all our options, we would have not chosen a pipelined processor for an application that is not so much concerned with speed as much as correctness. But alas, it was definitely a good learning experience.

4.3 Acknowledgement

We would like to thank Stephen Edwards and Marcio Buss for their patience and advice.

APPENDIX

SYSTEM FILES

system.mhs	.
system.mss	.
Makefile	.
system.ucf	./data/
bitgen.ut	./etc/
fast_runtime.opt	./etc/
rtproc_v2_1_0.mpd	./pcores/rtproc_v1_00_a/data/
rtproc_v2_1_0.pao	./pcores/rtproc_v1_00_a/data/
clkgen_v2_1_0.mpd	./pcores/clkgen_v1_00_a/data/
clkgen_v2_1_0.pao	./pcores/clkgen_v1_00_a/data/

ASSEMBLY CODE

./asm/rtproc.lst

HDL CODE

pack_mips.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
rtproc.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
minimips.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
pps_pf.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
pps_ei.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
pps_di.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
pps_ex.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
pps_mem.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
renvoi.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
syscop.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
alu.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
banc.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
bus_ctrl.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
video_controller.vhd	./pcores/rtproc_v1_00_a/hdl/vhdl
clkgen.v	./pcores/clkgen_v1_00_a/hdl/verilog

TEST BENCHES

video_controller_tbw.vhd
bus_ctrl_tbw.vhw
banc_tbw.vhw

SIMULATION WAVEFORMS

SYSTEM FILES

system.mhs

```
# Parameters
PARAMETER VERSION = 2.1.0

# Global Ports

PORT FPGA_CLK1 = FPGA_CLK1, DIR = IN

PORT VIDOUT_CLK = VIDOUT_CLK, DIR = OUT
PORT VIDOUT_HSYNC_N = VIDOUT_HSYNC_N, DIR = OUT
PORT VIDOUT_VSYNC_N = VIDOUT_VSYNC_N, DIR = OUT
PORT VIDOUT_BLANK_N = VIDOUT_BLANK_N, DIR = OUT
PORT VIDOUT_RED = VIDOUT_RED, DIR = OUT, VEC = [9:0]
PORT VIDOUT_GREEN = VIDOUT_GREEN, DIR = OUT, VEC = [9:0]
PORT VIDOUT_BLUE = VIDOUT_BLUE, DIR = OUT, VEC = [9:0]
PORT BAR_LED = BAR_LED, DIR = OUT, VEC = [9:0]

# Useful trick: put the IP you are developing first in this list
# so that it is compiled (and therefore may fail) before anything else

# Real-time MIPS Processor
BEGIN rtproc
 PARAMETER INSTANCE = mips
 PARAMETER HW_VER = 1.00.a
 PORT sys_clk = pixel_clock
 PORT pixel_clock = pixel_clock
 PORT sys_rst = fpga_reset
 PORT VIDOUT_CLK = VIDOUT_CLK
 PORT VIDOUT_RED = VIDOUT_RED
 PORT VIDOUT_GREEN = VIDOUT_GREEN
 PORT VIDOUT_BLUE = VIDOUT_BLUE
 PORT VIDOUT_BLANK_N = VIDOUT_BLANK_N
 PORT VIDOUT_HSYNC_N = VIDOUT_HSYNC_N
 PORT VIDOUT_VSYNC_N = VIDOUT_VSYNC_N
 PORT BAR_LED = BAR_LED
END

# Clock divider to make the whole thing run

BEGIN clkgen
 PARAMETER INSTANCE = clkgen_0
 PARAMETER HW_VER = 1.00.a
 PORT FPGA_CLK1 = FPGA_CLK1
 PORT sys_clk = sys_clk
 PORT pixel_clock = pixel_clock
 PORT fpga_reset = fpga_reset
END
```

system.mss

```
PARAMETER VERSION = 2.2.0
PARAMETER HW_SPEC_FILE = system.mhs

# Assign a null driver for hardware that doesn't need one,
# mostly to disable warnings

BEGIN DRIVER
 PARAMETER HW_INSTANCE = mips
 PARAMETER DRIVER_NAME = generic
 PARAMETER DRIVER_VER = 1.00.a
END
```

Makefile

```
# Makefile for RTPROC

SYSTEM = system

NETLIST = implementation/$(SYSTEM).ngc

# Bitstreams for the FPGA

FPGA_BITFILE = implementation/$(SYSTEM).bit

MHSFILE = $(SYSTEM).mhs
MSSFILE = $(SYSTEM).mss

FPGA_ARCH = spartan2e
DEVICE = xc2s300epq208-6

LANGUAGE = vhdl
PLATGEN_OPTIONS = -p $(FPGA_ARCH) -lang $(LANGUAGE)

# Paths for programs

XILINX = /usr/cad/xilinx/ise6.2i
ISEBINDIR = $(XILINX)/bin/lin
ISEENVCMDS = LD_LIBRARY_PATH=$(ISEBINDIR) XILINX=$(XILINX)\ PATH=$(ISEBINDIR):$(PATH)

XILINX_EDK = /usr/cad/xilinx/edk6.2i
EDKBINDIR = $(XILINX_EDK)/bin/lin
EDKENVCMDS = LD_LIBRARY_PATH=$(ISEBINDIR):$(EDKBINDIR) XILINX=$(XILINX)\
XILINX_EDK=$(XILINX_EDK) PATH=$(ISEBINDIR):$(EDKBINDIR):$(PATH)

XESSBINDIR = /usr/cad/xess/bin

# Executables

PLATGEN = $(EDKENVCMDS) $(EDKBINDIR)/platgen
LIBGEN = $(EDKENVCMDS) $(EDKBINDIR)/libgen

XST = $(ISEENVCMDS) $(ISEBINDIR)/xst
XFLOW = $(ISEENVCMDS) $(ISEBINDIR)/xflow
BITGEN = $(ISEENVCMDS) $(ISEBINDIR)/bitgen
DATA2MEM = $(ISEENVCMDS) $(ISEBINDIR)/data2mem
XSLOAD = $(XESSBINDIR)/xsload
XESS_BOARD = XSB-300E

# External Targets

all :
@echo "Makefile to build a Microprocessor system :"
@echo "Run make with any of the following targets"
@echo "  make netlist  : Generates the netlist for this system ($(SYSTEM))"
@echo "  make bits     : Runs Implementation tools to generate the bitstream"
@echo "  make download : Downloads the bitstream onto the board"
@echo "  make netlistclean: Deletes netlist"
@echo "  make hwclean  : Deletes implementation dir"
@echo "  make libsclean: Deletes sw libraries"
@echo "  make programclean: Deletes compiled ELF files"
@echo "  make clean    : Deletes all generated files/directories"
@echo " "
@echo "  make <target> : (Default)"
@echo "      Creates a Microprocessor system using default initializations"
```

```
@echo "      specified for each processor in MSS file"


bits : $(FPGA_BITFILE)

netlist : $(NETLIST)

libs : $(LIBRARIES)

clean : hwclean libsclean programclean
	rm -f bram_init.sh platgen.log platgen.opt
	rm -f _impact.cmd xflow.his

hwclean : netlistclean
	rm -rf implementation synthesis xst hdl
	rm -rf xst.srp $(SYSTEM)_xst.srp

oneclean :
	rm -rf $(NETLIST)
	@echo "Now rm other .ngc files in implementation/ and implentation/cache"

netlistclean :
	rm -f $(FPGA_BITFILE) $(MERGED_BITFILE) \
	  $(NETLIST) implementation/$(SYSTEM)_bd.bmm


#
# Hardware rules
#

# Hardware compilation : optimize the netlist, place and route

$(FPGA_BITFILE) : $(NETLIST) \
		etc/fast_runtime.opt etc/bitgen.ut data/$(SYSTEM).ucf
	cp -f etc/bitgen.ut implementation/
	cp -f etc/fast_runtime.opt implementation/
	cp -f data/$(SYSTEM).ucf implementation/$(SYSTEM).ucf
	$(XFLOW) -wd implementation -p $(DEVICE) -implement fast_runtime.opt \
		$(SYSTEM).ngc
	cd implementation; $(BITGEN) -f bitgen.ut $(SYSTEM)

# Hardware assembly: Create the netlist from the .mhs file

$(NETLIST) : $(MHSFILE)
	$(PLATGEN) $(PLATGEN_OPTIONS) -st xst $(MHSFILE)
	$(XST) -ifn synthesis/$(SYSTEM)_xst.scr

# Download the files to the target board

download : $(FPGA_BITFILE)
	$(XSLOAD) -fpga -b $(XESS_BOARD) $(FPGA_BITFILE)
```

system.ucf

```
#net sys_clk period = 80.000;
net pixel_clock period = 40.000;

#net FPGA_RST  loc="p109";
net FPGA_CLK1 loc="p77";

net BAR_LED<0> loc="p83";
net BAR_LED<1> loc="p84";
net BAR_LED<2> loc="p86";
net BAR_LED<3> loc="p87";
net BAR_LED<4> loc="p88";
net BAR_LED<5> loc="p89";
net BAR_LED<6> loc="p93";
net BAR_LED<7> loc="p94";
net BAR_LED<8> loc="p140";
net BAR_LED<9> loc="p146";

net VIDOUT_CLK loc="p23";
net VIDOUT_BLANK_N loc="p24";
net VIDOUT_HSYNC_N loc="p8";
net VIDOUT_VSYNC_N loc="p7";

net VIDOUT_RED<0> loc="p41";
net VIDOUT_RED<1> loc="p40";
net VIDOUT_RED<2> loc="p36";
net VIDOUT_RED<3> loc="p35";
net VIDOUT_RED<4> loc="p34";
net VIDOUT_RED<5> loc="p33";
net VIDOUT_RED<6> loc="p31";
net VIDOUT_RED<7> loc="p30";
net VIDOUT_RED<8> loc="p29";
net VIDOUT_RED<9> loc="p27";

net VIDOUT_GREEN<0> loc="p9" ;
net VIDOUT_GREEN<1> loc="p10";
net VIDOUT_GREEN<2> loc="p11";
net VIDOUT_GREEN<3> loc="p15";
net VIDOUT_GREEN<4> loc="p16";
net VIDOUT_GREEN<5> loc="p17";
net VIDOUT_GREEN<6> loc="p18";
net VIDOUT_GREEN<7> loc="p20";
net VIDOUT_GREEN<8> loc="p21";
net VIDOUT_GREEN<9> loc="p22";

net VIDOUT_BLUE<0> loc="p42";
net VIDOUT_BLUE<1> loc="p43";
net VIDOUT_BLUE<2> loc="p44";
net VIDOUT_BLUE<3> loc="p45";
net VIDOUT_BLUE<4> loc="p46";
net VIDOUT_BLUE<5> loc="p47";
net VIDOUT_BLUE<6> loc="p48";
net VIDOUT_BLUE<7> loc="p49";
net VIDOUT_BLUE<8> loc="p55";
net VIDOUT_BLUE<9> loc="p56";
```

bitgen.ut

```
-g DebugBitstream:No
-w
-g Binary:no
-g Gclkdel0:11111
-g Gclkdel1:11111
-g Gclkdel2:11111
-g Gclkdel3:11111
-g ConfigRate:4
-g CclkPin:PullUp
-g M0Pin:PullUp
-g M1Pin:PullUp
-g M2Pin:PullUp
-g ProgPin:PullUp
-g DonePin:PullUp
-g TckPin:PullUp
-g TdiPin:PullUp
-g TdoPin:PullUp
-g TmsPin:PullUp
-g UnusedPin:PullDown
-g UserID:0xFFFFFFFF
-g StartUpClk:CClk
-g DONE_cycle:4
-g GTS_cycle:5
-g GSR_cycle:6
-g GWE_cycle:6
-g LCK_cycle:NoWait
-g Security:None
-g DonePipe:No
-g DriveDone:No
```

fast_runtime.opt

```
FLOWTYPE = FPGA;
############################################################
## Filename: fast_runtime.opt
##
## Option File For Xilinx FPGA Implementation Flow for Fast
## Runtime.
##
## Version: 4.1.1
## $Header: /devl/xcs/repo/env/Jobs/MDT/sw/IDE/data/fast_runtime.opt,v 1.1 2002/08/07
18:44:47 akasat Exp $
############################################################
#
# Options for Translator
#
# Type "ngdbuild -h" for a detailed list of ngdbuild command line options
#
Program ngdbuild
-p <partname>;         # Partname to use - picked from xflow commandline
-nt timestamp;         # NGO File generation. Regenerate only when
                       # source netlist is newer than existing
                       # NGO file (default)
#-bm <design>.bmm      # Block RAM memory map file
<userdesign>;          # User design - pick from xflow command line
-uc <design>.ucf;      # ucf constraints
<design>.ngd;          # Name of NGD file. Filebase same as design filebase
End Program ngdbuild

#
# Options for Mapper
#
# Type "map -h <arch>" for a detailed list of map command line options
#
Program map
-o <design>_map.ncd;      # Output Mapped ncd file
#-fp <design>.mfp;        # Floorplan file
<inputdir><design>.ngd;  # Input NGD file
<inputdir><design>.pcf;  # Physical constraints file
-r;   # Needed to make the design fit, otherwise too big
END Program map

#
# Options for Post Map Trace
#
# Type "trce -h" for a detailed list of trce command line options
#
Program post_map_trce
-e 3;                  # Produce error report limited to 3 items per constraint
#-o <design>_map.twr;  # Output trace report file
-xml <design>_map.twx;     # Output XML version of the timing report
#-tsi <design>_map.tsi; # Produce Timing Specification Interaction report
<inputdir><design>_map.ncd;  # Input mapped ncd
<inputdir><design>.pcf;      # Physical constraints file
END Program post_map_trce

#
# Options for Place and Route
#
# Type "par -h" for a detailed list of par command line options
#
Program par
-w;                 # Overwrite existing placed and routed ncd
```

```
-ol 2;              # Overall effort level
#-d 0;               # Number of delay based cleanup passes
<inputdir><design>_map.ncd;  # Input mapped NCD file
<design>.ncd;                # Output placed and routed NCD
<inputdir><design>.pcf;      # Input physical constraints file
END Program par

#
# Options for Post Par Trace
#
# Type "trce -h" for a detailed list of trce command line options
#
Program post_par_trce
-e 3;                # Produce error report limited to 3 items per constraint
#-o <design>.twr;     # Output trace report file
-xml <design>.twx;    # Output XML version of the timing report
#-tsi <design>.tsi;  # Produce Timing Specification Interaction report
<inputdir><design>.ncd;   # Input placed and routed ncd
<inputdir><design>.pcf;   # Physical constraints file
END Program post_par_trce
```

rtproc_v2_1_0.mpd

```
BEGIN rtproc, IPTYPE = PERIPHERAL, EDIF=TRUE

OPTION HDL = VHDL
OPTION IMP_NETLIST = TRUE

# Signals
PORT sys_clk = "", DIR=IN
PORT sys_rst = "", DIR=IN

PORT pixel_clock = pixel_clock, DIR=IN

PORT VIDOUT_CLK = "", DIR=OUT
PORT VIDOUT_RED = "", DIR=OUT, VEC=[0:9]
PORT VIDOUT_GREEN = "", DIR=OUT, VEC=[0:9]
PORT VIDOUT_BLUE = "", DIR=OUT, VEC=[0:9]
PORT VIDOUT_BLANK_N = "", DIR=OUT
PORT VIDOUT_HSYNC_N = "", DIR=OUT
PORT VIDOUT_VSYNC_N = "", DIR=OUT

PORT BAR_LED = "", DIR = OUT, VEC = [0:9]

END
```

rtproc_v2_1_0.pao

```
lib rtproc_v1_00_a pack_mips
lib rtproc_v1_00_a rtproc
lib rtproc_v1_00_a video_controller
lib rtproc_v1_00_a minimips
lib rtproc_v1_00_a banc
lib rtproc_v1_00_a alu
lib rtproc_v1_00_a bus_ctrl
lib rtproc_v1_00_a pps_di
lib rtproc_v1_00_a pps_ei
lib rtproc_v1_00_a pps_ex
lib rtproc_v1_00_a pps_mem
lib rtproc_v1_00_a pps_pf
lib rtproc_v1_00_a renvoi
lib rtproc_v1_00_a syscop
```

clkgen_v2_1_0.mpd

```
##############################################################################
##
## Microprocessor Peripheral Definition : generated by psfutil
##
## Template MPD for Peripheral:MicroBlaze_Brd_ZBT_ClkGen
##
##############################################################################

BEGIN clkgen ,IPTYPE = IP

## Peripheral Options
#OPTION IPTYPE = IP
OPTION HDL = VERILOG

OPTION IMP_NETLIST = TRUE


## Ports
PORT FPGA_CLK1 = "", DIR = IN , IOB_STATE = BUF

PORT sys_clk = "", DIR = OUT
PORT pixel_clock = "", DIR = OUT
PORT fpga_reset = "", DIR = OUT


END
```

clkgen_v2_1_0.pao

```
##############################################################################
###
##
## Copyright (c) 1995-2002 Xilinx, Inc.  All rights reserved.  Xilinx, Inc.
##
## MicroBlaze_Brd_ZBT_ClkGen_v2_0_0_a.pao
##
## Peripheral Analyze Order
##
##############################################################################
###


lib clkgen_v1_00_a clkgen
```

ASSEMBLY CODE WITH BIT CODE (PRODUCED BY THE GASM ASSEMBLER)

```
1                       ; Julio A. Rios
2                       ; May 2005
3                       ; rtproc.asm
4                       ;
5                       ; program code to implement NTSC video signal generation
6                       ; using wait instruction (addi $31, $0, [wait count])
7                       ; no loop for first section since there are only two iterations
8                       ;
9                       ; assembled using gasm assembler
10                      ; provided with miniMIPS package
11                      ; CVS: http://www.opencores.org/cvsweb.shtml/minimips/
12
13                          org 0
14
15  0000  20140050          addi $20, $0, 80        ; const width = 80
16  0004  201F0060      LOOP:       addi $31, $0, 96        ; wait 96
17  0008  AC004004          sw $0, 16388($0)        ; 0x4004 VHSYNC
18  000C  201F02C0          addi $31, $0, 704       ; wait 704
19  0010  AC004002          sw $0, 16386($0)        ; 0x4002 HSYNC
20  0014  201F0060          addi $31, $0, 96        ; wait 96
21  0018  AC004002          sw $0, 16386($0)        ; 0x4002 HSYNC
22  001C  201F02C0          addi $31, $0, 704       ; wait 704
23  0020  AC004002          sw $0, 16386($0)        ; 0x4002 HSYNC
24  0024  201F0060          addi $31, $0, 96        ; wait 96
25  0028  AC004004          sw $0, 16388($0)        ; 0x4004 VHSYNC
26  002C  201F02C0          addi $31, $0, 704       ; wait 704
27  0030  AC004002          sw $0, 16386($0)        ; 0x4002 HSYNC
28                      ; Vertical Back Porch
29                              ; handle first line independently
30  0034  201F0060          addi $31, $0, 96        ; wait 96
31  0038  AC004002          sw $0, 16386($0)        ; 0x4002 HSYNC
32  003C  201F02C0          addi $31, $0, 704       ; wait 704
33  0040  00000820          add $1, $0, $0 ; i = 0
34  0044  201F02C0      VBP: addi $31, $0, 704      ; wait 704
35  0048  AC004002          sw $0, 16386($0)        ; 0x4002 HSYNC
36  004C  201F0060          addi $31, $0, 96        ; wait 96
37  0050  AC004002          sw $0, 16386($0)        ; 0x4002 HSYNC
38  0054  20210001          addi $1, $1, 1 ; i++
39  0058  2822001F          slti $2,$1, 31 ; if i < 31
40  005C  1440FFFA          bne $2, $0, VBP         ;    goto VBP
41                      ; Vertical Active Region
42  0060  00000820          add $1, $0, $0 ; row=0
43  0064  00001020      ROWS:       add $2, $0, $0 ; line=0
44  0068  201F0060      LINES:      addi $31, $0, 96        ; wait 96
45  006C  AC004002          sw $0, 16386($0)        ; 0x4002 HSYNC
46  0070  201F002F          addi $31, $0, 47        ; wait 47
47  0074  AC004002          sw $0, 16386($0)        ; 0x4002 HSYNC
48  0078  00201820          add $3, $1, $0 ; i = row * 80
49  007C  00740018          mult $3, $20            ;
50  0080  00001812          mflo $3
51                      ; Begin Active Region
52                              ; handle first character independently
53                              ; because also needs to assert BLANK
54  0084  8C681000          lw $8, 4096($3)         ; c = CHARRAM[i]
55  0088  201F0008          addi $31, $0, 8         ; wait 8
56  008C  AC484000          sw $8, 16384($2)        ; 0x4000 LOAD
57                                                  ; sr = FONTRAM(c,line)
58  0090  AC004001          sw $0, 16385($0)        ; 0x4001 BLANK
59  0094  20040000          addi $4, $0, 0 ; col=0
60  0098  20630001      COLS:       addi $3, $3, 1 ; i++
61  009C  8C681000          lw $8, 4096($3)         ; c = CHARRAM[i]
62  00A0  201F0008          addi $31, $0, 8         ; wait 8
63  00A4  AC484000          sw $8, 16384($2)        ; 0x4000 LOAD
64                                                  ; sr = FONTRAM(c,line)
65  00A8  20840001          addi $4, $4, 1 ; col++
66  00AC  28850050          slti $5, $4, 80         ; if col < 80
67  00B0  14A0FFFA          bne $5, $0, COLS        ;    goto COLS
68
```

```
69  00B4  201F0001        addi $31, $0, 1       ; wait 1
70  00B8  201F0010        addi $31, $0, 16      ; wait 16
71  00BC  AC004001        sw $0, 16385($0)      ; 0x4001 BLANK
72
73  00C0  20420001        addi $2, $2, 1 ; line++
74  00C4  28450010        slti $5, $2, 16       ; if line < 16
75  00C8  14A0FFE8        bne $5, $0, LINES     ;    goto LINES
76
77  00CC  20210001        addi $1, $1, 1 ; row++
78  00D0  2825001E        slti $5, $1, 30       ; if row < 30
79  00D4  14A0FFE4        bne $5, $0, ROWS      ;    goto ROWS
80                    ; End Active Region
81
82                    ; Vertical Front Porch
83  00D8  20010000        addi $1, $0, 0 ; i = 0
84  00DC  201F0060    VFP: addi $31, $0, 96      ; wait 96
85  00E0  AC004002        sw $0, 16386($0)      ; 0x4002 HSYNC
86  00E4  201F02C0        addi $31, $0, 704     ; wait 704
87  00E8  AC004002        sw $0, 16386($0)      ; 0x4002 HSYNC
88  00EC  20210001        addi $1, $1, 1 ; i++
89  00F0  2825000A        slti $5, $1, 10       ; if i < 10
90  00F4  14A0FFC4        bne $5, $0, LOOP      ;    goto VFP
91  00F8  08000001           j LOOP
92                        hlt
93
94
95
Table des étiquettes :
VFP 00DC
COLS       0098
LINES      0068
ROWS       0064
VBP 0044
LOOP       0004
```

HDL CODE

------------------pack_mips.vhd : Package file----------------------

```
library ieee;
use ieee.std_logic_1164.all;

package pack_mips is

    -- Type signal on n bits
    subtype bus64 is std_logic_vector(63 downto 0);
    subtype bus33 is std_logic_vector(32 downto 0);
    subtype bus32 is std_logic_vector(31 downto 0);
    subtype bus31 is std_logic_vector(30 downto 0);
    subtype bus26 is std_logic_vector(25 downto 0);
    subtype bus24 is std_logic_vector(23 downto 0);
    subtype bus16 is std_logic_vector(15 downto 0);
    subtype bus8 is std_logic_vector(7 downto 0);
    subtype bus6 is std_logic_vector(5 downto 0);
    subtype bus5 is std_logic_vector(4 downto 0);
    subtype bus4 is std_logic_vector(3 downto 0);
    subtype bus2 is std_logic_vector(1 downto 0);
--    subtype bus1 is std_logic;

    -- Address of a register type
    subtype adr_reg_type is std_logic_vector(5 downto 0);

    -- Coding of the level of data availability for UR
    subtype level_type is std_logic_vector(1 downto 0);
    constant LVL_DI  : level_type := "11";  -- Data available from the op2 of DI stage
    constant LVL_EX  : level_type := "10";  -- Data available from the data_ual register of
EX stage
    constant LVL_MEM : level_type := "01";  -- Data available from the data_ecr register of
MEM stage
    constant LVL_REG : level_type := "00";  -- Data available only in the register bank

    -- Different values of cause exceptions
    constant IT_NOEXC : bus32 := X"00000000";
    constant IT_ITMAT : bus32 := X"00000001";
    constant IT_OVERF : bus32 := X"00000002";
    constant IT_ERINS : bus32 := X"00000004";
    constant IT_BREAK : bus32 := X"00000008";
    constant IT_SCALL : bus32 := X"00000010";


    -- Operation type of the coprocessor system (only the low 16 bits are valid)
    constant SYS_NOP    : bus32 := X"0000_0000";
    constant SYS_MASK   : bus32 := X"0000_0001";
    constant SYS_UNMASK : bus32 := X"0000_0002";
    constant SYS_ITRET  : bus32 := X"0000_0004";

    -- Type for the alu control
    subtype alu_ctrl_type is std_logic_vector(27 downto 0);


-- lower bits = rightmost 16 bits?
-- upper bits = leftmost 16 bits?

    -- Arithmetical operations
    constant OP_ADD   : alu_ctrl_type := "1000000000000000000000000000"; -- op1 + op2 signed
    constant OP_ADDU  : alu_ctrl_type := "0100000000000000000000000000"; -- op1 + op2
```

```
unsigned
    constant OP_SUB   : alu_ctrl_type := "00100000000000000000000000000000"; -- op1 - op2 signed
    constant OP_SUBU  : alu_ctrl_type := "00010000000000000000000000000000"; -- op1 - op2
unsigned
    -- Logical operations
    constant OP_AND   : alu_ctrl_type := "00001000000000000000000000000000"; -- logical AND
    constant OP_OR    : alu_ctrl_type := "00000100000000000000000000000000"; -- logical OR
    constant OP_XOR   : alu_ctrl_type := "00000010000000000000000000000000"; -- logical XOR
    constant OP_NOR   : alu_ctrl_type := "00000001000000000000000000000000"; -- logical NOR
    -- Tests : result to one if ok
    constant OP_SLT   : alu_ctrl_type := "00000000100000000000000000000000"; -- op1 < op2
(signed)
    constant OP_SLTU  : alu_ctrl_type := "00000000010000000000000000000000"; -- op1 < op2
(unsigned)
    constant OP_EQU   : alu_ctrl_type := "00000000001000000000000000000000"; -- op1 = op2
    constant OP_NEQU  : alu_ctrl_type := "00000000000100000000000000000000"; -- op1 /= op2
    constant OP_SNEG  : alu_ctrl_type := "00000000000010000000000000000000"; -- op1 < 0
    constant OP_SPOS  : alu_ctrl_type := "00000000000001000000000000000000"; -- op1 > 0
    constant OP_LNEG  : alu_ctrl_type := "00000000000000100000000000000000"; -- op1 <= 0
    constant OP_LPOS  : alu_ctrl_type := "00000000000000010000000000000000"; -- op1 >= 0
    -- Multiplications
    constant OP_MULT  : alu_ctrl_type := "00000000000000000100000000000000"; -- op1 * op2 signed
(load lower bits)
    constant OP_MULTU : alu_ctrl_type := "00000000000000000010000000000000"; -- op1 * op2
unsigned (load lower bits)
    -- Shifts
    constant OP_SLL   : alu_ctrl_type := "00000000000000000001000000000000"; -- logical shift
left
    constant OP_SRL   : alu_ctrl_type := "00000000000000000000100000000000"; -- logical shift
right
    constant OP_SRA   : alu_ctrl_type := "00000000000000000000010000000000"; -- arithmetic shift
right
    constant OP_LUI   : alu_ctrl_type := "00000000000000000000001000000000"; -- load immediate
value
    -- Access to internal registers
    constant OP_MFHI  : alu_ctrl_type := "00000000000000000000000100000000"; -- read upper bits
    constant OP_MFLO  : alu_ctrl_type := "00000000000000000000000010000000"; -- read lower bits
    constant OP_MTHI  : alu_ctrl_type := "00000000000000000000000001000000"; -- write upper bits
    constant OP_MTLO  : alu_ctrl_type := "00000000000000000000000000100000"; -- write lower bits
    -- Operations which do nothing but are useful
    constant OP_OUI   : alu_ctrl_type := "00000000000000000000000000010000"; -- puts a 1 bit on
exit (lower bits)
    constant OP_OP2   : alu_ctrl_type := "00000000000000000000000000001000"; -- copy operand 2
on exit



    -- Starting boot address (after reset)
    constant ADR_INIT : bus32 := X"00000000";
    constant INS_NOP : bus32 := X"00000000";


    -- Internal component of the pipeline stage

    component alu
    port (
        clock : in std_logic;
        reset : in std_logic;
        op1 : in bus32;
        op2 : in bus32;
        ctrl : in alu_ctrl_type;

        res : out bus32;
```

```
        overflow : out std_logic
    );
    end component;

    -- Pipeline stage components

    component pps_pf
    port (
        clock : in std_logic;
        reset : in std_logic;
        stop_all : in std_logic;

            bra_cmd  : in std_logic;
        bra_adr  : in bus32;
        exch_cmd : in std_logic;
        exch_adr : in bus32;

        stop_pf : in std_logic;

        PF_pc : out bus32
    );
    end component;


    component pps_ei
    port (
        clock : in std_logic;
        reset : in std_logic;
        clear  : in std_logic;
        stop_all : in std_logic;

        stop_ei : in std_logic;
        genop : in std_logic;

        CTE_instr : in bus32;
        ETC_adr : out bus32;

        PF_pc : in bus32;

        EI_instr : out bus32;
        EI_adr : out bus32;
        EI_it_ok : out std_logic
    );
    end component;


    component pps_di
    port (
        clock : in std_logic;
        reset : in std_logic;
        stop_all : in std_logic;
        clear : in std_logic;

        bra_detect : out std_logic;

        adr_reg1 : out adr_reg_type;
        adr_reg2 : out adr_reg_type;
        use1 : out std_logic;
        use2 : out std_logic;

        stop_di : in std_logic;
        data1 : in bus32;
        data2 : in bus32;
```

```
        EI_adr : in bus32;
        EI_instr : in bus32;
        EI_it_ok : in std_logic;

        DI_bra : out std_logic;
        DI_link : out std_logic;
        DI_op1 : out bus32;
        DI_op2 : out bus32;
        DI_code_ual : out alu_ctrl_type;
        DI_offset : out bus32;
        DI_adr_reg_dest : out adr_reg_type;
        DI_ecr_reg : out std_logic;
        DI_mode : out std_logic;
        DI_op_mem : out std_logic;
        DI_r_w : out std_logic;
        DI_adr : out bus32;
        DI_exc_cause : out bus32;
        DI_level : out level_type;
        DI_it_ok : out std_logic
    );
    end component;


    component pps_ex
    port (
        clock : in std_logic;
        reset : in std_logic;
        stop_all : in std_logic;
        clear : in std_logic;

        DI_bra : in std_logic;
        DI_link : in std_logic;
        DI_op1 : in bus32;
        DI_op2 : in bus32;
        DI_code_ual : in alu_ctrl_type;
        DI_offset : in bus32;
        DI_adr_reg_dest : in adr_reg_type;
        DI_ecr_reg : in std_logic;
        DI_mode : in std_logic;
        DI_op_mem : in std_logic;
        DI_r_w : in std_logic;
        DI_adr : in bus32;
        DI_exc_cause : in bus32;
        DI_level : in level_type;
        DI_it_ok : in std_logic;

        EX_adr : out bus32;
        EX_bra_confirm : out std_logic;
        EX_data_ual : out bus32;
        EX_adresse : out bus32;
        EX_adr_reg_dest : out adr_reg_type;
        EX_ecr_reg : out std_logic;
        EX_op_mem : out std_logic;
        EX_r_w : out std_logic;
        EX_exc_cause : out bus32;
        EX_level : out level_type;
        EX_it_ok : out std_logic
    );
    end component;


    component pps_mem
```

```
    port (
        clock : in std_logic;
        reset : in std_logic;
        stop_all : in std_logic;
        clear : in std_logic;

        MTC_data : out bus32;
        MTC_adr : out bus32;
        MTC_r_w : out std_logic;
        MTC_req : out std_logic;
        CTM_data : in bus32;

        EX_adr : in bus32;
        EX_data_ual : in bus32;
        EX_adresse : in bus32;
        EX_adr_reg_dest : in adr_reg_type;
        EX_ecr_reg : in std_logic;
        EX_op_mem : in std_logic;
        EX_r_w : in std_logic;
        EX_exc_cause : in bus32;
        EX_level : in level_type;
        EX_it_ok : in std_logic;

        MEM_adr : out bus32;
        MEM_adr_reg_dest : out adr_reg_type;
        MEM_ecr_reg : out std_logic;
        MEM_data_ecr : out bus32;
        MEM_exc_cause : out bus32;
        MEM_level : out level_type;
        MEM_it_ok : out std_logic
    );
    end component;


    component renvoi  -- reference
    port (
        adr1 : in adr_reg_type;
        adr2 : in adr_reg_type;
        use1 : in std_logic;
        use2 : in std_logic;

        data1 : out bus32;
        data2 : out bus32;
        alea : out std_logic;

        DI_level : in level_type;
        DI_adr : in adr_reg_type;
        DI_ecr : in std_logic;
        DI_data : in bus32;

        EX_level : in level_type;
        EX_adr : in adr_reg_type;
        EX_ecr : in std_logic;
        EX_data : in bus32;

        MEM_level : in level_type;
        MEM_adr : in adr_reg_type;
        MEM_ecr : in std_logic;
        MEM_data : in bus32;

        interrupt : in std_logic;

        write_data : out bus32;
```

```
        write_adr : out bus5;
        write_GPR : out std_logic;
        write_SCP : out std_logic;

        read_adr1 : out bus5;
        read_adr2 : out bus5;
        read_data1_GPR : in bus32;
        read_data1_SCP : in bus32;
        read_data2_GPR : in bus32;
        read_data2_SCP : in bus32
    );
    end component;


    component banc  -- register file
    port (
        clock : in std_logic;
        reset : std_logic;

        reg_src1 : in bus5;
        reg_src2 : in bus5;

        reg_dest : in bus5;
        donnee   : in bus32; -- data in

        cmd_ecr  : in std_logic;  -- write enable

        data_src1 : out bus32;    -- data out 1
        data_src2 : out bus32;    -- data out 2

       stop_all   : out std_logic
    );
    end component;


    component bus_ctrl
    port
    (
        clock : std_logic;
        reset : std_logic;

        -- Interruption in the pipeline
        interrupt      : in std_logic;

        -- Interface for the Instruction Extraction Stage
        adr_from_ei    : in bus32;      -- The address of the data to read
        instr_to_ei    : out bus32;     -- Instruction from the memory

        -- Interface with the MEMory Stage
        req_from_mem   : in std_logic;  -- Request to access the ram
        r_w_from_mem   : in std_logic;  -- Read/Write request
        adr_from_mem   : in bus32;      -- Address in ram
        data_from_mem  : in bus32;      -- Data to write in ram
        data_to_mem    : out bus32;     -- Data from the ram to the MEMory stage

        load_to_video  : out std_logic;
        vsync_to_video : out std_logic;
        hsync_to_video : out std_logic;
        blank_to_video : out std_logic;
        byte_to_video  : out bus8;

        -- Pipeline progress control signal
        stop_all       : in std_logic
```

```
);
end component;


component syscop
port
(
    clock         : in std_logic;
    reset         : in std_logic;

    MEM_adr       : in bus32;
    MEM_exc_cause : in bus32;
    MEM_it_ok     : in std_logic;

    it_mat        : in std_logic;

    interrupt     : out std_logic;
    vecteur_it    : out bus32;

    write_data    : in bus32;
    write_adr     : in bus5;
    write_SCP     : in std_logic;

    read_adr1     : in bus5;
    read_adr2     : in bus5;
    read_data1    : out bus32;
    read_data2    : out bus32
);
end component;


component minimips
port (
    clock    : in std_logic;
    reset    : in std_logic;

    load_to_video  : out std_logic;
    vsync_to_video : out std_logic;
    hsync_to_video : out std_logic;
    blank_to_video : out std_logic;
    byte_to_video  : out bus8;

    it_mat   : in std_logic
);
end component;


component video_controller
    port (
    Pixel_Clock     : in std_logic;
   Reset           : in std_logic;

    --minimips output : input video controller
   RT_Font_Byte    : in std_logic_vector(7 downto 0);
   RT_Shift_Load   : in std_logic;
   RT_Blank        : in std_logic;
   RT_Hsync        : in std_logic;
   RT_VSync        : in std_logic;

    --video connections
   VIDOUT_CLK     : out std_logic;
   VIDOUT_RED     : out std_logic_vector (9 downto 0);
   VIDOUT_GREEN   : out std_logic_vector (9 downto 0);
```

```
      VIDOUT_BLUE    : out std_logic_vector (9 downto 0);
      VIDOUT_BLANK_N : out std_logic;
      VIDOUT_HSYNC_N : out std_logic;
      VIDOUT_VSYNC_N : out std_logic
   );
   end component;

   component RTProc
   port (

      -- signals to video DAC
      Pixel_Clock      : in std_logic;
      VIDOUT_CLK     : out std_logic;
      VIDOUT_RCR     : out std_logic_vector (9 downto 0);
      VIDOUT_GY      : out std_logic_vector (9 downto 0);
      VIDOUT_BCB     : out std_logic_vector (9 downto 0);
      VIDOUT_BLANK_N : out std_logic;
      VIDOUT_HSYNC_N : out std_logic;
      VIDOUT_VSYNC_N : out std_logic;

      BAR_LED        : out std_logic_vector (9 downto 0)
   );
   end component;

end pack_mips;
```

----------rtproc.vhd : Top level ----------

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.pack_mips.all;

entity RTPRoc is
port (
    sys_clk       : in std_logic;
    sys_rst       : in std_logic;
    -- signals to video DAC
    pixel_clock   : in std_logic;
    -- Video outputs to the Video DAC
    VIDOUT_CLK     : out std_logic;
    VIDOUT_RED     : out std_logic_vector(9 downto 0);
    VIDOUT_GREEN   : out std_logic_vector(9 downto 0);
    VIDOUT_BLUE    : out std_logic_vector(9 downto 0);
    VIDOUT_BLANK_N : out std_logic;
    VIDOUT_HSYNC_N : out std_logic;
    VIDOUT_VSYNC_N : out std_logic;
    BAR_LED        : out std_logic_vector(9 downto 0)
    );

end RTPRoc;

architecture rtl of RTPRoc is
    -- Video signals to the video_controller
    signal vsync_to_video : std_logic;
    signal hsync_to_video : std_logic;
    signal blank_to_video : std_logic;
    signal load_to_video  : std_logic;
    signal byte_to_video  : std_logic_vector(7 downto 0);

    signal VS, HS, B : std_logic;

begin

   --BAR_LED(9 downto 2) <= byte_to_video;
   BAR_LED(6 downto 0) <= "1111111";

   BAR_LED(9) <= B;
   BAR_LED(8) <= HS;
   BAR_LED(7) <= VS;

VIDOUT_BLANK_N <= B;
VIDOUT_HSYNC_N <= HS;
VIDOUT_VSYNC_N <= VS;

   MM : minimips port map (
        clock => sys_clk,
        reset => sys_rst,
        vsync_to_video => vsync_to_video,
        hsync_to_video => hsync_to_video,
        blank_to_video => blank_to_video,
        load_to_video => load_to_video,
        byte_to_video => byte_to_video,

        it_mat => '0'
    );
```

```
    V_CONTROL : video_controller port map (
       Pixel_Clock => pixel_clock,
       Reset => sys_rst,

       RT_Font_Byte => byte_to_video,
       RT_Shift_Load => load_to_video,
       RT_Blank => blank_to_video,
       RT_Hsync => hsync_to_video,
       RT_VSync => vsync_to_video,

       VIDOUT_CLK  => VIDOUT_CLK,
       VIDOUT_RED => VIDOUT_RED,
       VIDOUT_GREEN=> VIDOUT_GREEN,
       VIDOUT_BLUE => VIDOUT_BLUE,

       VIDOUT_BLANK_N =>B,
       VIDOUT_HSYNC_N =>HS,
       VIDOUT_VSYNC_N =>VS


       --VIDOUT_BLANK_N =>VIDOUT_BLANK_N,
       --VIDOUT_HSYNC_N =>VIDOUT_HSYNC_N,
       --VIDOUT_VSYNC_N =>VIDOUT_VSYNC_N
);


end rtl;
```

----------minimips.vhd : Top level Microprocessor----------

```
library ieee;
use ieee.std_logic_1164.all;

library work;
use work.pack_mips.all;

entity minimips is
port (
    clock    : in std_logic;
    reset    : in std_logic;

    load_to_video  : out std_logic;
    vsync_to_video : out std_logic;
    hsync_to_video : out std_logic;
    blank_to_video : out std_logic;
    byte_to_video  : out bus8;

    -- Hardware interruption
    it_mat : in std_logic
);
end minimips;

architecture rtl of minimips is

    -- General signals
    signal stop_all : std_logic;           -- Lock the pipeline evolution

    signal it_mat_clk : std_logic;         -- Synchronised hardware interruption
    signal stop_pf : std_logic;            -- Lock the pc
    signal genop : std_logic;              -- envoi de nops

    -- interface PF - EI
    signal PF_pc : bus32;                  -- PC value
```

```
    -- interface Controler - EI
    signal CTE_instr : bus32;               -- Instruction from the memory
    signal ETC_adr : bus32;                 -- Address to read in memory

    -- interface EI - DI
    signal EI_instr : bus32;                -- Read interface
    signal EI_adr : bus32;                  -- Address from the read instruction
    signal EI_it_ok : std_logic;            -- Allow hardware interruptions

    -- DI output
    signal bra_detect : std_logic;          -- Branch detection in the current instruction

    -- Asynchronous connexion with the bypass unit
    signal adr_reg1 : adr_reg_type;         -- Operand 1 address
    signal adr_reg2 : adr_reg_type;         -- Operand 2 address
    signal use1 : std_logic;                -- Operand 1 utilisation
    signal use2 : std_logic;                -- Operand 2 utilisation
    signal data1 : bus32;                   -- First register value
    signal data2 : bus32;                   -- Second register value
    signal alea : std_logic;                -- Unresolved hazards detected


    -- interface DI - EX
    signal DI_bra : std_logic;              -- Branch decoded
    signal DI_link : std_logic;             -- A link for that instruction
    signal DI_op1 : bus32;                  -- operand 1 for alu
    signal DI_op2 : bus32;                  -- operand 2 for alu
    signal DI_code_ual : alu_ctrl_type;     -- Alu operation
    signal DI_offset : bus32;               -- Offset for the address calculation
    signal DI_adr_reg_dest : adr_reg_type;  -- Address of the destination register of the
result
    signal DI_ecr_reg : std_logic;          -- Effective writing of the result
    signal DI_mode : std_logic;             -- Address mode (relative to pc or indexed to a
register)
    signal DI_op_mem : std_logic;           -- Memory operation request
    signal DI_r_w : std_logic;              -- Type of memory operation (reading or writing)
    signal DI_adr : bus32;                  -- Address of the decoded instruction
    signal DI_exc_cause : bus32;            -- Potential exception detected
    signal DI_level : level_type;           -- Availability of the result for the data
bypass
    signal DI_it_ok : std_logic;            -- Allow hardware interruptions

    -- interface EX - MEM
    signal EX_adr : bus32;                  -- Instruction address
    signal EX_bra_confirm : std_logic;      -- Branch execution confirmation
    signal EX_data_ual : bus32;             -- Ual result
    signal EX_adresse : bus32;              -- Address calculation result
    signal EX_adr_reg_dest : adr_reg_type;  -- Destination register for the result
    signal EX_ecr_reg : std_logic;          -- Effective writing of the result
    signal EX_op_mem : std_logic;           -- Memory operation needed
    signal EX_r_w : std_logic;              -- Type of memory operation (read or write)
    signal EX_exc_cause : bus32;            -- Potential cause exception
    signal EX_level : level_type;           -- Availability stage of result for bypassing
    signal EX_it_ok : std_logic;            -- Allow hardware interruptions

    -- interface Controler - MEM
    signal MTC_data : bus32;                -- Data to write in memory
    signal MTC_adr : bus32;                 -- Address for memory
    signal MTC_r_w : std_logic;             -- Read/Write in memory
    signal MTC_req : std_logic;             -- Request access to memory
    signal CTM_data : bus32;                -- Data from memory
```

```
    -- interface MEM - REG
    signal MEM_adr : bus32;                 -- Instruction address
    signal MEM_adr_reg_dest : adr_reg_type; -- Destination register address
    signal MEM_ecr_reg : std_logic;         -- Writing of the destination register
    signal MEM_data_ecr : bus32;            -- Data to write (from alu or memory)
    signal MEM_exc_cause : bus32;           -- Potential exception cause
    signal MEM_level : level_type;          -- Availability stage for the result for
bypassing
    signal MEM_it_ok : std_logic;           -- Allow hardware interruptions

    -- connexion to the register banks

      -- Writing commands in the register banks
    signal write_data : bus32;              -- Data to write
    signal write_adr : bus5;                -- Address of the register to write
    signal write_GPR : std_logic;           -- Selection in the internal registers
    signal write_SCP : std_logic;           -- Selection in the coprocessor system registers

      -- Reading commands for Reading in the registers
    signal read_adr1 : bus5;                -- Address of the first register to read
    signal read_adr2 : bus5;                -- Address of the second register to read
    signal read_data1_GPR : bus32;          -- Value of operand 1 from the internal
registers
    signal read_data1_SCP : bus32;          -- Value of operand 2 from the internal
registers
    signal read_data2_GPR : bus32;          -- Value of operand 1 from the coprocessor
system registers
    signal read_data2_SCP : bus32;          -- Value of operand 2 from the coprocessor
system registers

    -- Interruption controls
    signal interrupt     : std_logic;       -- Interruption to take into account
    signal vecteur_it    : bus32;           -- Interruption vector

begin

    stop_pf <= DI_bra or bra_detect or alea;
    genop <= bra_detect or EX_bra_confirm or DI_bra;

    -- Take into account the hardware interruption on rising edge
    process (clock)
    begin
        if clock='1' and clock'event then
            it_mat_clk <= it_mat;
        end if;
    end process;

    U1_pf : pps_pf port map (
        clock => clock,
        reset => reset,
        stop_all => stop_all,               -- Unconditionnal locking of the pipeline stage

        -- entrees asynchrones
        bra_adr => EX_adresse,              -- Branch
        bra_cmd => EX_bra_confirm,          -- Address to load when an effective branch
        exch_adr => vecteur_it,             -- Exception branch
        exch_cmd => interrupt,              -- Exception vector
        stop_pf => stop_pf,                 -- Lock the stage


        -- Synchronous output to EI stage
        PF_pc => PF_pc                      -- PC value
    );
```

```
    U2_ei : pps_ei port map (
        clock => clock,
        reset => reset,
        clear => interrupt,         -- Clear the pipeline stage
        stop_all => stop_all,       -- Evolution locking signal

        -- Asynchronous inputs
        stop_ei => alea,            -- Lock the EI_adr and Ei_instr registers
        genop => genop,             -- Send nops

        -- interface Controler - EI
        CTE_instr => CTE_instr,     -- Instruction from the memory
        ETC_adr => ETC_adr,         -- Address to read in memory

        -- Synchronous inputs from PF stage
        PF_pc => PF_pc,             -- Current value of the pc

        -- Synchronous outputs to DI stage
        EI_instr => EI_instr,       -- Read interface
        EI_adr => EI_adr,           -- Address from the read instruction
        EI_it_ok => EI_it_ok        -- Allow hardware interruptions
    );


    U3_di : pps_di port map (
        clock => clock,
        reset => reset,
        stop_all => stop_all,               -- Unconditionnal locking of the outputs
        clear => interrupt,                 -- Clear the pipeline stage (nop in the outputs)

        -- Asynchronous outputs
        bra_detect => bra_detect,           -- Branch detection in the current instruction

        -- Asynchronous connexion with the register management and data bypass unit
        adr_reg1 => adr_reg1,               -- Address of the first register operand
        adr_reg2 => adr_reg2,               -- Address of the second register operand
        use1 => use1,                       -- Effective use of operand 1
        use2 => use2,                       -- Effective use of operand 2

        stop_di => alea,                    -- Unresolved detected : send nop in the
pipeline
        data1 => data1,                     -- Operand register 1
        data2 => data2,                     -- Operand register 2

        -- Datas from EI stage
        EI_adr => EI_adr,                   -- Address of the instruction
        EI_instr => EI_instr,               -- The instruction to decode
        EI_it_ok => EI_it_ok,               -- Allow hardware interruptions

        -- Synchronous output to EX stage
        DI_bra => DI_bra,                   -- Branch decoded
        DI_link => DI_link,                 -- A link for that instruction
        DI_op1 => DI_op1,                   -- operand 1 for alu
        DI_op2 => DI_op2,                   -- operand 2 for alu
        DI_code_ual => DI_code_ual,         -- Alu operation
        DI_offset => DI_offset,             -- Offset for the address calculation
        DI_adr_reg_dest => DI_adr_reg_dest, -- Address of the destination register of the
result
        DI_ecr_reg => DI_ecr_reg,           -- Effective writing of the result
        DI_mode => DI_mode,                 -- Address mode (relative to pc or indexed to a
register)
```

```
        DI_op_mem => DI_op_mem,             -- Memory operation request
        DI_r_w => DI_r_w,                   -- Type of memory operation (reading or writing)
        DI_adr => DI_adr,                   -- Address of the decoded instruction
        DI_exc_cause => DI_exc_cause,       -- Potential exception detected
        DI_level => DI_level,               -- Availability of the result for the data
bypass
        DI_it_ok => DI_it_ok                -- Allow hardware interruptions
    );


    U4_ex : pps_ex port map (
        clock => clock,
        reset => reset,
        stop_all => stop_all,               -- Unconditionnal locking of outputs
        clear => interrupt,                 -- Clear the pipeline stage

        -- Datas from DI stage
        DI_bra => DI_bra,                   -- Branch instruction
        DI_link => DI_link,                 -- Branch with link
        DI_op1 => DI_op1,                   -- Operand 1 for alu
        DI_op2 => DI_op2,                   -- Operand 2 for alu
        DI_code_ual => DI_code_ual,         -- Alu operation
        DI_offset => DI_offset,             -- Offset for address calculation
        DI_adr_reg_dest => DI_adr_reg_dest, -- Destination register address for the result
        DI_ecr_reg => DI_ecr_reg,           -- Effective writing of the result
        DI_mode => DI_mode,                 -- Address mode (relative to pc ou index by a
register)
        DI_op_mem => DI_op_mem,             -- Memory operation
        DI_r_w => DI_r_w,                   -- Type of memory operation (read or write)
        DI_adr => DI_adr,                   -- Instruction address
        DI_exc_cause => DI_exc_cause,       -- Potential cause exception
        DI_level => DI_level,               -- Availability stage of the result for
bypassing
        DI_it_ok => DI_it_ok,               -- Allow hardware interruptions

        -- Synchronous outputs to MEM stage
        EX_adr => EX_adr,                   -- Instruction address
        EX_bra_confirm => EX_bra_confirm,   -- Branch execution confirmation
        EX_data_ual => EX_data_ual,         -- Ual result
        EX_adresse => EX_adresse,           -- Address calculation result
        EX_adr_reg_dest => EX_adr_reg_dest, -- Destination register for the result
        EX_ecr_reg => EX_ecr_reg,           -- Effective writing of the result
        EX_op_mem => EX_op_mem,             -- Memory operation needed
        EX_r_w => EX_r_w,                   -- Type of memory operation (read or write)
        EX_exc_cause => EX_exc_cause,       -- Potential cause exception
        EX_level => EX_level,               -- Availability stage of result for bypassing
        EX_it_ok => EX_it_ok                -- Allow hardware interruptions
    );


    U5_mem : pps_mem port map (
        clock => clock,
        reset => reset,
        stop_all => stop_all,               -- Unconditionnal locking of the outputs
        clear => interrupt,                 -- Clear the pipeline stage

        -- Interface with the control bus
        MTC_data => MTC_data,               -- Data to write in memory
        MTC_adr => MTC_adr,                 -- Address for memory
        MTC_r_w => MTC_r_w,                 -- Read/Write in memory
        MTC_req => MTC_req,                 -- Request access to memory
        CTM_data => CTM_data,               -- Data from memory
```

```
        -- Datas from Execution stage
        EX_adr => EX_adr,                    -- Instruction address
        EX_data_ual => EX_data_ual,          -- Result of alu operation
        EX_adresse => EX_adresse,            -- Result of the calculation of the address
        EX_adr_reg_dest => EX_adr_reg_dest, -- Destination register address for the result
        EX_ecr_reg => EX_ecr_reg,            -- Effective writing of the result
        EX_op_mem => EX_op_mem,              -- Memory operation needed
        EX_r_w => EX_r_w,                    -- Type of memory operation (read or write)
        EX_exc_cause => EX_exc_cause,        -- Potential exception cause
        EX_level => EX_level,                -- Availability stage for the result for
bypassing
        EX_it_ok => EX_it_ok,                -- Allow hardware interruptions

        -- Synchronous outputs for bypass unit
        MEM_adr => MEM_adr,                  -- Instruction address
        MEM_adr_reg_dest=>MEM_adr_reg_dest, -- Destination register address
        MEM_ecr_reg => MEM_ecr_reg,          -- Writing of the destination register
        MEM_data_ecr => MEM_data_ecr,        -- Data to write (from alu or memory)
        MEM_exc_cause => MEM_exc_cause,      -- Potential exception cause
        MEM_level => MEM_level,              -- Availability stage for the result for
bypassing
        MEM_it_ok => MEM_it_ok               -- Allow hardware interruptions
    );


    U6_renvoi : renvoi port map (
        -- Register access signals
        adr1 => adr_reg1,                    -- Operand 1 address
        adr2 => adr_reg2,                    -- Operand 2 address
        use1 => use1,                        -- Operand 1 utilisation
        use2 => use2,                        -- Operand 2 utilisation

        data1 => data1,                      -- First register value
        data2 => data2,                      -- Second register value
        alea => alea,                        -- Unresolved hazards detected

        -- Bypass signals of the intermediary datas
        DI_level => DI_level,                -- Availability level of the data
        DI_adr => DI_adr_reg_dest,           -- Register destination of the result
        DI_ecr => DI_ecr_reg,                -- Writing register request
        DI_data => DI_op2,                   -- Data to used

        EX_level => EX_level,                -- Availability level of the data
        EX_adr => EX_adr_reg_dest,           -- Register destination of the result
        EX_ecr => EX_ecr_reg,                -- Writing register request
        EX_data => EX_data_ual,              -- Data to used

        MEM_level => MEM_level,              -- Availability level of the data
        MEM_adr => MEM_adr_reg_dest,         -- Register destination of the result
        MEM_ecr => MEM_ecr_reg,              -- Writing register request
        MEM_data => MEM_data_ecr,            -- Data to used

        interrupt => interrupt,              -- Exceptions or interruptions

        -- Connexion to the differents bank of register

          -- Writing commands for writing in the registers
        write_data => write_data,            -- Data to write
        write_adr => write_adr,              -- Address of the register to write
        write_GPR => write_GPR,              -- Selection in the internal registers
        write_SCP => write_SCP,              -- Selection in the coprocessor system registers

          -- Reading commands for Reading in the registers
```

```
        read_adr1 => read_adr1,             -- Address of the first register to read
        read_adr2 => read_adr2,             -- Address of the second register to read
        read_data1_GPR => read_data1_GPR,   -- Value of operand 1 from the internal
registers
        read_data1_SCP => read_data1_SCP,   -- Value of operand 2 from the internal
registers
        read_data2_GPR => read_data2_GPR,   -- Value of operand 1 from the coprocessor
system registers
        read_data2_SCP => read_data2_SCP    -- Value of operand 2 from the coprocessor
system registers
    );


    U7_banc : banc port map(
        clock => clock,
        reset => reset,

        -- Register addresses to read
        reg_src1 => read_adr1,
        reg_src2 => read_adr2,

        -- Register address to write and its data
        reg_dest => write_adr,
        donnee   => write_data,

        -- Write signal
        cmd_ecr  => write_GPR,

        -- Bank outputs
        data_src1 => read_data1_GPR,
        data_src2 => read_data2_GPR,
           stop_all   => stop_all
    );


    U8_syscop : syscop port map (
        clock         => clock,
        reset         => reset,

        -- Datas from the pipeline
        MEM_adr       => MEM_adr,           -- Address of the current instruction in the
pipeline end -> responsible of the exception
        MEM_exc_cause => MEM_exc_cause,     -- Potential cause exception of that instruction
        MEM_it_ok     => MEM_it_ok,         -- Allow hardware interruptions

        -- Hardware interruption
        it_mat        => it_mat_clk,        -- Hardware interruption detected

        -- Interruption controls
        interrupt     => interrupt,         -- Interruption to take into account
        vecteur_it    => vecteur_it,        -- Interruption vector

        -- Writing request in register bank
        write_data    => write_data,        -- Data to write
        write_adr     => write_adr,         -- Address of the register to write
        write_SCP     => write_SCP,         -- Writing request

        -- Reading request in register bank
        read_adr1     => read_adr1,         -- Address of the first register
        read_adr2     => read_adr2,         -- Address of the second register
        read_data1    => read_data1_SCP,    -- Value of register 1
        read_data2    => read_data2_SCP     -- Value of register 2
    );
```

```
    U9_bus_ctrl : bus_ctrl port map (
        clock          => clock,
        reset          => reset,

        -- Interruption in the pipeline
        interrupt      => interrupt,

        -- Interface for the Instruction Extraction Stage
        adr_from_ei    => ETC_adr,          -- The address of the data to read
        instr_to_ei    => CTE_instr,        -- Instruction from the memory

        -- Interface with the MEMory Stage
        req_from_mem   => MTC_req,          -- Request to access the ram
        r_w_from_mem   => MTC_r_w,          -- Read/Write request
        adr_from_mem   => MTC_adr,          -- Address in ram
        data_from_mem  => MTC_data,         -- Data to write in ram
        data_to_mem    => CTM_data,         -- Data from the ram to the MEMory stage

        load_to_video  => load_to_video,
        vsync_to_video => vsync_to_video,
        hsync_to_video => hsync_to_video,
        blank_to_video => blank_to_video,
        byte_to_video  => byte_to_video,

        -- Pipeline progress control signal
        stop_all       => stop_all
    );

end rtl;
```

```
----------------------pps_pf.vhd 1st level pipeline--------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.pack_mips.all;

entity pps_pf is
port (
    clock       : in std_logic;
    reset       : in std_logic;
    stop_all    : in std_logic;                 -- Unconditionnal locking of the
pipeline stage

    -- Asynchronous inputs
    bra_cmd     : in std_logic;                 -- Branch
    bra_adr     : in bus32;       -- Address to load when an effective branch
    exch_cmd    : in std_logic;                 -- Exception branch
    exch_adr    : in bus32;       -- Exception vector
    stop_pf     : in std_logic;                 -- Lock the stage

    -- Synchronous output to EI stage
    PF_pc       : out bus32       -- PC value
);
```

```
end pps_pf;

architecture rtl of pps_pf is

    signal suivant : bus32;       -- Preparation of the future pc
    signal pc_interne : bus32;    -- Value of the pc output, needed for an internal
reading
    signal lock : std_logic;              -- Specify the authorization of the pc
evolution

begin

    -- Connexion the pc to the internal pc
    PF_pc <= pc_interne;

    -- Elaboration of an potential future pc
    suivant <= exch_adr when exch_cmd='1'   else
               bra_adr  when bra_cmd='1'    else
               bus32(unsigned(pc_interne) + 4);

    lock <= '1' when stop_all='1' else -- Lock this stage when all the pipeline is
locked
            '0' when exch_cmd='1' else -- Exception
            '0' when bra_cmd='1'  else -- Branch
            '1' when stop_pf='1'  else -- Wait for the branch hazard
            '0';                       -- Normal evolution

    -- Synchronous evolution of the pc
    process(clock)
    begin
        if clock='1' and clock'event then
            if reset='1' then
                -- PC reinitialisation with the boot address
                pc_interne <= ADR_INIT;
            elsif lock='0' then
                -- PC not locked
                pc_interne <= suivant;
            end if;
        end if;
    end process;

end rtl;
```

----------pps_ei.vhd : 2nd Level pipeline----------

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.pack_mips.all;

entity pps_ei is
port (
  clock : in std_logic;
  reset : in std_logic;
```

```
  clear : in std_logic;    -- Clear the pipeline stage
  stop_all : in std_logic; -- Evolution locking signal

  -- Asynchronous inputs
  stop_ei : in std_logic;  -- Lock the EI_adr and Ei_instr registers
  genop : in std_logic;    -- Send nops

  -- Bus controler interface
  CTE_instr : in bus32;    -- Instruction from the memory
  ETC_adr : out bus32;     -- Address to read in memory

  -- Synchronous inputs from PF stage
  PF_pc : in bus32;        -- Current value of the pc

  -- Synchronous outputs to DI stage
  EI_instr : out bus32;    -- Read interface
  EI_adr : out bus32;      -- Address from the read instruction
  EI_it_ok : out std_logic -- Allow hardware interruptions
);
end pps_ei;

architecture rtl of pps_ei is
begin

  ETC_adr <= PF_pc; -- Connexion of the PC to the memory address bus

  -- Set the results
  process (clock)
  begin
    if (clock='1' and clock'event) then
      if reset='1' then
        EI_instr <= INS_NOP;
        EI_adr <= (others => '0');
        EI_it_ok <= '0';
      elsif stop_all='0' then
        if clear='1' then
          -- Clear the stage
          EI_instr <= INS_NOP;
          EI_it_ok <= '0';
        elsif genop='1' and stop_ei='0' then
          -- Send a nop
          EI_instr <= INS_NOP;
          EI_it_ok <= '1';
        elsif stop_ei='0' then
          -- Normal evolution
          EI_adr <= PF_pc;
          EI_instr <= CTE_instr;
          EI_it_ok <= '1';
        end if;
      end if;
    end if;
  end process;
end rtl;
```

----------pps_di.vhd: 3rd Level pipeline----------

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.pack_mips.all;

entity pps_di is
```

```
port (
    clock : in std_logic;
    reset : in std_logic;
    stop_all : in std_logic;            -- Unconditionnal locking of the outputs
    clear : in std_logic;               -- Clear the pipeline stage (nop in the outputs)

    -- Asynchronous outputs
    bra_detect : out std_logic;         -- Branch detection in the current instruction

    -- Asynchronous connexion with the register management and data bypass unit
    adr_reg1 : out adr_reg_type;        -- Address of the first register operand
    adr_reg2 : out adr_reg_type;        -- Address of the second register operand
    use1 : out std_logic;               -- Effective use of operand 1
    use2 : out std_logic;               -- Effective use of operand 2

    stop_di : in std_logic;             -- Unresolved detected : send nop in the pipeline
    data1 : in bus32;                   -- Operand register 1
    data2 : in bus32;                   -- Operand register 2

    -- Datas from EI stage
    EI_adr : in bus32;                  -- Address of the instruction
    EI_instr : in bus32;                -- The instruction to decode
    EI_it_ok : in std_logic;            -- Allow hardware interruptions

    -- Synchronous output to EX stage
    DI_bra : out std_logic;             -- Branch decoded
    DI_link : out std_logic;            -- A link for that instruction
    DI_op1 : out bus32;                 -- operand 1 for alu
    DI_op2 : out bus32;                 -- operand 2 for alu
    DI_code_ual : out alu_ctrl_type;    -- Alu operation
    DI_offset : out bus32;              -- Offset for the address calculation
    DI_adr_reg_dest : out adr_reg_type; -- Address of the destination register of the result
    DI_ecr_reg : out std_logic;         -- Effective writing of the result
    DI_mode : out std_logic;            -- Address mode (relative to pc or indexed to a
register)
    DI_op_mem : out std_logic;          -- Memory operation request
    DI_r_w : out std_logic;             -- Type of memory operation (reading or writing)
    DI_adr : out bus32;                 -- Address of the decoded instruction
    DI_exc_cause : out bus32;           -- Potential exception detected
    DI_level : out level_type;          -- Availability of the result for the data bypass
    DI_it_ok : out std_logic            -- Allow hardware interruptions
);
end entity;


architecture rtl of pps_di is

    -- Enumeration type used for the micro-code of the instruction
    type op_mode_type is (OP_NORMAL, OP_SPECIAL, OP_REGIMM, OP_COP0);   -- selection du mode
de l'instruction
    type off_sel_type is (OFS_PCRL, OFS_NULL, OFS_SESH, OFS_SEXT);      -- selection de la
valeur de l'offset
    type rdest_type is ( D_RT, D_RD, D_31, D_00);                       -- selection du
registre destination

    -- Record type containg the micro-code of an instruction
    type micro_instr_type is
    record
        op_mode : op_mode_type;    -- Instruction codop mode
        op_code : bus6;            -- Instruction codop
        bra : std_logic;           -- Branch instruction
        link : std_logic;          -- Branch with link : the return address is saved in a
register
```

```
        code_ual : alu_ctrl_type;  -- Operation code for the alu
        op_mem : std_logic;        -- Memory operation needed
        r_w : std_logic;           -- Read/Write selection in memory
        mode : std_logic;          -- Address calculation from the current pc ('1') or the
alu operand 1 ('0')
        off_sel : off_sel_type;    -- Offset source : PC(31..28) & Adresse & 00 || 0 ||
sgn_ext(Imm) & 00 || sgn_ext(Imm)
        exc_cause : bus32;         -- Unconditionnal exception cause to generate
        cop_org1 : std_logic;      -- Source register 1 : general register if 0, coprocessor
register if 1
        cop_org2 : std_logic;      -- Source register 2 : general register if 0, coprocessor
register if 1
        cs_imm1 : std_logic;       -- Use of immediat operand 1 instead of register bank
        cs_imm2 : std_logic;       -- Use of immediat operand 2 instead of register bank
        imm1_sel : std_logic;      -- Origine of immediat operand 1
        imm2_sel : std_logic;      -- Origine of immediat operand 2
        level : level_type;        -- Data availability stage for the bypass
        ecr_reg : std_logic;       -- Writing the result in a register
        bank_des : std_logic;      -- Register bank selection : GPR if 0, coprocessor system
if 1
        des_sel : rdest_type ;     -- Destination register address : Rt, Rd, $31, $0
    end record;

    type micro_code_type is array (natural range <>) of micro_instr_type;

    constant micro_code : micro_code_type :=
( -- Instruction decoding in micro-instructions table
(OP_SPECIAL, "100000", '0', '0', OP_ADD  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- ADD
(OP_NORMAL , "001000", '0', '0', OP_ADD  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'1', '0', '1', LVL_EX , '1', '0', D_RT), -- ADDI
(OP_NORMAL , "001001", '0', '0', OP_ADDU , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_EX , '1', '0', D_RT), -- ADDIU
(OP_SPECIAL, "100001", '0', '0', OP_ADDU , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- ADDU
(OP_SPECIAL, "100100", '0', '0', OP_AND  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- AND
(OP_NORMAL , "001100", '0', '0', OP_AND  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_EX , '1', '0', D_RT), -- ANDI
(OP_NORMAL , "000100", '1', '0', OP_EQU  , '0', '0', '1', OFS_SESH, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_DI , '0', '0', D_RT), -- BEQ
(OP_REGIMM , "000001", '1', '0', OP_LPOS , '0', '0', '1', OFS_SESH, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_DI , '0', '0', D_RT), -- BGEZ
(OP_REGIMM , "010001", '1', '1', OP_LPOS , '0', '0', '1', OFS_SESH, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_EX , '1', '0', D_31), -- BGEZAL
(OP_NORMAL , "000111", '1', '0', OP_SPOS , '0', '0', '1', OFS_SESH, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_DI , '0', '0', D_RT), -- BGTZ
(OP_NORMAL , "000110", '1', '0', OP_LNEG , '0', '0', '1', OFS_SESH, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_DI , '0', '0', D_RT), -- BLEZ
(OP_REGIMM , "000000", '1', '0', OP_SNEG , '0', '0', '1', OFS_SESH, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_DI , '0', '0', D_RT), -- BLTZ
(OP_REGIMM , "010000", '1', '1', OP_SNEG , '0', '0', '1', OFS_SESH, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_EX , '1', '0', D_31), -- BLTZAL
(OP_NORMAL , "000101", '1', '0', OP_NEQU , '0', '0', '1', OFS_SESH, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_DI , '0', '0', D_RT), -- BNE
(OP_SPECIAL, "001101", '0', '0', OP_OUI  , '0', '0', '0', OFS_PCRL, IT_BREAK, '0', '0', '1',
'1', '0', '0', LVL_DI , '0', '0', D_RT), -- BREAK
(OP_COP0   , "000001", '0', '0', OP_OP2  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '1',
'1', '0', '0', LVL_DI , '1', '1', D_00), -- COP0
(OP_NORMAL , "000010", '1', '0', OP_OUI  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '1',
'1', '0', '0', LVL_DI , '0', '0', D_RT), -- J
(OP_NORMAL , "000011", '1', '1', OP_OUI  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '1',
'1', '0', '0', LVL_EX , '1', '0', D_31), -- JAL
```

```
(OP_SPECIAL, "001001", '1', '1', OP_OUI  , '0', '0', '0', OFS_NULL, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_EX , '1', '0', D_RD), -- JALR
(OP_SPECIAL, "001000", '1', '0', OP_OUI  , '0', '0', '0', OFS_NULL, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_DI , '0', '0', D_RT), -- JR
(OP_NORMAL , "001111", '0', '0', OP_LUI  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '1',
'1', '0', '0', LVL_EX , '1', '0', D_RT), -- LUI
(OP_NORMAL , "100011", '0', '0', OP_OUI  , '1', '0', '0', OFS_SEXT, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_MEM, '1', '0', D_RT), -- LW
(OP_NORMAL , "110000", '0', '0', OP_OUI  , '1', '0', '0', OFS_SEXT, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_MEM, '1', '1', D_RT), -- LWC0
(OP_COP0   , "000000", '0', '0', OP_OP2  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '1', '1',
'0', '0', '0', LVL_DI , '1', '0', D_RD), -- MFC0
(OP_SPECIAL, "010000", '0', '0', OP_MFHI , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '1',
'1', '0', '0', LVL_EX , '1', '0', D_RD), -- MFHI
(OP_SPECIAL, "010010", '0', '0', OP_MFLO , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '1',
'1', '0', '0', LVL_EX , '1', '0', D_RD), -- MFLO
(OP_COP0   , "000100", '0', '0', OP_OP2  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '1',
'0', '0', '0', LVL_DI , '1', '1', D_RD), -- MTC0
(OP_SPECIAL, "010001", '0', '0', OP_MTHI , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_DI , '0', '0', D_RT), -- MTHI
(OP_SPECIAL, "010011", '0', '0', OP_MTLO , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_DI , '0', '0', D_RT), -- MTLO
(OP_SPECIAL, "011000", '0', '0', OP_MULT , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '0', '0', D_RT), -- MULT
(OP_SPECIAL, "011001", '0', '0', OP_MULTU, '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '0', '0', D_RT), -- MULT
(OP_SPECIAL, "100111", '0', '0', OP_NOR  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- NOR
(OP_SPECIAL, "100101", '0', '0', OP_OR   , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- OR
(OP_NORMAL , "001101", '0', '0', OP_OR   , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_EX , '1', '0', D_RT), -- ORI
(OP_SPECIAL, "000000", '0', '0', OP_SLL  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '1',
'0', '1', '0', LVL_EX , '1', '0', D_RD), -- SLL
(OP_SPECIAL, "000100", '0', '0', OP_SLL  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- SLLV
(OP_SPECIAL, "101010", '0', '0', OP_SLT  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- SLT
(OP_NORMAL , "001010", '0', '0', OP_SLT  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'1', '0', '1', LVL_EX , '1', '0', D_RT), -- SLTI
(OP_NORMAL , "001011", '0', '0', OP_SLTU , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'1', '0', '1', LVL_EX , '1', '0', D_RT), -- SLTIU
(OP_SPECIAL, "101011", '0', '0', OP_SLTU , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- SLTU
(OP_SPECIAL, "000011", '0', '0', OP_SRA  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '1',
'0', '1', '0', LVL_EX , '1', '0', D_RD), -- SRA
(OP_SPECIAL, "000111", '0', '0', OP_SRA  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- SRAV
(OP_SPECIAL, "000010", '0', '0', OP_SRL  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '1',
'0', '1', '0', LVL_EX , '1', '0', D_RD), -- SRL
(OP_SPECIAL, "000110", '0', '0', OP_SRL  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- SRLV
(OP_SPECIAL, "100010", '0', '0', OP_SUB  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- SUB
(OP_SPECIAL, "100011", '0', '0', OP_SUBU , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- SUBU
(OP_NORMAL , "101011", '0', '0', OP_OP2  , '1', '1', '0', OFS_SEXT, IT_NOEXC, '0', '0', '0',
'0', '0', '0', LVL_DI , '0', '0', D_RT), -- SW
(OP_NORMAL , "111000", '0', '0', OP_OP2  , '1', '1', '0', OFS_SEXT, IT_NOEXC, '0', '1', '0',
'0', '0', '0', LVL_DI , '0', '0', D_RT), -- SWC0
(OP_SPECIAL, "001100", '0', '0', OP_OUI  , '0', '0', '0', OFS_PCRL, IT_SCALL, '0', '0', '1',
'1', '0', '0', LVL_DI , '0', '0', D_RT), -- SYSC
(OP_SPECIAL, "100110", '0', '0', OP_XOR  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
```

```
'0', '0', '0', LVL_EX , '1', '0', D_RD), -- XOR
(OP_NORMAL , "001110", '0', '0', OP_XOR  , '0', '0', '0', OFS_PCRL, IT_NOEXC, '0', '0', '0',
'1', '0', '0', LVL_EX , '1', '0', D_RT)  -- XORI
);

    -- Preparation of the synchronous outputs
    signal PRE_bra : std_logic;             -- Branch operation
    signal PRE_link : std_logic;            -- Branch with link
    signal PRE_op1 : bus32;                 -- operand 1 of the ual
    signal PRE_op2 : bus32;                 -- operand 2 of the ual
    signal PRE_code_ual : alu_ctrl_type;    -- Alu operation
    signal PRE_offset : bus32;              -- Address offset for calculation
    signal PRE_adr_reg_dest : adr_reg_type; -- Destination register adress for result
    signal PRE_ecr_reg : std_logic;         -- Writing of result in the bank register
    signal PRE_mode : std_logic;            -- Address calculation with current pc
    signal PRE_op_mem : std_logic;          -- Memory access operation instruction
    signal PRE_r_w : std_logic;             -- Read/write selection in memory
    signal PRE_exc_cause : bus32;           -- Potential exception cause
    signal PRE_level : level_type;          -- Result availability stage for bypass

begin


    -- Instruction decoding
    process (EI_instr, EI_adr, data1, data2)
        variable op_code : bus6;             -- Effective codop of the instruction
        variable op_mode : op_mode_type;     -- Instruction mode
        variable flag : boolean;             -- Is true if valid instruction
        variable instr : integer;            -- Current micro-instruction adress

        -- Instruction fields
        variable rs : bus5;
        variable rt : bus5;
        variable rd : bus5;
        variable shamt : bus5;
        variable imm : bus16;
        variable address : bus26;
    begin

        -- Selection of the instruction codop and its mode
        case EI_instr(31 downto 26) is
            when "000000" => -- special mode
                op_mode := OP_SPECIAL;
                op_code := EI_instr(5 downto 0);
            when "000001" => -- regimm mode
                op_mode := OP_REGIMM;
                op_code := '0' & EI_instr(20 downto 16);
            when "010000" => -- cop0 mode
                op_mode := OP_COP0;
                op_code := '0' & EI_instr(25 downto 21);
            when others   => -- normal mode
                op_mode := OP_NORMAL;
                op_code := EI_instr(31 downto 26);
        end case;


        -- Search the current instruction in the micro-code table
        flag := false;
        instr := 0;
        for i in micro_code'range loop
            if micro_code(i).op_mode=op_mode and micro_code(i).op_code=op_code then
                flag := true;           -- The instruction exists
                instr := i;             -- Index memorisation
```

```
        end if;
    end loop;

    -- Read the instruction field
    rs      := EI_instr(25 downto 21);
    rt      := EI_instr(20 downto 16);
    rd      := EI_instr(15 downto 11);
    shamt   := EI_instr(10 downto  6);
    imm     := EI_instr(15 downto  0);
    address := EI_instr(25 downto  0);

    if not flag then -- Unknown instruction

        -- Synchronous output preparation
        PRE_bra          <= '0';              -- Branch operation
        PRE_link         <= '0';              -- Branch with link
        PRE_op1          <= (others => '0');  -- operand 1 of the ual
        PRE_op2          <= (others => '0');  -- operand 2 of the ual
        PRE_code_ual     <= OP_OUI;           -- Alu operation
        PRE_offset       <= (others => '0');  -- Address offset for calculation
        PRE_adr_reg_dest <= (others => '0');  -- Destination register adress for result
        PRE_ecr_reg      <= '0';              -- Writing of result in the bank register
        PRE_mode         <= '0';              -- Address calculation with current pc
        PRE_op_mem       <= '0';              -- Memory access operation instruction
        PRE_r_w          <= '0';              -- Read/write selection in memory
        PRE_exc_cause    <= IT_ERINS;         -- Potential exception cause
        PRE_level        <= LVL_DI;           -- Result availability stage for bypass

        -- Set asynchronous outputs
        adr_reg1 <= (others => '0');    -- First operand register
        adr_reg2 <= (others => '0');    -- Second operand register
        bra_detect <= '0';              -- Detection of a branch in current instruction
        use1 <= '0';                    -- Effective use of operand 1
        use2 <= '0';                    -- Effective use of operand 2

    else -- Valid instruction

        -- Offset signal preparation
        case micro_code(instr).off_sel is
            when OFS_PCRL => -- PC(31..28) & Adresse & 00
                PRE_offset <= EI_adr(31 downto 28) & address & "00";
            when OFS_NULL => -- 0
                PRE_offset <= (others => '0');
            when OFS_SESH => -- sgn_ext(Imm) & 00
                if imm(15)='1' then
                    PRE_offset <= "11111111111111" & imm & "00";
                else
                    PRE_offset <= "00000000000000" & imm & "00";
                end if;
            when OFS_SEXT => -- sgn_ext(Imm)
                if imm(15)='1' then
                    PRE_offset <= "1111111111111111" & imm;
                else
                    PRE_offset <= "0000000000000000" & imm;
                end if;
        end case;


        -- Alu operand preparation
        if micro_code(instr).cs_imm1='0' then
            -- Datas from register banks
            PRE_op1 <= data1;
        else
```

```
                -- Immediate datas
                if micro_code(instr).imm1_sel='0' then
                    PRE_op1 <= (others => '0');             -- Immediate operand = 0
                else
                    PRE_op1 <= X"000000" & "000" & shamt;   -- Immediate operand = shamt
                end if;
            end if;


            if micro_code(instr).cs_imm2='0' then
                -- Datas from register banks
                PRE_op2 <= data2;
            else
                -- Immediate datas
                if micro_code(instr).imm2_sel='0' then
                    PRE_op2 <= X"0000" & imm;               -- Immediate operand = imm
                else
                    if imm(15)='1' then                     -- Immediate operand =
sgn_ext(imm)
                        PRE_op2 <= X"FFFF" & imm;
                    else
                        PRE_op2 <= X"0000" & imm;
                    end if;
                end if;
            end if;

            -- Selection of destination register address
            case micro_code(instr).des_sel is
                when D_RT => PRE_adr_reg_dest <= micro_code(instr).bank_des & rt;
                when D_RD => PRE_adr_reg_dest <= micro_code(instr).bank_des & rd;
                when D_31 => PRE_adr_reg_dest <= micro_code(instr).bank_des & "11111";
                when D_00 => PRE_adr_reg_dest <= micro_code(instr).bank_des & "00000";
            end case;

            -- Command signal affectation
            PRE_bra       <= micro_code(instr).bra;        -- Branch operation
            PRE_link      <= micro_code(instr).link;       -- Branch with link
            PRE_code_ual  <= micro_code(instr).code_ual;   -- Alu operation
            PRE_ecr_reg   <= micro_code(instr).ecr_reg;    -- Writing the result in a bank
register
            PRE_mode      <= micro_code(instr).mode;       -- Type of calculation for the
address with current pc
            PRE_op_mem    <= micro_code(instr).op_mem;     -- Memory operation needed
            PRE_r_w       <= micro_code(instr).r_w;        -- Read/Write in memory selection
            PRE_exc_cause <= micro_code(instr).exc_cause;  -- Potential cause exception
            PRE_level     <= micro_code(instr).level;

            -- Set asynchronous outputs
            adr_reg1 <= micro_code(instr).cop_org1 & rs; -- First operand register address
            adr_reg2 <= micro_code(instr).cop_org2 & rt; -- Second operand register address
            bra_detect <= micro_code(instr).bra;         -- Branch detection in current
instruction
            use1 <= not micro_code(instr).cs_imm1;       -- Effective use of operande 1
            use2 <= not micro_code(instr).cs_imm2;       -- Effective use of operande 2
        end if;

    end process;
```

```
    -- Set the synchronous outputs
    process (clock)
    begin
        if clock='1' and clock'event then
            if reset='1' then
                DI_bra <= '0';
                DI_link <= '0';
                DI_op1 <= (others => '0');
                DI_op2 <= (others => '0');
                DI_code_ual <= OP_OUI;
                DI_offset <= (others => '0');
                DI_adr_reg_dest <= (others => '0');
                DI_ecr_reg <= '0';
                DI_mode <= '0';
                DI_op_mem <= '0';
                DI_r_w <= '0';
                DI_adr <= (others => '0');
                DI_exc_cause <= IT_NOEXC;
                DI_level <= LVL_DI;
                DI_it_ok <= '0';
            elsif stop_all='0' then
                if clear='1' or stop_di='1' then
                    -- Nop instruction
                    DI_bra <= '0';
                    DI_link <= '0';
                    DI_op1 <= (others => '0');
                    DI_op2 <= (others => '0');
                    DI_code_ual <= OP_OUI;
                    DI_offset <= (others => '0');
                    DI_adr_reg_dest <= (others => '0');
                    DI_ecr_reg <= '0';
                    DI_mode <= '0';
                    DI_op_mem <= '0';
                    DI_r_w <= '0';
                    DI_adr <= EI_adr;
                    DI_exc_cause <= IT_NOEXC;
                    DI_level <= LVL_DI;
                    if clear='1' then
                      DI_it_ok <= '0';
                    else
                      DI_it_ok <= EI_it_ok;
                    end if;
                else -- Noraml step
                    DI_bra <= PRE_bra;
                    DI_link <= PRE_link;
                    DI_op1 <= PRE_op1;
                    DI_op2 <= PRE_op2;
                    DI_code_ual <= PRE_code_ual;
                    DI_offset <= PRE_offset;
                    DI_adr_reg_dest <= PRE_adr_reg_dest;
                    DI_ecr_reg <= PRE_ecr_reg;
                    DI_mode <= PRE_mode;
                    DI_op_mem <= PRE_op_mem;
                    DI_r_w <= PRE_r_w;
                    DI_adr <= EI_adr;
                    DI_exc_cause <= PRE_exc_cause;
                    DI_level <= PRE_level;
                    DI_it_ok <= EI_it_ok;
                end if;
            end if;
        end if;
    end process;
end rtl;
```

----------pps_ex.vhd : 4th Level pipeline----------

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.pack_mips.all;
use work.alu;

entity pps_ex is
port(
    clock : in std_logic;
    reset : in std_logic;
    stop_all : in std_logic;            -- Unconditionnal locking of outputs
    clear : in std_logic;               -- Clear the pipeline stage

    -- Datas from DI stage
    DI_bra : in std_logic;              -- Branch instruction
    DI_link : in std_logic;             -- Branch with link
    DI_op1 : in bus32;                  -- Operand 1 for alu
    DI_op2 : in bus32;                  -- Operand 2 for alu
    DI_code_ual : in alu_ctrl_type;     -- Alu operation
    DI_offset : in bus32;               -- Offset for address calculation
    DI_adr_reg_dest : in adr_reg_type;  -- Destination register address for the result
    DI_ecr_reg : in std_logic;          -- Effective writing of the result
    DI_mode : in std_logic;             -- Address mode (relative to pc ou index by a
register)
    DI_op_mem : in std_logic;           -- Memory operation
    DI_r_w : in std_logic;              -- Type of memory operation (read or write)
    DI_adr : in bus32;                  -- Instruction address
    DI_exc_cause : in bus32;            -- Potential cause exception
    DI_level : in level_type;           -- Availability stage of the result for bypassing
    DI_it_ok : in std_logic;            -- Allow hardware interruptions

    -- Synchronous outputs to MEM stage
    EX_adr : out bus32;                 -- Instruction address
    EX_bra_confirm : out std_logic;     -- Branch execution confirmation
    EX_data_ual : out bus32;            -- Ual result
    EX_adresse : out bus32;             -- Address calculation result
    EX_adr_reg_dest : out adr_reg_type; -- Destination register for the result
    EX_ecr_reg : out std_logic;         -- Effective writing of the result
    EX_op_mem : out std_logic;          -- Memory operation needed
    EX_r_w : out std_logic;             -- Type of memory operation (read or write)
    EX_exc_cause : out bus32;           -- Potential cause exception
    EX_level : out level_type;          -- Availability stage of result for bypassing
    EX_it_ok : out std_logic            -- Allow hardware interruptions
);
end entity;


architecture rtl of pps_ex is

component alu
    port (
        clock : in std_logic;
        reset : in std_logic;
        op1 : in bus32;                      -- Operand 1
        op2 : in bus32;                      -- Operand 2
        ctrl : in alu_ctrl_type;               -- Operation

        res : out bus32;                 -- Result
        overflow : out std_logic                 -- Overflow
    );
```

```
    end component;

    signal res_ual         : bus32;      -- Alu result output
    signal base_adr        : bus32;      -- Output of the address mode mux selection

    signal pre_ecr_reg     : std_logic;  -- Output of mux selection for writing command to
register
    signal pre_data_ual    : bus32;      -- Mux selection of the data to write
    signal pre_bra_confirm : std_logic;  -- Result of the test in alu when branch
instruction
    signal pre_exc_cause   : bus32;      -- Preparation of the exception detection signal
    signal overflow_ual    : std_logic;  -- Dectection of the alu overflow

begin

    -- Alu instantiation
    U1_alu : alu port map (clock => clock, reset => reset, op1=>DI_op1, op2=>DI_op2,
ctrl=>DI_code_ual,
                           res=>res_ual, overflow=>overflow_ual);

    -- Calculation of the future outputs
    base_adr <= DI_op1 when DI_mode='0' else DI_adr;
    pre_ecr_reg <= DI_ecr_reg when DI_link='0' else pre_bra_confirm;
    pre_data_ual <= res_ual when DI_link='0' else bus32(unsigned(DI_adr) + 4);
    pre_bra_confirm <= DI_bra and res_ual(0);
    pre_exc_cause <= DI_exc_cause when DI_exc_cause/=IT_NOEXC else
                     IT_OVERF when overflow_ual='1' else
                     IT_NOEXC;

    -- Set the synchronous outputs
    process(clock) is
    begin
        if clock='1' and clock'event then
            if reset='1' then
                EX_adr <= (others => '0');
                EX_bra_confirm <= '0';
                EX_data_ual <= (others => '0');
                EX_adresse <= (others => '0');
                EX_adr_reg_dest <= (others => '0');
                EX_ecr_reg <= '0';
                EX_op_mem <= '0';
                EX_r_w <= '0';
                EX_exc_cause <= IT_NOEXC;
                EX_level <= LVL_DI;
                EX_it_ok <= '0';
            elsif stop_all = '0' then
                if clear = '1' then -- Clear the stage
                    EX_adr <= DI_adr;
                    EX_bra_confirm <= '0';
                    EX_data_ual <= (others => '0');
                    EX_adresse <= (others => '0');
                    EX_adr_reg_dest <= (others => '0');
                    EX_ecr_reg <= '0';
                    EX_op_mem <= '0';
                    EX_r_w <= '0';
                    EX_exc_cause <= IT_NOEXC;
                    EX_level <= LVL_DI;
                    EX_it_ok <= '0';
                else -- Normal evolution
                    EX_adr <= DI_adr;
                    EX_bra_confirm <= pre_bra_confirm;
                    EX_data_ual <= pre_data_ual;
                    EX_adr_reg_dest <= DI_adr_reg_dest;
```

```
                    EX_ecr_reg <= pre_ecr_reg;
                    EX_op_mem <= DI_op_mem;
                    EX_r_w <= DI_r_w;
                    EX_exc_cause <= pre_exc_cause;
                    EX_level <= DI_level;
                    EX_it_ok <= DI_it_ok;
                  EX_adresse <= bus32(unsigned(DI_offset) + unsigned(base_adr));
                end if;
            end if;
        end if;
    end process;

end architecture;
```

----------pps_mem.vhd ----------

```
library IEEE;
use IEEE.std_logic_1164.all;

library work;
use work.pack_mips.all;

entity pps_mem is
port
(
    clock : in std_logic;
    reset : in std_logic;
    stop_all : in std_logic;             -- Unconditionnal locking of the outputs
    clear : in std_logic;                -- Clear the pipeline stage

    -- Interface with the control bus
    MTC_data : out bus32;                -- Data to write in memory
    MTC_adr : out bus32;                 -- Address for memory
    MTC_r_w : out std_logic;             -- Read/Write in memory
    MTC_req : out std_logic;             -- Request access to memory
    CTM_data : in bus32;                 -- Data from memory

    -- Datas from Execution stage
    EX_adr : in bus32;                   -- Instruction address
    EX_data_ual : in bus32;              -- Result of alu operation
    EX_adresse : in bus32;               -- Result of the calculation of the address
    EX_adr_reg_dest : in adr_reg_type;   -- Destination register address for the result
    EX_ecr_reg : in std_logic;           -- Effective writing of the result
    EX_op_mem : in std_logic;            -- Memory operation needed
    EX_r_w : in std_logic;               -- Type of memory operation (read or write)
    EX_exc_cause : in bus32;             -- Potential exception cause
    EX_level : in level_type;            -- Availability stage for the result for bypassing
    EX_it_ok : in std_logic;             -- Allow hardware interruptions

    -- Synchronous outputs for bypass unit
    MEM_adr : out bus32;                 -- Instruction address
    MEM_adr_reg_dest : out adr_reg_type; -- Destination register address
    MEM_ecr_reg : out std_logic;         -- Writing of the destination register
    MEM_data_ecr : out bus32;            -- Data to write (from alu or memory)
    MEM_exc_cause : out bus32;           -- Potential exception cause
    MEM_level : out level_type;          -- Availability stage for the result for bypassing
    MEM_it_ok : out std_logic            -- Allow hardware interruptions
);
end pps_mem;


architecture rtl of pps_mem is

    signal tmp_data_ecr : bus32;         -- Selection of the data source (memory or alu)
```

```
begin

    -- Bus controler connexions
    MTC_adr <= EX_adresse;              -- Connexion of the address
    MTC_r_w <= EX_r_w;                  -- Connexion of R/W
    MTC_data <= EX_data_ual;            -- Connexion of the data bus
    MTC_req <= EX_op_mem and not clear; -- Connexion of the request (if there is no clearing
of the pipeline)


    -- Preselection of the data source for the outputs
    tmp_data_ecr <= CTM_data when EX_op_mem = '1' else EX_data_ual;


    -- Set the synchronous outputs
    process (clock)
    begin
        if clock = '1' and clock'event then
            if reset = '1' then
                MEM_adr  <= (others => '0');
                MEM_adr_reg_dest <= (others => '0');
                MEM_ecr_reg <= '0';
                MEM_data_ecr <= (others => '0');
                MEM_exc_cause <= IT_NOEXC;
                MEM_level <= LVL_DI;
                MEM_it_ok <= '0';
            elsif stop_all = '0' then
                if clear = '1' then -- Clear the stage
                    MEM_adr <= EX_adr;
                    MEM_adr_reg_dest <= (others => '0');
                    MEM_ecr_reg <= '0';
                    MEM_data_ecr <= (others => '0');
                    MEM_exc_cause <= IT_NOEXC;
                    MEM_level <= LVL_DI;
                    MEM_it_ok <= '0';
                else -- Normal evolution
                    MEM_adr <= EX_adr;
                    MEM_adr_reg_dest <= EX_adr_reg_dest;
                    MEM_ecr_reg <= EX_ecr_reg;
                    MEM_data_ecr <= tmp_data_ecr;
                    MEM_exc_cause <= EX_exc_cause;
                    MEM_level <= EX_level;
                    MEM_it_ok <= EX_it_ok;
                end if;
            end if;
        end if;
    end process;

end rtl;
```

----------renvoi.vhd----------

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.pack_mips.all;

entity renvoi is
port (
    -- Register access signals
    adr1 : in adr_reg_type;    -- Operand 1 address
    adr2 : in adr_reg_type;    -- Operand 2 address
```

```
    use1 : in std_logic;       -- Operand 1 utilisation
    use2 : in std_logic;       -- Operand 2 utilisation

    data1 : out bus32;         -- First register value
    data2 : out bus32;         -- Second register value
    alea : out std_logic;      -- Unresolved hazards detected

    -- Bypass signals of the intermediary datas
    DI_level : in level_type;  -- Availability level of the data
    DI_adr : in adr_reg_type;  -- Register destination of the result
    DI_ecr : in std_logic;     -- Writing register request
    DI_data : in bus32;        -- Data to used

    EX_level : in level_type;  -- Availability level of the data
    EX_adr : in adr_reg_type;  -- Register destination of the result
    EX_ecr : in std_logic;     -- Writing register request
    EX_data : in bus32;        -- Data to used

    MEM_level : in level_type; -- Availability level of the data
    MEM_adr : in adr_reg_type; -- Register destination of the result
    MEM_ecr : in std_logic;    -- Writing register request
    MEM_data : in bus32;       -- Data to used

    interrupt : in std_logic;  -- Exceptions or interruptions

    -- Connexion to the differents bank of register

      -- Writing commands for writing in the registers
    write_data : out bus32;    -- Data to write
    write_adr : out bus5;      -- Address of the register to write
    write_GPR : out std_logic; -- Selection in the internal registers
    write_SCP : out std_logic; -- Selection in the coprocessor system registers

      -- Reading commands for Reading in the registers
    read_adr1 : out bus5;      -- Address of the first register to read
    read_adr2 : out bus5;      -- Address of the second register to read
    read_data1_GPR : in bus32; -- Value of operand 1 from the internal registers
    read_data2_GPR : in bus32; -- Value of operand 2 from the internal registers
    read_data1_SCP : in bus32; -- Value of operand 1 from the coprocessor system registers
    read_data2_SCP : in bus32  -- Value of operand 2 from the coprocessor system registers
);
end renvoi;

architecture rtl of renvoi is
    signal dep_r1 : level_type; -- Dependency level for operand 1
    signal dep_r2 : level_type; -- Dependency level for operand 2
    signal read_data1 : bus32;  -- Data contained in the register asked by operand 1
    signal read_data2 : bus32;  -- Data contained in the register asked by operand 2
    signal res_reg, res_mem, res_ex, res_di : std_logic;
    signal resolution : bus4;   -- Verification of the resolved hazards

    signal idx1, idx2 : integer range 0 to 3;
begin

    -- Connexion of the writing command signals
    write_data <= MEM_data;
    write_adr <= MEM_adr(4 downto 0);
    write_GPR <= not MEM_adr(5) and MEM_ecr when interrupt = '0' else  -- The high bit to 0
selects the internal registers
                 '0';
    write_SCP <= MEM_adr(5) and MEM_ecr;      -- The high bit to 1 selects the coprocessor
system registers
```

```
    -- Connexion of the writing command signals
    read_adr1 <= adr1(4 downto 0);            -- Connexion of the significative address bits
    read_adr2 <= adr2(4 downto 0);            -- Connexion of the significative address bits

    -- Evaluation of the level of dependencies
    dep_r1 <= LVL_REG when adr1(4 downto 0)="00000" or use1='0' else -- No dependency with
register 0
              LVL_DI  when adr1=DI_adr  and DI_ecr ='1' else         -- Dependency with DI
stage
              LVL_EX  when adr1=EX_adr  and EX_ecr ='1' else         -- Dependency with DI
stage
              LVL_MEM when adr1=MEM_adr and MEM_ecr='1' else         -- Dependency with DI
stage
              LVL_REG;                                               -- No dependency
detected

    dep_r2 <= LVL_REG when adr2(4 downto 0)="00000" or use2='0' else -- No dependency with
register 0
              LVL_DI  when adr2=DI_adr  and DI_ecr ='1' else         -- Dependency with DI
stage
              LVL_EX  when adr2=EX_adr  and EX_ecr ='1' else         -- Dependency with DI
stage
              LVL_MEM when adr2=MEM_adr and MEM_ecr='1' else         -- Dependency with DI
stage
              LVL_REG;                                               -- No dependency
detected

    -- Elaboration of the signals with the datas form the bank registers
    read_data1 <= read_data1_GPR when adr1(5)='0' else       -- Selection of the internal
registers
                  read_data1_SCP when adr1(5)='1' else       -- Selection of the coprocessor
registers
                  (others => '0');

    read_data2 <= read_data2_GPR when adr2(5)='0' else       -- Selection of the internal
registers
                  read_data2_SCP when adr2(5)='1' else       -- Selection of the coprocessor
registers
                  (others => '0');

    -- Bypass the datas (the validity is tested later when detecting the hazards)
    data1 <= read_data1 when dep_r1=LVL_REG else
             MEM_data   when dep_r1=LVL_MEM else
             EX_data    when dep_r1=LVL_EX  else
             DI_data;

    data2 <= read_data2 when dep_r2=LVL_REG else
             MEM_data   when dep_r2=LVL_MEM else
             EX_data    when dep_r2=LVL_EX  else
             DI_data;

    -- Detection of a potential unresolved hazard
    res_reg <= '1'; -- This hazard is always resolved
    res_mem <= '1' when MEM_level>=LVL_MEM else '0';
    res_ex  <= '1' when EX_level >=LVL_EX  else '0';
    res_di  <= '1' when DI_level >=LVL_DI  else '0';

    -- Table defining the resolved hazard for each stage
    resolution <= res_di & res_ex & res_mem & res_reg;

    -- Verification of the validity of the transmitted datas (test the good resolution of
the hazards)
    idx1 <= to_integer(unsigned(dep_r1(1 downto 0)));
```

```
    idx2 <= to_integer(unsigned(dep_r2(1 downto 0)));
    alea <= not resolution(idx1) or not resolution(idx2);

end rtl;
```

----------syscop.vhd----------

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.pack_mips.all;

-- By convention in the commentary, the term interruption means hardware interruptions and
software exceptions

entity syscop is
port
(
    clock         : in std_logic;
    reset         : in std_logic;

    -- Datas from the pipeline
    MEM_adr       : in bus32;       -- Address of the current instruction in the pipeline
end -> responsible of the exception
    MEM_exc_cause : in bus32;       -- Potential cause exception of that instruction
    MEM_it_ok     : in std_logic;   -- Allow hardware interruptions

    -- Hardware interruption
    it_mat        : in std_logic;   -- Hardware interruption detected

    -- Interruption controls
    interrupt     : out std_logic;  -- Interruption to take into account
    vecteur_it    : out bus32;      -- Interruption vector

    -- Writing request in register bank
    write_data    : in bus32;       -- Data to write
    write_adr     : in bus5;        -- Address of the register to write
    write_SCP     : in std_logic;   -- Writing request

    -- Reading request in register bank
    read_adr1     : in bus5;        -- Address of the first register
    read_adr2     : in bus5;        -- Address of the second register
    read_data1    : out bus32;      -- Value of register 1
    read_data2    : out bus32       -- Value of register 2
);
end syscop;


architecture rtl of syscop is

    subtype adr_scp_reg is integer range 12 to 15;

    type scp_reg_type is array (integer range adr_scp_reg'low to adr_scp_reg'high) of bus32;

    -- Constants to define the coprocessor registers
    constant COMMAND   : integer      := 0;   -- False register to command the coprocessor
system
    constant STATUS    : adr_scp_reg  := 12;  -- Registre 12 of the coprocessor system
    constant CAUSE     : adr_scp_reg  := 13;  -- Registre 13 of the coprocessor system
    constant ADRESSE   : adr_scp_reg  := 14;  -- Registre 14 of the coprocessor system
    constant VECTIT    : adr_scp_reg  := 15;  -- Registre 15 of the coprocessor system
```

```
    signal scp_reg : scp_reg_type;          -- Internal register bank
    signal pre_reg : scp_reg_type;          -- Register bank preparation

    signal adr_src1 : integer range 0 to 31;
    signal adr_src2 : integer range 0 to 31;
    signal adr_dest : integer range 0 to 31;

    signal exception : std_logic;           -- Set to '1' when exception detected
    signal interruption : std_logic;        -- Set to '1' when interruption detected
    signal cmd_itret    : std_logic;        -- Set to '1' when interruption return command
is detected

    signal save_msk  : std_logic;           -- Save the mask state when an interruption
occurs

begin

    -- Detection of the interruptions
    exception <= '1' when MEM_exc_cause/=IT_NOEXC else '0';
    interruption <= '1' when it_mat='1' and scp_reg(STATUS)(0)='1' and MEM_it_ok='1' else
'0';

    -- Update asynchronous outputs
    interrupt <= exception or interruption; -- Detection of interruptions
    vecteur_it <= scp_reg(ADRESSE) when cmd_itret = '1' else -- Send the return adress when
a return instruction appears
                  scp_reg(VECTIT);                          -- Send the interruption vector
in other cases


    -- Decode the address of the registers
    adr_src1 <= to_integer(unsigned(read_adr1));
    adr_src2 <= to_integer(unsigned(read_adr2));
    adr_dest <= to_integer(unsigned(write_adr));

    -- Read the two registers
    read_data1 <= (others => '0') when (adr_src1<scp_reg'low or adr_src1>scp_reg'high) else
                  scp_reg(adr_src1);
    read_data2 <= (others => '0') when adr_src2<scp_reg'low or adr_src2>scp_reg'high else
                  scp_reg(adr_src2);

    -- Define the pre_reg signal, next value for the registers
    process (scp_reg, adr_dest, write_SCP, write_data, interruption,
             exception, MEM_exc_cause, MEM_adr, reset)
    begin
        pre_reg <= scp_reg;
        cmd_itret <= '0'; -- No IT return in most cases

        -- Potential writing in a register
        if (write_SCP='1' and adr_dest>=pre_reg'low and adr_dest<=pre_reg'high) then
            pre_reg(adr_dest) <= write_data;
        end if;

        -- Command from the core
        if write_SCP='1' and adr_dest=COMMAND then
            case write_data is -- Different operations
                when SYS_UNMASK => pre_reg(STATUS)(0) <= '1'; -- Unamsk command
                when SYS_MASK   => pre_reg(STATUS)(0) <= '0'; -- Mask command
                when SYS_ITRET  => -- Interruption return command
                                   pre_reg(STATUS)(0) <= save_msk; -- Restore the mask
before the interruption
                                   cmd_itret          <= '1';      -- False interruption
request (to clear the pipeline)
```

```
                when others     => null;
            end case;
        end if;

        -- Modifications from the interruptions
        if interruption='1' then
            pre_reg(STATUS)(0) <= '0';       -- Mask the interruptions
            pre_reg(CAUSE) <= IT_ITMAT;      -- Save the interruption cause
            pre_reg(ADRESSE) <= MEM_adr;     -- Save the return address
        end if;

        -- Modifications from the exceptions
        if exception='1' then
            pre_reg(STATUS)(0) <= '0';       -- Mask the interruptions
            pre_reg(CAUSE) <= MEM_exc_cause; -- Save the exception cause
            pre_reg(ADRESSE) <= MEM_adr;     -- Save the return address
        end if;

        -- The reset has the priority on the other cuases
        if reset='1' then
            pre_reg <= (others => (others => '0'));
            -- NB : The processor is masked after a reset
            --      The exception handler is set at address 0
        end if;
    end process;

    -- Memorisation of the modifications in the register bank
    process(clock)
    begin
        if clock='1' and clock'event then

            -- Save the mask when an interruption appears
            if (exception='1') or (interruption='1') then
                save_msk <= scp_reg(STATUS)(0);
            end if;

            scp_reg <= pre_reg;

        end if;
    end process;

end rtl;
```

----------alu.vhd----------

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

library work;
use work.pack_mips.all;

entity alu is
port
(
    clock : in std_logic;
    reset : in std_logic;
    op1 : in bus32;             -- Operand 1
    op2 : in bus32;             -- Operand 2
    ctrl : in alu_ctrl_type;    -- Opearator control

    res : out bus32;            -- The result is 32 bit long
    overflow : out std_logic        -- Overflow of the result
);
```

```
end alu;


architecture rtl of alu is

    -- Signals to pre-process the operands
    signal efct_op1, efct_op2 : bus33;        -- Effective operands of the adder (33 bits)
    signal comp_op2 : std_logic;                  -- Select the opposite of operand 2
    signal igno_op2 : std_logic;                  -- Ignore op 2 (put zeros)
    signal sign_op1 : std_logic;                  -- High bit of op 1
    signal sign_op2 : std_logic;                  -- High bit of op 2
    signal signe : std_logic;                     -- Signed operation (bit sign extension)
    signal shift_val : natural range 0 to 31; -- Value of the shift

    -- Signals for internal results
    signal res_shl, res_shr : bus32; -- Results of left and right shifter
    signal res_lui : bus32;          -- Result of Load Upper Immediate
    signal res_add : bus33;          -- Result of the adder
    signal carry : bus33;            -- Carry for the adder
    signal nul : std_logic;              -- Check if the adder result is zero
    signal hilo : bus64;             -- Internal registers to store the multiplication
operation
    signal tmp_hilo : bus64;         -- Internal registers to store the multiplication
operation (synchronised)

begin

    --  Process if the operation is signed compliant
    signe <= '1' when (ctrl=OP_ADD or ctrl=OP_SUB or ctrl=OP_SLT or ctrl=OP_SNEG or
ctrl=OP_SPOS or ctrl=OP_LNEG or ctrl=OP_LPOS)
                 else
            '0';

    sign_op1 <= signe and op1(31);
    sign_op2 <= signe and op2(31);

    -- Selection of the value of the second operand : op2 or -op2 (ie not op2 + 1)
    comp_op2 <= '1' when -- The opposite of op2 is used
                         (ctrl=OP_SUB or ctrl=OP_SUBU) -- Opposite of the operand 2 to
obtain a substraction
                      or (ctrl=OP_SLT or ctrl=OP_SLTU) -- Process the difference to check
the lesser than operation for the operands
                      or (ctrl=OP_EQU or ctrl=OP_NEQU) -- Process the difference to check
the equality of the operands
                    else
                '0'; -- by default, op2 is used

    igno_op2 <= '1' when -- Op 2 will be zero (when comp_op2='0')
                         (ctrl=OP_SPOS or ctrl=OP_LNEG) -- Process if the op1 is nul with
op1+0
                    else
                '0';

    -- Effective signals for the adder
    efct_op2 <= not (sign_op2 & op2) when (comp_op2='1') else -- We take the opposite of op2
to get -op2 (we will add 1 with the carry)
                (others => '0')  when (igno_op2='1') else     -- Op2 is zero
                (sign_op2 & op2);                     -- by default we use op2 (33 bits long)

    efct_op1 <= sign_op1 & op1;

    -- Execution of the addition
    carry <= X"00000000" & comp_op2; -- Carry to one when -op2 is needed
```

```
    res_add <= std_logic_vector(unsigned(efct_op1) + unsigned(efct_op2) + unsigned(carry));
    -- Check the nullity of the result
    nul <= '1' when (res_add(31 downto 0)=X"00000000") else '0';
    -- Value of the shift for the programmable shifter
    shift_val <= to_integer(unsigned(op1(4 downto 0)));

    res_shl <= bus32(shift_left(unsigned(op2), shift_val));
    res_shr <= not bus32(shift_right(unsigned(not op2) , shift_val)) when (ctrl=OP_SRA and
op2(31)='1') else
               bus32(shift_right(unsigned(op2), shift_val));
    res_lui <= op2(15 downto 0) & X"0000";

    -- Affectation of the hilo register if necessary
    tmp_hilo <= std_logic_vector(signed(op1)*signed(op2)) when (ctrl=OP_MULT) else
                std_logic_vector(unsigned(op1)*unsigned(op2)) when (ctrl=OP_MULTU) else
                op1 & hilo(31 downto 0) when (ctrl=OP_MTHI) else
                hilo(63 downto 32) & op1 when (ctrl=OP_MTLO) else
                (others => '0');

    -- Check the overflows
    overflow <= '1' when ((ctrl=OP_ADD and op1(31)=efct_op2(31) and op1(31)/=res_add(31))
                        or  (ctrl=OP_SUB and op1(31)/=op2(31) and op1(31)/=res_add(31))) else
                '0'; -- Only ADD and SUB can overflow

    -- Result affectation
    res <=
        -- Arithmetical operations
        res_add(31 downto 0)                        when (ctrl=OP_ADD or ctrl=OP_ADDU or
ctrl=OP_SUB or ctrl=OP_SUBU) else
        -- Logical operations
        op1 and op2                                 when (ctrl=OP_AND)  else
        op1 or op2                                  when (ctrl=OP_OR)   else
        op1 nor op2                                 when (ctrl=OP_NOR)  else
        op1 xor op2                                 when (ctrl=OP_XOR)  else
        -- Different tests : the result is one when the test is succesful
        (0 => res_add(32), others=>'0')             when (ctrl=OP_SLTU or ctrl=OP_SLT) else
        (0 => nul, others=>'0')                     when (ctrl=OP_EQU)  else
        (0 => not nul, others=>'0')                 when (ctrl=OP_NEQU) else
        (0 => op1(31), others=>'0')                 when (ctrl=OP_SNEG) else
        (0 => not (op1(31) or nul), others=>'0')    when (ctrl=OP_SPOS) else
        (0 => (op1(31) or nul), others=>'0')        when (ctrl=OP_LNEG) else
        (0 => not op1(31), others=>'0')             when (ctrl=OP_LPOS) else
        -- Shifts
        res_shl                                     when (ctrl=OP_SLL)  else
        res_shr                                     when (ctrl=OP_SRL or ctrl=OP_SRA)  else
        res_lui                                     when (ctrl=OP_LUI)  else
        -- Internal registers
        hilo(63 downto 32)                          when (ctrl=OP_MFHI) else
        hilo(31 downto 0)                           when (ctrl=OP_MFLO or ctrl=OP_MULT or
ctrl=OP_MULTU) else
        op1                                         when (ctrl=OP_MTHI or ctrl=OP_MTLO) else
        op2                                         when (ctrl=OP_OP2) else
        -- Always true
        X"00000001"                                 when (ctrl=OP_OUI) else
        -- Unknown operation or nul result desired
        (others => '0');

    -- Save the hilo register
    process (clock)
    begin
        if clock = '1' and clock'event then
            if reset = '1' then
                hilo <= (others => '0');
```

```
            elsif (ctrl = OP_MULT) or (ctrl = OP_MULTU) or (ctrl = OP_MTLO) or (ctrl =
OP_MTHI) then
                  hilo <= tmp_hilo;
            end if;
        end if;
    end process;
end rtl;
```

----------banc.vhd----------

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

library work;
use work.pack_mips.all;

entity banc is
port (
       clock : in std_logic;
       reset : in std_logic;

       -- Register addresses to read
       reg_src1 : in bus5;
       reg_src2 : in bus5;

       -- Register address to write and its data
       reg_dest : in bus5;
       donnee   : in bus32;

       -- Write signal
       cmd_ecr  : in std_logic;

       -- Bank outputs
       data_src1 : out bus32;
       data_src2 : out bus32;

       stop_all  : out std_logic
     );
end banc;


architecture rtl of banc is

    -- The register bank
    type tab_reg is array (1 to 31) of bus32;
    signal registres : tab_reg;

    signal adr_src1 : integer range 0 to 31;
    signal adr_src2 : integer range 0 to 31;
    signal adr_dest : integer range 0 to 31;

    signal wait_reg : bus32;  -- register 31 is wait reg
    signal wr_val   : bus32;  -- save value waiting to be written to WR
    signal wr_ecr, wr_dec : std_logic;
    signal wr_src : std_logic;

    signal cs : std_logic;
begin

    adr_src1 <= to_integer(unsigned(reg_src1));
    adr_src2 <= to_integer(unsigned(reg_src2));
    adr_dest <= to_integer(unsigned(reg_dest));
```

```
data_src1 <= (others => '0') when adr_src1=0 else
             wait_reg  when adr_src1=31 else
              registres(adr_src1);

data_src2 <= (others => '0') when adr_src2=0 else
              wait_reg when adr_src2=31 else
              registres(adr_src2);

-- handle regular register writing
process(clock)
begin
    if clock = '1' and clock'event then
        if reset='1' then
            for i in 1 to 30 loop
                registres(i) <= (others => '0');
            end loop;
        else
            if cmd_ecr = '1' and adr_dest /= 0 and adr_dest /= 31 then      --write req
                registres(adr_dest) <= donnee;
            end if;
        end if;
    end if;
end process;



-- handle stop_all condition and setup new value to write
process(reset,wr_ecr, wr_dec)
begin
        if reset = '1' then
            cs <= '0';
            stop_all <= '0';
            wr_val  <= X"00000000";
        else
            case cs is
                when '0' =>
                    wr_src <= '0';  -- store input
                    if wr_ecr = '1' and wr_dec = '1' then
                        wr_val <= donnee;
                        stop_all <= '1';
                        cs <= '1';
                    end if;
                when '1' =>
                    wr_src <= '1';  -- store register
                    if wr_dec = '0' then
                        stop_all <= '0';
                        cs <= '0';
                    end if;
                when others =>
                    stop_all <= '0';
                    cs <= '0';
            end case;
        end if;
end process;
                    --          -- means there was a previous write req
wr_ecr <= '1' when (cmd_ecr = '1' and adr_dest = 31) or  cs = '1' else
          '0';

wr_dec <= '0' when wait_reg = X"000000" else
          '1';
```

```
    -- wait register controller
    process(clock,reset,wr_ecr, wr_dec)
    begin
        if clock = '1' and clock'event then
            if reset = '1' then
                wait_reg <= (others =>'0');
            elsif wr_ecr = '1' and wr_dec = '0' then
                if wr_src = '0' then
                    wait_reg <= donnee;
                else
                    wait_reg <= wr_val;
                end if;
            elsif wr_dec = '1' then
                wait_reg <= wait_reg - 1;
            end if;
        end if;
    end process;

end rtl;
```

----------bus_ctrl.vhd----------

```
-------------------------------------------------------------
-- Data and Instruction RAM      == 00000000 - 000000FF
-- Character RAM           == 00001000 - 000019FF
-- Font RAM                == 00002000 - 000025FF
-------------------------------------------------------------

--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

library work;
use work.pack_mips.all;

entity bus_ctrl is
port
(
    clock : std_logic;
    reset : std_logic;

    -- Interruption in the pipeline
    interrupt      : in std_logic;

    -- Interface for the Instruction Extraction Stage
    adr_from_ei    : in bus32;      -- The address of the data to read
    instr_to_ei    : out bus32;     -- Instruction from the memory

    -- Interface with the MEMory Stage
    req_from_mem   : in std_logic;  -- Request to access the ram
    r_w_from_mem   : in std_logic;  -- Read/Write request
    adr_from_mem   : in bus32;      -- Address in ram
    data_from_mem  : in bus32;      -- Data to write in ram
    data_to_mem    : out bus32;     -- Data from the ram to the MEMory stage

    load_to_video  : out std_logic;
    vsync_to_video : out std_logic;
```

```
    hsync_to_video : out std_logic;
    blank_to_video : out std_logic;
    byte_to_video  : out bus8;

    -- Pipeline progress control signal
    stop_all       : in std_logic
);

end bus_ctrl;

architecture Behavioral of bus_ctrl is


------------------------------------------------------------
-- Data and Instruction RAM      == 00000000 - 000000FF
-- Character RAM          == 00001000 - 000019FF
-- Font RAM               == 00002000 - 000025FF
------------------------------------------------------------


constant videoShiftLoad : bus32 := X"00004000"; -- Addresses to send signal
constant videoBLANK     : bus32 := X"00004001";      -- to video controller
constant videoHSYNC     : bus32 := X"00004002";
constant videoVSYNC     : bus32 := X"00004003";
constant videoHandVSYNC : bus32 := X"00004004";


signal data_out_to_mem, char_out_to_mem : bus32 := X"00000000";
signal data_out_to_ei : bus32;
signal char_out_to_mem0, char_out_to_mem1, char_out_to_mem2 : bus8 := X"00";
signal char_out_to_mem3, char_out_to_mem4 : bus8 := X"00";
signal dataRAM_RST, instrRAM_RST : std_logic := '1';
signal charRAM_RST : bus5 := "11111";
signal fontRAM_RST : std_logic_vector (2 downto 0) := "111";
signal dataRAM_WE : std_logic := '0';
signal charRAM_WE : bus5 := "00000";
--signal fontRAM_WE : std_logic_vector(2 downto 0) := "000";
signal real_adr_from_mem : bus32;
signal font_adr : std_logic_vector(8 downto 0);
signal font_in, font_out0, font_out1, font_out2 : bus8 := X"00";


 component RAMB4_S8
  -- Initialize the RAM (xapp173.pdf)
-- synopsis translate on
       generic(    INIT_00, INIT_01, INIT_02, INIT_03,
                   INIT_04, INIT_05, INIT_06, INIT_07,
                   INIT_08, INIT_09, INIT_0a, INIT_0b,
                   INIT_0c, INIT_0d, INIT_0e,
                   INIT_0f : BIT_VECTOR(255 downto 0) :=
            X"0000000000000000000000000000000000000000000000000000000000000000");
-- synopsis translate off
    port (
      DO   : out std_logic_vector(7 downto 0);
      ADDR : in  std_logic_vector(8 downto 0);
      CLK  : in  std_logic;
      DI   : in  std_logic_vector(7 downto 0);
      EN   : in  std_logic;
      RST  : in  std_logic;
      WE   : in  std_logic);
  end component;
```

```
   component RAMB4_S16_S16
       -- synopsis translate on
        generic(     INIT_00, INIT_01, INIT_02, INIT_03,
                     INIT_04, INIT_05, INIT_06, INIT_07,
                     INIT_08, INIT_09, INIT_0a, INIT_0b,
                     INIT_0c, INIT_0d, INIT_0e,
                     INIT_0f : BIT_VECTOR(255 downto 0) :=
              X"0000000000000000000000000000000000000000000000000000000000000000");
       -- synopsis translate off
    port (
      DOA   : out std_logic_vector (15 downto 0);
      ADDRA : in std_logic_vector (7 downto 0);
      CLKA  : in std_logic;
      DIA   : in std_logic_vector (15 downto 0);
      ENA   : in std_logic;
      RSTA  : in std_logic;
      WEA   : in std_logic;
      DOB   : out std_logic_vector (15 downto 0);
      ADDRB : in std_logic_vector (7 downto 0);
      CLKB  : in std_logic;
      DIB   : in std_logic_vector (15 downto 0);
      ENB   : in std_logic;
      RSTB  : in std_logic;
      WEB   : in std_logic);
  end component;


begin

instrRAM_RST <= adr_from_ei(15) OR adr_from_ei(14) when adr_from_ei(31 downto 16) = X"0000"
else
                                    '1';
real_adr_from_mem <= adr_from_mem;-- when r_w_from_mem = '1' else
--                 ("00" & adr_from_mem(31 downto 2));

char_out_to_mem <= X"000000" & (char_out_to_mem0 or char_out_to_mem1 or char_out_to_mem2 or
char_out_to_mem3 or char_out_to_mem4);

dataRAM_WE   <=  not dataRAM_RST and r_w_from_mem;
charRAM_WE(0)       <= (not charRAM_RST(0)) and r_w_from_mem;
charRAM_WE(1)       <= (not charRAM_RST(1)) and r_w_from_mem;
charRAM_WE(2)       <= (not charRAM_RST(2)) and r_w_from_mem;
charRAM_WE(3)       <= (not charRAM_RST(3)) and r_w_from_mem;
charRAM_WE(4)       <= (not charRAM_RST(4)) and r_w_from_mem;

font_in <= data_from_mem(7 downto 0) - X"20";
font_adr <= font_in(4 downto 0) & real_adr_from_mem(3 downto 0);--font_in(4 downto 0) &
real_adr_from_mem(3 downto 0);


process (real_adr_from_mem, req_from_mem, reset)
begin
    charRAM_RST <= "11111";
    dataRAM_RST <= '1';
    if reset = '0' then
        if req_from_mem = '1' then
            if real_adr_from_mem(31 downto 16) = X"0000" then
                case real_adr_from_mem(15 downto 9) is
                    when "0000000" =>
                        dataRAM_RST <= '0';
                    when "0001000" =>
                        charRAM_RST(0) <= '0';
                    when "0001001" =>
```

```
                        charRAM_RST(1) <= '0';
                    when "0001010" =>
                        charRAM_RST(2) <= '0';
                    when "0001011" =>
                        charRAM_RST(3) <= '0';
                    when "0001100" =>
                        charRAM_RST(4) <= '0';
                    when others =>
                        charRAM_RST <= "11111";
                        dataRAM_RST <= '1';
                end case;
            end if;
        end if;
    end if;
end process;


-- Decode the font address and font data
process (real_adr_from_mem, font_in, reset)
begin
    fontRAM_RST <= "111";
    if reset = '0' then
        if real_adr_from_mem(31 downto 4) = X"0000200" then
            case font_in(7 downto 5) is
                when "000" =>
                    fontRAM_RST(0) <= '0';
                when "001" =>
                    fontRAM_RST(1) <= '0';
                when "010" =>
                    fontRAM_RST(2) <= '0';
                when others =>
                    fontRAM_RST <= "111";
            end case;
        end if;
   end if;
end process;

process (real_adr_from_mem)
begin
  if stop_all = '0' and clock = '1' and clock'event then
    load_to_video  <= '0';
    vsync_to_video <= '0';
    hsync_to_video <= '0';
    blank_to_video <= '0';
    case real_adr_from_mem is
        when videoShiftLoad =>
            load_to_video <= '1';
        when videoBLANK =>
            blank_to_video <= '1';
        when videoHSYNC =>
            hsync_to_video <= '1';
        when videoVSYNC =>
            vsync_to_video <= '1';
        when videoHandVSYNC =>
            vsync_to_video <= '1';
            hsync_to_video <= '1';
        when others =>
    end case;
   end if;
end process;

-------------------------------------------------
-- For a bigger size RAM, we will use 8 1k x 4 --
```

```
-- BRAMs to emulate a 1k x 32 RAM       --
-----------------------------------------------
-- If we need less memory, we could go with    --
-- 4 512 x 8, or 2 256 x 16                    --
-- for a 2k x 32, we could use 16 2k x 2 BRAMs --
-- I'm not sure if there are 16 available, though-
-----------------------------------------------

-------------------------------------------------------------
-- Data and Instruction RAM  == 00000000 - 000000FF
-------------------------------------------------------------

dataRAM_0 : RAMB4_S16_S16
-- synopsis translate on
generic map( INIT_00 => X"AC00201F1C00201F0000AC00201FAC00201FAC00201FAC00201FAC00201F2014",
             INIT_01 => X"AC00AC62201F8C68000000740020AC00201FAC00201F00000000144028222021",
             INIT_02 => X"2825202114A028452042AC00201F201F14A028852084AC62201F8C6820632004",
             INIT_03 => X"00000000000000000000000000000014A028252021AC00201FAC00201F200114A0")
-- synopsis translate off
port map( DOA   => data_out_to_ei(31 downto 16),
          ADDRA => adr_from_ei(9 downto 2),
          CLKA  => clock,
          DIA   => X"0000",
          ENA   => '1',
          RSTA  => instrRAM_RST,
          WEA   => '0',                       -- always reads
          DOB       => data_out_to_mem(31 downto 16),
          ADDRB => real_adr_from_mem(7 downto 0),
          CLKB  => clock,
          DIB   => data_from_mem(31 downto 16),
          ENB   => '1',
          RSTB  => dataRAM_RST,
          WEB   => dataRAM_WE);

dataRAM_1 : RAMB4_S16_S16
-- synopsis translate on
generic map(  INIT_00 =>
X"40020060400202C008204004006040020 2C040020060400202C040040060001E",
              INIT_01 =>
X"400140000008100018120018182040020 02F4002006010200820FFFA001F0001",
              INIT_02 =>
X"001E0001FFE800100001400100100001FFFA00500001400000081000000 10000",
              INIT_03 =>
X"00000000000000000000000000000FFC8000A0001400202C0400200600000FFE4")
-- synopsis translate off
port map( DOA   => data_out_to_ei(15 downto 0),
          ADDRA => adr_from_ei(9 downto 2),
          CLKA  => clock,
          DIA   => X"0000",
          ENA   => '1',
          RSTA  => instrRAM_RST,
          WEA   => '0',                       --always reads
          DOB       => data_out_to_mem(15 downto 0),
          ADDRB => real_adr_from_mem(7 downto 0),
          CLKB  => clock,
          DIB   => data_from_mem(15 downto 0),
          ENB   => '1',
          RSTB  => dataRAM_RST,
          WEB   => dataRAM_WE);


-------------------------------------------------------------
-- Character RAM  == 00001000 - 000019FF
```

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

CHAR_RAM_0 : RAMB4_S8
-- synopsis translate on
generic map(  INIT_00 =>
X"326e616d32656874326c696d45323232323232323232323232323232323232323232")
-- synopsis translate off
port map ( DO   => char_out_to_mem0,
           ADDR => real_adr_from_mem(8 downto 0),
           CLK  => clock,
           DI   => data_from_mem(7 downto 0),
           EN   => '1',
           RST  => charRAM_RST(0),
           WE   => charRAM_WE(0));

CHAR_RAM_1 : RAMB4_S8
-- synopsis translate on
generic map(  INIT_00 =>
X"b7309e101818001818183c3c3c180000000000000000000000000000000000000000")
-- synopsis translate off
port map ( DO   => char_out_to_mem1,
           ADDR => real_adr_from_mem(8 downto 0),
           CLK  => clock,
           DI   => data_from_mem(7 downto 0),
           EN   => '1',
           RST  => charRAM_RST(1),
           WE   => charRAM_WE(1));

CHAR_RAM_2 : RAMB4_S8
-- synopsis translate on
--generic map(  INIT_00 =>
X"b7309e101818001818183c3c3c180000000000000000000000FEDCBA987654321")
-- synopsis translate off
port map ( DO   => char_out_to_mem2,
           ADDR => real_adr_from_mem(8 downto 0),
           CLK  => clock,
           DI   => data_from_mem(7 downto 0),
           EN   => '1',
           RST  => charRAM_RST(2),
           WE   => charRAM_WE(2));

CHAR_RAM_3 : RAMB4_S8
-- synopsis translate on
--generic map(  INIT_00 =>
X"b7309e101818001818183c3c3c180000000000000000000000FEDCBA987654321")
-- synopsis translate off
port map ( DO   => char_out_to_mem3,
           ADDR => real_adr_from_mem(8 downto 0),
           CLK  => clock,
           DI   => data_from_mem(7 downto 0),
           EN   => '1',
           RST  => charRAM_RST(3),
           WE   => charRAM_WE(3));

CHAR_RAM_4 : RAMB4_S8
-- synopsis translate on
--generic map(  INIT_00 =>
X"b7309e101818001818183c3c3c180000000000000000000000FEDCBA987654321")
-- synopsis translate off
port map ( DO   => char_out_to_mem4,
           ADDR => real_adr_from_mem(8 downto 0),
           CLK  => clock,
           DI   => data_from_mem(7 downto 0),
```

```
           EN   => '1',
           RST  => charRAM_RST(4),
           WE   => charRAM_WE(4));


------------------------------------------------------------
-- Font RAM  == 00002000 - 000025FF
------------------------------------------------------------

FONT_RAM_0 : RAMB4_S8
-- synopsis translate on
generic map(  INIT_00 =>
X"0000000018180018181 83c3c3c180000000000000000000000000000000000000",
              INIT_01 =>
X"000000006c6cfe6c6c6cfe6c6c0000000000000000000000000000002466666600",
              INIT_02 =>
X"0000000086c66030180cc6c200000000000010107cd616167cd0d0d67c101000",
              INIT_03 =>
X"0000000000000000000000030181818000000000076ccccccdc76386c6c380000",
              INIT_04 =>
X"0000000030180c0c0c0c0c0c183000000000000000c18303030303030180c0000",
              INIT_05 =>
X"0000000000018187e1818000000000000000000000663cff3c660000000000",
              INIT_06 =>
X"0000000000000000fe000000000000000000000301818180000000000000000000",
              INIT_07 =>
X"0000000000c06030180c060000000000000000000018180000000000000000000",
              INIT_08 =>
X"000000007e1818181818187838180000000000007cc6e6e6d6d6cecec67c0000",
              INIT_09 =>
X"000000007cc60606063c0606c67c000000000000fec6c06030180c06c67c0000",
              INIT_0a =>
X"000000007cc6060606fcc0c0c0fe0000000000001e0c0c0cfecc6c3c1c0c0000",
              INIT_0b =>
X"0000000030303030180c0606c6fe0000000000007cc6c6c6c6fcc0c060380000",
              INIT_0c =>
X"00000000780c0606067ec6c6c67c0000000000007cc6c6c6c67cc6c6c67c0000",
              INIT_0d =>
X"0000000030181800000018180000000000000000001818000000181800000000",
              INIT_0e =>
X"00000000000000fe0000fe000000000000000000060c18306030180c06000000",
              INIT_0f =>
X"0000000018180018181 80cc6c67c0000000000006030180c060c183060000000")
-- synopsis translate off
port map ( DO   => font_out0,
           ADDR => font_adr,
           CLK  => clock,
           DI   => font_in,
           EN   => '1',
           RST  => fontRAM_RST(0),
           WE   => '0');

FONT_RAM_1 : RAMB4_S8
-- synopsis translate on
generic map(  INIT_00 =>
X"00000000c6c6c6c6fec6c66c38100000000000007cc0dcdededec6c6c67c0000",
              INIT_01 =>
X"000000003c66c2c0c0c0c0c2663c000000000000fc666666667c666666fc0000",
              INIT_02 =>
X"00000000fe6662606878686266fe000000000000f86c6666666666666cf80000",
              INIT_03 =>
X"000000003a66c6c6dec0c0c2663c000000000000f06060606878686266fe0000",
              INIT_04 =>
```

```
X"000000003c18181818181818183c000000000000c6c6c6c6c6fec6c6c6c60000",
              INIT_05 =>
X"00000000e666666c78786c6666e600000000000078cccccc0c0c0c0c0c1e0000",
              INIT_06 =>
X"00000000c6c6c6c6c6d6fefeeec6000000000000fe6662606060606060f00000",
              INIT_07 =>
X"000000007cc6c6c6c6c6c6c6c67c000000000000c6c6c6c6cedefef6e6c60000",
              INIT_08 =>
X"00000e0c7cded6c6c6c6c6c6c67c000000000000f0606060607c666666fc0000",
              INIT_09 =>
X"000000007cc6c6060c3860c6c67c000000000000e66666666c7c666666fc0000",
              INIT_0a =>
X"000000007cc6c6c6c6c6c6c6c6c6c60000000000000003c1818181818185a7e7e0000",
              INIT_0b =>
X"000000006ceefed6d6d6c6c6c6c6c6000000000000010386cc6c6c6c6c6c6c60000",
              INIT_0c =>
X"000000003c181818183c6666666600000000000c6c66c7c38387c6cc6c60000",
              INIT_0d =>
X"000000003c303030303030303c000000000000fec6c26030180c86c6fe0000",
              INIT_0e =>
X"000000003c0c0c0c0c0c0c0c0c3c0000000000000000060c183060c00000000000",
              INIT_0f =>
X"00ff0000000000000000000000000000000000000000000000000000c66c3810")
-- synopsis translate off
port map ( DO   => font_out1,
           ADDR => font_adr,
           CLK  => clock,
           DI   => font_in,
           EN   => '1',
           RST  => fontRAM_RST(1),
           WE   => '0');

FONT_RAM_2 : RAMB4_S8
-- synopsis translate on
generic map(  INIT_00 =>
X"0000000076cccccc7c0c7800000000000000000000000000000001830303000",
              INIT_01 =>
X"000000007cc6c0c0c0c67c0000000000000000007c666666666c786060e00000",
              INIT_02 => X"000000007cc6c0c0fec67c0000000000000000000076ccccccccc6c3c0c0c1c0000"
,
              INIT_03 => X"0078cc0c7ccccccccccc760000000000000000000f060606060f060646c380000"
,
              INIT_04 => X"000000003c1818181818380018180000000000000e666666666766c6060e00000"
,
              INIT_05 => X"00000000e6666c78786c666060e00000003c66660606060606060e0006060000"
,
              INIT_06 => X"00000000c6d6d6d6d6d6feec0000000000000000000018343030303030303030700000"
,
              INIT_07 => X"000000007cc6c6c6c6c67c0000000000000000000666666666666dc0000000000"
,
              INIT_08 => X"001e0c0c7ccccccccccc76000000000000f060607c6666666666dc0000000000"
,
              INIT_09 => X"000000007cc60c3860c67c0000000000000000000f06060606676dc0000000000"
,
              INIT_0A => X"0000000076ccccccccccc0000000000000000000001c3630303030fc3030100000"
,
              INIT_0B => X"000000006cfed6d6d6c6c60000000000000000000183c66666666600000000000"
,
              INIT_0C => X"00f80c067ec6c6c6c6c6c60000000000000000000c66c3838386cc60000000000"
,
              INIT_0D => X"000000000e18181818701818180e000000000000fec6603018ccfe0000000000"
,
              INIT_0E => X"00000000701818181806e1818187000000000000001818181818181818180000"
```

```
,
             INIT_0F =>
X"000000003c1818183c6666666600660000000000000000000000000dc760000")
-- synopsis translate off
port map ( DO   => font_out2,
           ADDR => font_adr,
           CLK  => clock,
           DI   => font_in,
           EN   => '1',
           RST  => fontRAM_RST(2),
           WE   => '0');

instr_to_ei <= data_out_to_ei;
data_to_mem <= data_out_to_mem OR char_out_to_mem;
stop_all <= '0';
byte_to_video <= font_out0 or font_out1 or font_out2;

end Behavioral;
```

----------video_controller.vhd----------

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity video_controller is
port (
        Pixel_Clock     : in std_logic;
       Reset           : in std_logic;

        --minimips output : input video controller
       RT_Font_Byte    : in std_logic_vector(7 downto 0);
       RT_Shift_Load   : in std_logic;
       RT_Blank        : in std_logic;
       RT_Hsync        : in std_logic;
       RT_VSync        : in std_logic;

        --video connections
       VIDOUT_CLK     : out std_logic;
       VIDOUT_RED     : out std_logic_vector (9 downto 0);
       VIDOUT_GREEN   : out std_logic_vector (9 downto 0);
       VIDOUT_BLUE    : out std_logic_vector (9 downto 0);
       VIDOUT_BLANK_N : out std_logic;
       VIDOUT_HSYNC_N : out std_logic;
       VIDOUT_VSYNC_N : out std_logic);
end video_controller;

architecture Behavioral of video_controller is

    -- Signals for the video controller
  signal LoadNShift : std_logic;        -- Shift register control
  signal Font_Data  : std_logic_vector(7 downto 0);   -- Input from MM to shift register
  signal ShiftData  : std_logic_vector(7 downto 0);  -- Shift register data
  signal VideoData  : std_logic;         -- Serial out ANDed with blanking

  signal HS    : std_logic:='1';
  signal VS    : std_logic:='1';
  signal Blank : std_logic:='1';

begin

    VidBlank: process (Pixel_Clock, Reset)
    begin
        if Reset = '1' then
```

```
            Blank <= '1';
        elsif Pixel_Clock'event and Pixel_Clock = '1' then
            if RT_Blank = '1' then
                Blank <= not Blank;
            end if;
        end if;
    end process VidBlank;

VidHsync: process (Pixel_Clock, Reset)
  begin
  if Reset = '1' then
      HS <= '1';
    elsif Pixel_Clock'event and Pixel_Clock = '1' then
              if (RT_Hsync = '1') then
                    HS <= not HS;
    end if;
        end if;

  end process VidHsync;
VIDOUT_HSYNC_N <= HS;

VidVsync: process (Pixel_Clock, Reset)
  begin
  if Reset = '1' then
      VS <= '1';
  elsif Pixel_Clock'event and Pixel_Clock = '1' then
              if RT_Vsync = '1' then
                    VS <= not VS;
    end if;
 end if;
  end process VidVsync;
VIDOUT_VSYNC_N <= VS;


-- Shift register
LoadNShift <= RT_Shift_Load;
Font_Data  <= RT_Font_Byte;


ShiftRegister: process (Pixel_Clock, Reset)
  begin
    if Reset = '1' then
      ShiftData <= X"AB";
    elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if LoadNShift = '1' then
        ShiftData <= Font_Data;
      else
        ShiftData <= ShiftData(6 downto 0) & ShiftData(7);
      end if;
    end if;
  end process ShiftRegister;

  VideoData <= ShiftData(7);

  -- Registered video signals going to the video DAC

  VideoOut: process (Pixel_Clock, Reset)
  begin
    if Reset = '1' then
    --  VIDOUT_BLANK_N <= '0';
      VIDOUT_RED   <= "0000000000";
      VIDOUT_BLUE  <= "0000000000";
      VIDOUT_GREEN <= "0000000000";
```

```
    elsif Pixel_Clock'event and Pixel_Clock = '1' then
      if VideoData = '1' then
        VIDOUT_RED   <= "1111111111";
        VIDOUT_GREEN <= "1111111111";
        VIDOUT_BLUE  <= "1111111111";
      else
        VIDOUT_RED   <= "0000000000";
        VIDOUT_GREEN <= "0000000000";
        VIDOUT_BLUE  <= "0000000000";
      end if;
    end if;

  end process VideoOut;

VIDOUT_BLANK_N <= Blank;
VIDOUT_CLK <= Pixel_Clock;

end Behavioral;
```

----------clkgen.v----------

```
module clkgen(
FPGA_CLK1,

sys_clk,
pixel_clock,
fpga_reset

);
input FPGA_CLK1;
output sys_clk, pixel_clock, fpga_reset;

wire clk1x, clk05x;

wire clk_ibuf, clk1x_i, clk05x_i;
wire locked;

IBUFG clkibuf(.I(FPGA_CLK1), .O(clk_ibuf));
BUFG bg1 (.I(clk1x_i), .O(pixel_clock));
BUFG bg05 (.I(clk05x_i), .O(sys_clk));

// synopsys translate_off
defparam vdll.CLKDV_DIVIDE = 2.0 ;
// synopsys translate_on
// synthesis attribute CLKDV_DIVIDE of vdll is 2
CLKDLL vdll(.CLKIN(clk_ibuf), .CLKFB(pixel_clock), .CLK0(clk1x_i),
.CLKDV(clk05x_i), .RST(1'b0), .LOCKED(locked));

assign fpga_reset = ~locked;

endmodule
```

Test Bench

bus_ctrl_tbw.vhw

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
library UNISIM;
use UNISIM.VComponents.all;
library work;
use work.pack_mips.all;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;

ENTITY bus_tbw IS
END bus_tbw;

ARCHITECTURE testbench_arch OF bus_tbw IS
    COMPONENT bus_ctrl
        PORT (
            clock : In std_logic;
            reset : In std_logic;
            interrupt : In std_logic;
            adr_from_ei : In std_logic_vector (31 DownTo 0);
            instr_to_ei : Out std_logic_vector (31 DownTo 0);
            req_from_mem : In std_logic;
            r_w_from_mem : In std_logic;
            adr_from_mem : In std_logic_vector (31 DownTo 0);
            data_from_mem : In std_logic_vector (31 DownTo 0);
            data_to_mem : Out std_logic_vector (31 DownTo 0);
            load_to_video : Out std_logic;
            vsync_to_video : Out std_logic;
            hsync_to_video : Out std_logic;
            blank_to_video : Out std_logic;
            byte_to_video : Out std_logic_vector (7 DownTo 0);
            stop_all : In std_logic
        );
    END COMPONENT;

    SIGNAL clock : std_logic := '0';
    SIGNAL reset : std_logic := '1';
    SIGNAL interrupt : std_logic := '0';
    SIGNAL adr_from_ei : std_logic_vector (31 DownTo 0) :=
"00000000000000000000000000000000";
    SIGNAL instr_to_ei : std_logic_vector (31 DownTo 0) :=
"UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU";
    SIGNAL req_from_mem : std_logic := '0';
    SIGNAL r_w_from_mem : std_logic := '0';
    SIGNAL adr_from_mem : std_logic_vector (31 DownTo 0) :=
"00000000000000000000000000000000";
    SIGNAL data_from_mem : std_logic_vector (31 DownTo 0) :=
"00000000000000000000000000000000";
    SIGNAL data_to_mem : std_logic_vector (31 DownTo 0) :=
"UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU";
    SIGNAL load_to_video : std_logic := 'U';
    SIGNAL vsync_to_video : std_logic := 'U';
    SIGNAL hsync_to_video : std_logic := 'U';
    SIGNAL blank_to_video : std_logic := 'U';
    SIGNAL byte_to_video : std_logic_vector (7 DownTo 0) := "UUUUUUUU";
    SIGNAL stop_all : std_logic := '0';

    SHARED VARIABLE TX_ERROR : INTEGER := 0;
```

```
SHARED VARIABLE TX_OUT : LINE;

constant PERIOD : time := 200 ns;
constant DUTY_CYCLE : real := 0.5;
constant OFFSET : time := 0 ns;

BEGIN
    UUT : bus_ctrl
    PORT MAP (
        clock => clock,
        reset => reset,
        interrupt => interrupt,
        adr_from_ei => adr_from_ei,
        instr_to_ei => instr_to_ei,
        req_from_mem => req_from_mem,
        r_w_from_mem => r_w_from_mem,
        adr_from_mem => adr_from_mem,
        data_from_mem => data_from_mem,
        data_to_mem => data_to_mem,
        load_to_video => load_to_video,
        vsync_to_video => vsync_to_video,
        hsync_to_video => hsync_to_video,
        blank_to_video => blank_to_video,
        byte_to_video => byte_to_video,
        stop_all => stop_all
    );

    PROCESS    -- clock process for clock
    BEGIN
        WAIT for OFFSET;
        CLOCK_LOOP : LOOP
            clock <= '0';
            WAIT FOR (PERIOD - (PERIOD * DUTY_CYCLE));
            clock <= '1';
            WAIT FOR (PERIOD * DUTY_CYCLE);
        END LOOP CLOCK_LOOP;
    END PROCESS;

    PROCESS
        PROCEDURE CHECK_blank_to_video(
            next_blank_to_video : std_logic;
            TX_TIME : INTEGER
        ) IS
            VARIABLE TX_STR : String(1 to 4096);
            VARIABLE TX_LOC : LINE;
            BEGIN
            IF (blank_to_video /= next_blank_to_video) THEN
                STD.TEXTIO.write(TX_LOC, string'("Error at time="));
                STD.TEXTIO.write(TX_LOC, TX_TIME);
                STD.TEXTIO.write(TX_LOC, string'("ns blank_to_video="));
                IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, blank_to_video);
                STD.TEXTIO.write(TX_LOC, string'(", Expected = "));
                IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, next_blank_to_video);
                STD.TEXTIO.write(TX_LOC, string'(" "));
                TX_STR(TX_LOC.all'range) := TX_LOC.all;
                STD.TEXTIO.Deallocate(TX_LOC);
                ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
                TX_ERROR := TX_ERROR + 1;
            END IF;
        END;
        PROCEDURE CHECK_byte_to_video(
            next_byte_to_video : std_logic_vector (7 DownTo 0);
            TX_TIME : INTEGER
```

```
) IS
    VARIABLE TX_STR : String(1 to 4096);
    VARIABLE TX_LOC : LINE;
    BEGIN
    IF (byte_to_video /= next_byte_to_video) THEN
        STD.TEXTIO.write(TX_LOC, string'("Error at time="));
        STD.TEXTIO.write(TX_LOC, TX_TIME);
        STD.TEXTIO.write(TX_LOC, string'("ns byte_to_video="));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, byte_to_video);
        STD.TEXTIO.write(TX_LOC, string'(", Expected = "));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, next_byte_to_video);
        STD.TEXTIO.write(TX_LOC, string'(" "));
        TX_STR(TX_LOC.all'range) := TX_LOC.all;
        STD.TEXTIO.Deallocate(TX_LOC);
        ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
        TX_ERROR := TX_ERROR + 1;
    END IF;
END;
PROCEDURE CHECK_data_to_mem(
    next_data_to_mem : std_logic_vector (31 DownTo 0);
    TX_TIME : INTEGER
) IS
    VARIABLE TX_STR : String(1 to 4096);
    VARIABLE TX_LOC : LINE;
    BEGIN
    IF (data_to_mem /= next_data_to_mem) THEN
        STD.TEXTIO.write(TX_LOC, string'("Error at time="));
        STD.TEXTIO.write(TX_LOC, TX_TIME);
        STD.TEXTIO.write(TX_LOC, string'("ns data_to_mem="));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, data_to_mem);
        STD.TEXTIO.write(TX_LOC, string'(", Expected = "));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, next_data_to_mem);
        STD.TEXTIO.write(TX_LOC, string'(" "));
        TX_STR(TX_LOC.all'range) := TX_LOC.all;
        STD.TEXTIO.Deallocate(TX_LOC);
        ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
        TX_ERROR := TX_ERROR + 1;
    END IF;
END;
PROCEDURE CHECK_hsync_to_video(
    next_hsync_to_video : std_logic;
    TX_TIME : INTEGER
) IS
    VARIABLE TX_STR : String(1 to 4096);
    VARIABLE TX_LOC : LINE;
    BEGIN
    IF (hsync_to_video /= next_hsync_to_video) THEN
        STD.TEXTIO.write(TX_LOC, string'("Error at time="));
        STD.TEXTIO.write(TX_LOC, TX_TIME);
        STD.TEXTIO.write(TX_LOC, string'("ns hsync_to_video="));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, hsync_to_video);
        STD.TEXTIO.write(TX_LOC, string'(", Expected = "));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, next_hsync_to_video);
        STD.TEXTIO.write(TX_LOC, string'(" "));
        TX_STR(TX_LOC.all'range) := TX_LOC.all;
        STD.TEXTIO.Deallocate(TX_LOC);
        ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
        TX_ERROR := TX_ERROR + 1;
    END IF;
END;
PROCEDURE CHECK_instr_to_ei(
    next_instr_to_ei : std_logic_vector (31 DownTo 0);
    TX_TIME : INTEGER
```

```
) IS
    VARIABLE TX_STR : String(1 to 4096);
    VARIABLE TX_LOC : LINE;
    BEGIN
    IF (instr_to_ei /= next_instr_to_ei) THEN
        STD.TEXTIO.write(TX_LOC, string'("Error at time="));
        STD.TEXTIO.write(TX_LOC, TX_TIME);
        STD.TEXTIO.write(TX_LOC, string'("ns instr_to_ei="));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, instr_to_ei);
        STD.TEXTIO.write(TX_LOC, string'(", Expected = "));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, next_instr_to_ei);
        STD.TEXTIO.write(TX_LOC, string'(" "));
        TX_STR(TX_LOC.all'range) := TX_LOC.all;
        STD.TEXTIO.Deallocate(TX_LOC);
        ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
        TX_ERROR := TX_ERROR + 1;
    END IF;
END;
PROCEDURE CHECK_load_to_video(
    next_load_to_video : std_logic;
    TX_TIME : INTEGER
) IS
    VARIABLE TX_STR : String(1 to 4096);
    VARIABLE TX_LOC : LINE;
    BEGIN
    IF (load_to_video /= next_load_to_video) THEN
        STD.TEXTIO.write(TX_LOC, string'("Error at time="));
        STD.TEXTIO.write(TX_LOC, TX_TIME);
        STD.TEXTIO.write(TX_LOC, string'("ns load_to_video="));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, load_to_video);
        STD.TEXTIO.write(TX_LOC, string'(", Expected = "));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, next_load_to_video);
        STD.TEXTIO.write(TX_LOC, string'(" "));
        TX_STR(TX_LOC.all'range) := TX_LOC.all;
        STD.TEXTIO.Deallocate(TX_LOC);
        ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
        TX_ERROR := TX_ERROR + 1;
    END IF;
END;
PROCEDURE CHECK_vsync_to_video(
    next_vsync_to_video : std_logic;
    TX_TIME : INTEGER
) IS
    VARIABLE TX_STR : String(1 to 4096);
    VARIABLE TX_LOC : LINE;
    BEGIN
    IF (vsync_to_video /= next_vsync_to_video) THEN
        STD.TEXTIO.write(TX_LOC, string'("Error at time="));
        STD.TEXTIO.write(TX_LOC, TX_TIME);
        STD.TEXTIO.write(TX_LOC, string'("ns vsync_to_video="));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, vsync_to_video);
        STD.TEXTIO.write(TX_LOC, string'(", Expected = "));
        IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, next_vsync_to_video);
        STD.TEXTIO.write(TX_LOC, string'(" "));
        TX_STR(TX_LOC.all'range) := TX_LOC.all;
        STD.TEXTIO.Deallocate(TX_LOC);
        ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
        TX_ERROR := TX_ERROR + 1;
    END IF;
END;
BEGIN
    -- -------------  Current Time:  85ns
    WAIT FOR 85 ns;
```

```
            reset <= '0';
            req_from_mem <= '1';
            -- -------------------------------------
            -- -------------  Current Time:  115ns
            WAIT FOR 30 ns;
            CHECK_load_to_video('0', 115);
            CHECK_vsync_to_video('0', 115);
            CHECK_hsync_to_video('0', 115);
            CHECK_blank_to_video('0', 115);
            CHECK_instr_to_ei("00100000000101000000000000011110", 115);
            CHECK_data_to_mem("00100000000101000000000000011110", 115);
            CHECK_byte_to_video("00000000", 115);
            -- -------------------------------------
            -- -------------  Current Time:  485ns
            WAIT FOR 370 ns;
            stop_all <= '1';
            -- -------------------------------------
            -- -------------  Current Time:  1085ns
            WAIT FOR 600 ns;
            adr_from_mem <= "00000000000000000100000000000010";
            -- -------------------------------------
            -- -------------  Current Time:  1115ns
            WAIT FOR 30 ns;
            CHECK_data_to_mem("00000000000000000000000000000000", 1115);
            -- -------------------------------------
            -- -------------  Current Time:  3485ns
            WAIT FOR 2370 ns;
            req_from_mem <= '0';
            stop_all <= '0';
            -- -------------------------------------
            -- -------------  Current Time:  3515ns
            WAIT FOR 30 ns;
            CHECK_hsync_to_video('1', 3515);
            -- -------------------------------------
            WAIT FOR 1685 ns;

        IF (TX_ERROR = 0) THEN
            STD.TEXTIO.write(TX_OUT, string'("No errors or warnings"));
            ASSERT (FALSE) REPORT
              "Simulation successful (not a failure).  No problems detected."
              SEVERITY FAILURE;
        ELSE
            STD.TEXTIO.write(TX_OUT, TX_ERROR);
            STD.TEXTIO.write(TX_OUT,
                string'(" errors found in simulation"));
            ASSERT (FALSE) REPORT "Errors found during simulation"
                 SEVERITY FAILURE;
        END IF;
    END PROCESS;

END testbench_arch;
```

banc_tbw.vhw

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
library work;
use work.pack_mips.all;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;
```

```
ENTITY tbw IS
END tbw;

ARCHITECTURE testbench_arch OF tbw IS
    COMPONENT banc
        PORT (
            clock : In std_logic;
            reset : In std_logic;
            reg_src1 : In std_logic_vector (4 DownTo 0);
            reg_src2 : In std_logic_vector (4 DownTo 0);
            reg_dest : In std_logic_vector (4 DownTo 0);
            donnee : In std_logic_vector (31 DownTo 0);
            cmd_ecr : In std_logic;
            data_src1 : Out std_logic_vector (31 DownTo 0);
            data_src2 : Out std_logic_vector (31 DownTo 0);
            stop_all : Out std_logic
        );
    END COMPONENT;

    SIGNAL clock : std_logic := '0';
    SIGNAL reset : std_logic := '1';
    SIGNAL reg_src1 : std_logic_vector (4 DownTo 0) := "00000";
    SIGNAL reg_src2 : std_logic_vector (4 DownTo 0) := "00000";
    SIGNAL reg_dest : std_logic_vector (4 DownTo 0) := "00000";
    SIGNAL donnee : std_logic_vector (31 DownTo 0) :=
"00000000000000000000000000000000";
    SIGNAL cmd_ecr : std_logic := '0';
    SIGNAL data_src1 : std_logic_vector (31 DownTo 0) :=
"00000000000000000000000000000000";
    SIGNAL data_src2 : std_logic_vector (31 DownTo 0) :=
"00000000000000000000000000000000";
    SIGNAL stop_all : std_logic := '';

    SHARED VARIABLE TX_ERROR : INTEGER := 0;
    SHARED VARIABLE TX_OUT : LINE;

    constant PERIOD : time := 100 ns;
    constant DUTY_CYCLE : real := 0.5;
    constant OFFSET : time := 0 ns;

    BEGIN
        UUT : banc
        PORT MAP (
            clock => clock,
            reset => reset,
            reg_src1 => reg_src1,
            reg_src2 => reg_src2,
            reg_dest => reg_dest,
            donnee => donnee,
            cmd_ecr => cmd_ecr,
            data_src1 => data_src1,
            data_src2 => data_src2,
            stop_all => stop_all
        );

        PROCESS    -- clock process for clock
        BEGIN
            WAIT for OFFSET;
            CLOCK_LOOP : LOOP
                clock <= '0';
                WAIT FOR (PERIOD - (PERIOD * DUTY_CYCLE));
                clock <= '1';
```

```
        WAIT FOR (PERIOD * DUTY_CYCLE);
    END LOOP CLOCK_LOOP;
END PROCESS;

PROCESS
    PROCEDURE CHECK_data_src1(
        next_data_src1 : std_logic_vector (31 DownTo 0);
        TX_TIME : INTEGER
    ) IS
        VARIABLE TX_STR : String(1 to 4096);
        VARIABLE TX_LOC : LINE;
        BEGIN
        IF (data_src1 /= next_data_src1) THEN
            STD.TEXTIO.write(TX_LOC, string'("Error at time="));
            STD.TEXTIO.write(TX_LOC, TX_TIME);
            STD.TEXTIO.write(TX_LOC, string'("ns data_src1="));
            IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, data_src1);
            STD.TEXTIO.write(TX_LOC, string'(", Expected = "));
            IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, next_data_src1);
            STD.TEXTIO.write(TX_LOC, string'(" "));
            TX_STR(TX_LOC.all'range) := TX_LOC.all;
            STD.TEXTIO.Deallocate(TX_LOC);
            ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
            TX_ERROR := TX_ERROR + 1;
        END IF;
    END;
    PROCEDURE CHECK_data_src2(
        next_data_src2 : std_logic_vector (31 DownTo 0);
        TX_TIME : INTEGER
    ) IS
        VARIABLE TX_STR : String(1 to 4096);
        VARIABLE TX_LOC : LINE;
        BEGIN
        IF (data_src2 /= next_data_src2) THEN
            STD.TEXTIO.write(TX_LOC, string'("Error at time="));
            STD.TEXTIO.write(TX_LOC, TX_TIME);
            STD.TEXTIO.write(TX_LOC, string'("ns data_src2="));
            IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, data_src2);
            STD.TEXTIO.write(TX_LOC, string'(", Expected = "));
            IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, next_data_src2);
            STD.TEXTIO.write(TX_LOC, string'(" "));
            TX_STR(TX_LOC.all'range) := TX_LOC.all;
            STD.TEXTIO.Deallocate(TX_LOC);
            ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
            TX_ERROR := TX_ERROR + 1;
        END IF;
    END;
    PROCEDURE CHECK_stop_all(
        next_stop_all : std_logic;
        TX_TIME : INTEGER
    ) IS
        VARIABLE TX_STR : String(1 to 4096);
        VARIABLE TX_LOC : LINE;
        BEGIN
        IF (stop_all /= next_stop_all) THEN
            STD.TEXTIO.write(TX_LOC, string'("Error at time="));
            STD.TEXTIO.write(TX_LOC, TX_TIME);
            STD.TEXTIO.write(TX_LOC, string'("ns stop_all="));
            IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, stop_all);
            STD.TEXTIO.write(TX_LOC, string'(", Expected = "));
            IEEE.STD_LOGIC_TEXTIO.write(TX_LOC, next_stop_all);
            STD.TEXTIO.write(TX_LOC, string'(" "));
            TX_STR(TX_LOC.all'range) := TX_LOC.all;
```

```
        STD.TEXTIO.Deallocate(TX_LOC);
        ASSERT (FALSE) REPORT TX_STR SEVERITY ERROR;
        TX_ERROR := TX_ERROR + 1;
    END IF;
END;
BEGIN
    -- -------------  Current Time:  49ns
    WAIT FOR 49 ns;
    reg_dest <= "11111";
    -- -------------------------------------
    -- -------------  Current Time:  149ns
    WAIT FOR 100 ns;
    reset <= '0';
    reg_src2 <= "00010";
    -- -------------------------------------
    -- -------------  Current Time:  249ns
    WAIT FOR 100 ns;
    cmd_ecr <= '1';
    donnee <= "00000000000000000000000000001000";
    -- -------------------------------------
    -- -------------  Current Time:  349ns
    WAIT FOR 100 ns;
    cmd_ecr <= '0';
    reg_src1 <= "11111";
    -- -------------------------------------
    -- -------------  Current Time:  360ns
    WAIT FOR 11 ns;
    CHECK_data_src1("00000000000000000000000000000111", 360);
    -- -------------------------------------
    -- -------------  Current Time:  460ns
    WAIT FOR 100 ns;
    CHECK_data_src1("00000000000000000000000000000110", 460);
    -- -------------------------------------
    -- -------------  Current Time:  560ns
    WAIT FOR 100 ns;
    CHECK_data_src1("00000000000000000000000000000101", 560);
    -- -------------------------------------
    -- -------------  Current Time:  660ns
    WAIT FOR 100 ns;
    CHECK_data_src1("00000000000000000000000000000100", 660);
    -- -------------------------------------
    -- -------------  Current Time:  760ns
    WAIT FOR 100 ns;
    CHECK_data_src1("00000000000000000000000000000011", 760);
    -- -------------------------------------
    -- -------------  Current Time:  849ns
    WAIT FOR 89 ns;
    cmd_ecr <= '1';
    -- -------------------------------------
    -- -------------  Current Time:  860ns
    WAIT FOR 11 ns;
    CHECK_data_src1("00000000000000000000000000000010", 860);
    -- -------------------------------------
    -- -------------  Current Time:  960ns
    WAIT FOR 100 ns;
    CHECK_data_src1("00000000000000000000000000000001", 960);
    -- -------------------------------------
    -- -------------  Current Time:  1049ns
    WAIT FOR 89 ns;
    donnee <= "00000000000000000000000000000100";
    -- -------------------------------------
    -- -------------  Current Time:  1060ns
    WAIT FOR 11 ns;
```

```
CHECK_data_src1("00000000000000000000000000000000", 1060);
-- -------------------------------------
-- -------------  Current Time:  1149ns
WAIT FOR 89 ns;
cmd_ecr <= '1';
-- -------------------------------------
-- -------------  Current Time:  1160ns
WAIT FOR 11 ns;
CHECK_data_src1("00000000000000000000000000001000", 1160);
-- -------------------------------------
-- -------------  Current Time:  1260ns
WAIT FOR 100 ns;
CHECK_data_src1("00000000000000000000000000000111", 1260);
-- -------------------------------------
-- -------------  Current Time:  1349ns
WAIT FOR 89 ns;
cmd_ecr <= '0';
-- -------------------------------------
-- -------------  Current Time:  1360ns
WAIT FOR 11 ns;
CHECK_data_src1("00000000000000000000000000000110", 1360);
-- -------------------------------------
-- -------------  Current Time:  1460ns
WAIT FOR 100 ns;
CHECK_data_src1("00000000000000000000000000000101", 1460);
-- -------------------------------------
-- -------------  Current Time:  1560ns
WAIT FOR 100 ns;
CHECK_data_src1("00000000000000000000000000000100", 1560);
-- -------------------------------------
-- -------------  Current Time:  1660ns
WAIT FOR 100 ns;
CHECK_data_src1("00000000000000000000000000000011", 1660);
-- -------------------------------------
-- -------------  Current Time:  1760ns
WAIT FOR 100 ns;
CHECK_data_src1("00000000000000000000000000000010", 1760);
-- -------------------------------------
-- -------------  Current Time:  1860ns
WAIT FOR 100 ns;
CHECK_data_src1("00000000000000000000000000000001", 1860);
-- -------------------------------------
-- -------------  Current Time:  1960ns
WAIT FOR 100 ns;
CHECK_data_src1("00000000000000000000000000000000", 1960);
-- -------------------------------------
-- -------------  Current Time:  2249ns
WAIT FOR 289 ns;
donnee <= "00000000000000000000000000000101";
-- -------------------------------------
-- -------------  Current Time:  3349ns
WAIT FOR 1100 ns;
cmd_ecr <= '1';
donnee <= "00000000000000000000000000001000";
-- -------------------------------------
-- -------------  Current Time:  3360ns
WAIT FOR 11 ns;
CHECK_data_src1("00000000000000000000000000001000", 3360);
-- -------------------------------------
-- -------------  Current Time:  3449ns
WAIT FOR 89 ns;
cmd_ecr <= '0';
-- -------------------------------------
```

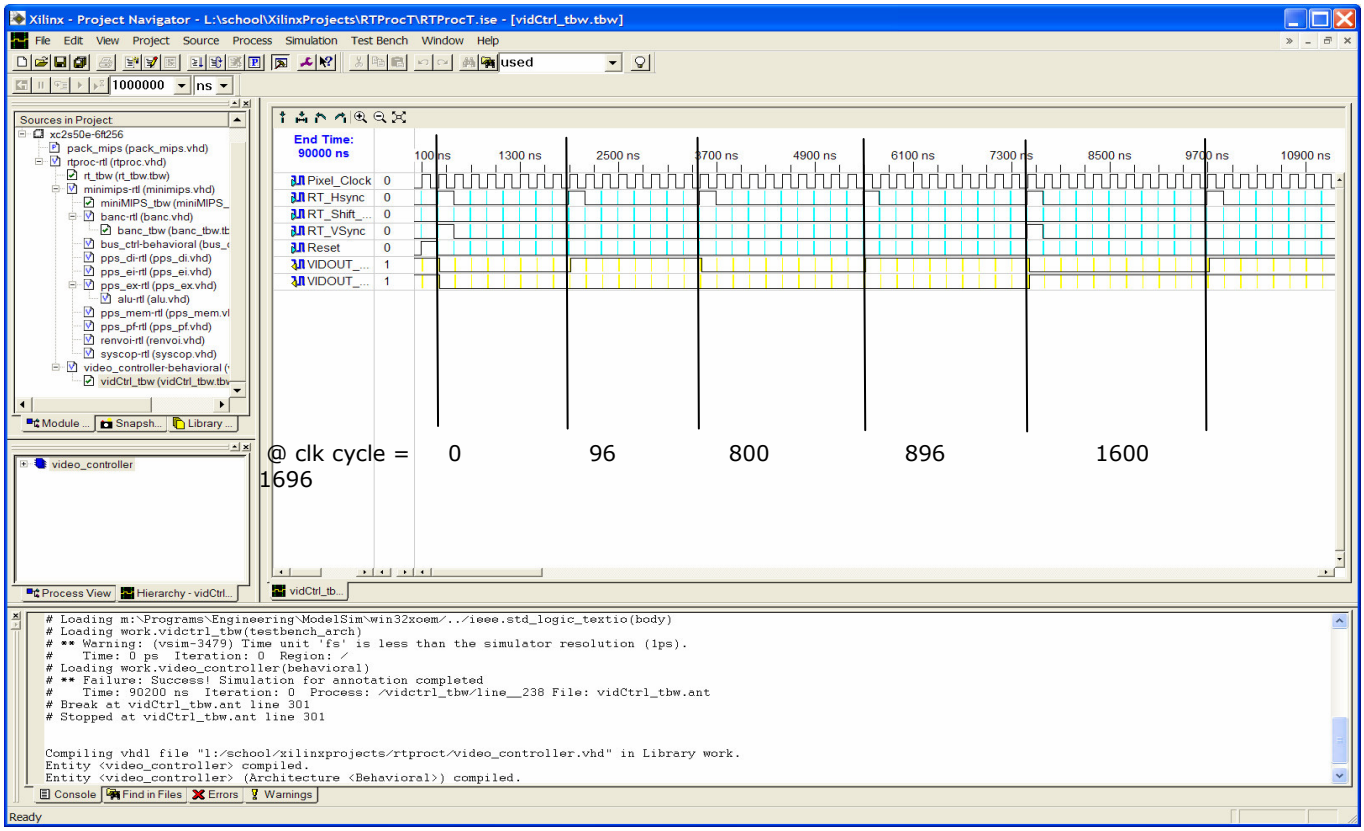
```
            -- -------------  Current Time:  3460ns
            WAIT FOR 11 ns;
            CHECK_data_src1("00000000000000000000000000000111", 3460);
            -- -------------------------------------
            -- -------------  Current Time:  3560ns
            WAIT FOR 100 ns;
            CHECK_data_src1("00000000000000000000000000000110", 3560);
            -- -------------------------------------
            -- -------------  Current Time:  3660ns
            WAIT FOR 100 ns;
            CHECK_data_src1("00000000000000000000000000000101", 3660);
            -- -------------------------------------
            -- -------------  Current Time:  3760ns
            WAIT FOR 100 ns;
            CHECK_data_src1("00000000000000000000000000000100", 3760);
            -- -------------------------------------
            -- -------------  Current Time:  3860ns
            WAIT FOR 100 ns;
            CHECK_data_src1("00000000000000000000000000000011", 3860);
            -- -------------------------------------
            -- -------------  Current Time:  3960ns
            WAIT FOR 100 ns;
            CHECK_data_src1("00000000000000000000000000000010", 3960);
            -- -------------------------------------
            -- -------------  Current Time:  4060ns
            WAIT FOR 100 ns;
            CHECK_data_src1("00000000000000000000000000000001", 4060);
            -- -------------------------------------
            -- -------------  Current Time:  4160ns
            WAIT FOR 100 ns;
            CHECK_data_src1("00000000000000000000000000000000", 4160);
            -- -------------------------------------
            WAIT FOR 940 ns;

            IF (TX_ERROR = 0) THEN
                STD.TEXTIO.write(TX_OUT, string'("No errors or warnings"));
                ASSERT (FALSE) REPORT
            "Simulation successful (not a failure).  No problems detected."
                  SEVERITY FAILURE;
            ELSE
                STD.TEXTIO.write(TX_OUT, TX_ERROR);
                STD.TEXTIO.write(TX_OUT,
                    string'(" errors found in simulation"));
                ASSERT (FALSE) REPORT "Errors found during simulation"
                     SEVERITY FAILURE;
            END IF;
        END PROCESS;

END testbench_arch;
```

video_ctrl_tbw.vhw

Not included because it is too long. See attached compressed directory testbench

Video Controller Simulations for correct output behavior

VSYNC: Fist two lines



Start and end of horizontal timing line