Processor Documentation

For the final project[1] (Tetris!), you have been provided a functioning single-cycle processor (yay!). We have provided this processor so you can immediately begin work on the project without having to worry about your processor's functionality. This opportunity comes with a catch: this processor is written in VHDL and implements a slightly different ISA from the processor implemented in project checkpoint 3.

VHDL vs. Verilog

As mentioned above, the given processor is not written in Verilog as yours was: it is written in VHDL. VHDL is a hardware description language like Verilog, however, its syntax is based on the programming language Pascal rather than C (which Verilog is based on). Don't worry: you will only need to modify this code if you intend on implementing interrupts, and even then, little modification is needed. You will certainly need to understand it enough to be able to use the processor, though. Here are a few useful tutorials about Verilog and VHDL:

http://www.referencedesigner.com/tutorials/verilog/verilog_01.php

https://www.seas.upenn.edu/~ese171/vhdl/vhdl_primer.html

VHDL modules, just like Verilog modules, can be instantiated and used in other modules – even Verilog modules. This means a VHDL module (i.e. skeleton, processor) can be used in your Verilog files in the exact same way any of your Verilog modules are.

Overall Structural Differences

The provided skeleton includes the top-level entity ("**skeleton**"), the processor itself ("**processor**"), drivers for the LCD ("**lcd**"), PS/2 keyboard ("**ps2**"), and several supporting modules (see section 2 for details).

The control unit ("**control**") is responsible for interpreting instruction opcodes and setting certain signals correctly so that the correct instruction can execute. It implements a very small RISC ISA detailed below which utilizes very similar opcodes and instructions to your processor. Note that there are instructions called input and output, which will be explained below. The processor module has following inputs and outputs:

```
clock        : IN STD_LOGIC;
reset        : IN STD_LOGIC;
keyboard_in  : IN STD_LOGIC_VECTOR(31 downto 0);
keyboard_ack : OUT[1] STD_LOGIC;
lcd_write    : OUT STD_LOGIC;
lcd_data     : OUT STD_LOGIC_VECTOR(31 downto 0);
```

[1] This documentation was adopted from a processor assignment brief by Dr. Bletsch.

The input **clock** is the 50MHz clock input to the skeleton. The internals of this processor handle slowing down the clock and clocking individual components correctly. Before **reset** is asserted high, the state of the processor is undefined. After **reset** is asserted, the processor begins execution from instruction memory address zero with all zeros in the register file.

Provided Materials

The structure of the provided skeleton project is as follows:

- **skeleton**: The top-level entity for the finished system; includes the processor as well as the keyboard and LCD.
- **lcd**, **ps2**: Interfaces to the LCD display and PS/2 keyboard.
- **processor**: The processor itself.
- **control**: A module to translate the opcode to various control signals needed by the CPU datapath.
- **pll**: The phase-lock-loop that provides the clock for your system. It's provided as an Altera IP module, and it's set to 10MHz by default.
- **imem:** Instruction memory ROM. Implemented as an Altera IP module. To change the program, provide this component with an appropriate memory (MIF or HEX) file.
- **dmem:** Data memory RAM. Implemented as an Altera IP module. To change the program, provide this component with an appropriate memory (MIF or HEX) file.
- **alu**: An Arithmetic/Logic Unit (ALU) with support for the required math operations.
- **adder_rc**: A basic reusable n-bit ripple-carry adder module.
- **adder_cs**: A basic reusable n-bit carry-select adder module.
- **shifter**: A basic 32-bit bit shifter.
- **regfile**: A register file with 32 32-bit registers, one write port, and two read ports.
- **reg**: A basic reusable n-bit register module.
- **reg_2port**: Similar to the basic **reg** module, but with two tri-state buffered output ports.
- **reg0_2port**: Similar to the **reg_2port** module, but with a permanent value of 0.
- **decoder5to32**: A basic 5-to-32-bit decoder.
- **mux**: A basic reusable n-bit 2-to-1 mux module.

ISA Differences

Two big differences between the ISA implemented in your processor and this processor are the **input** and **output** instructions (explained below). **blt** and **bne** have also been replaced with **bgt** and **beq** in this instruction set. Finally, **sra** has been replaced with **srl** in this instruction set. All of these differences can be seen in the detailed ISA description below. The only other major difference is the slightly different opcode setup. Please carefully analyze the description below to understand how to use this processor!

input instruction

This instruction is used to read in a value from the PS2 keyboard. On the same cycle that the data provided through **keyboard_in** is read in, **keyboard_ack** is asserted high for that clock cycle and only that clock cycle.

output instruction

This instruction is used to print a value to the LCD display. On the same cycle that data is written to **lcd_data**, **lcd_write** is asserted high for that clock cycle and only that clock cycle.

The Instruction Set

This CPU has 32 general purpose registers: $r0-$r31. Register $r0 is the constant value 0, as in your processor. The register $r31 is the link register for the jal instruction.

instruction	opcode	type	usage	operation
add	00000	R	add $rd, $rs, $rt	$rd = $rs + $rt
sub	00001	R	sub $rd, $rs, $rt	$rd = $rs – $rt
and	00010	R	and $rd, $rs, $rt	$rd = $rs AND $rt
or	00011	R	or $rd, $rs, $rt	$rd = $rs OR $rt
sll	00100	R	sll $rd, $rs, $rt	$rd = $rs shifted left by $rt[4:0], zero-fill
srl	00101	R	srl $rd, $rs, $rt	$rd = $rs shifted right by $rt[4:0], zero-extend
addi	00110	I	addi $rd, $rs, N	$rd = $rs + N
lw	00111	I	lw $rd, N($rs)	$rd = Mem[$rs+N]
sw	01000	I	sw $rd, N($rs)	Mem[$rs+N] = $rd
beq	01001	I	beq $rd, $rs, N	if ($rd==$rs) then PC=PC+1+N
bgt	01010	I	bgt $rd, $rs, N	if ($rd>$rs) then PC=PC+1+N
jr	01011	I	jr $rd	PC = $rd
j	01100	J	j N	PC = N
jal	01101	J	jal N	$r31=PC+1; PC = N
input	01110	I	input $rd	$rd = keyboard input
output	01111	I	output $rd	print character $rd[7:0] on LCD display

The formats of the R, I, and J type instructions are shown below.

Type	Format				
R	Opcode [31:27]	Rd [26:22]	Rs [21:17]	Rt [16:12]	Zeroes [11:0]
I	Opcode [31:27]	Rd [26:22]	Rs [21:17]	Immediate [16:0]	
J	Opcode [31:27]	Target [26:0]			

Notes:

- The immediate field in I-Type instructions (bits 16 downto 0) is signed 2s complement.
- Register fields that are undefined are filled with zeroes by the assembler (for example, the jr instruction will have an $rt field which isn't used; the assembler will set this field to zero). That said, this shouldn't matter, as such instructions shouldn't be doing anything with such registers anyway.
- Register $r0 always equals zero.
- Registers $r1 through $r30 are general purpose.
- Register $r31 stores the link address of a jump-and-link instruction.
- The **`input`** instruction shall assert high on **`input_ack`** for the cycle only when the input is read from the keyboard controller; otherwise it shall assert low.
- The **`output`** instruction shall assert high on **`LCD_wren`** for the cycle only when the data is output to the LCD controller; otherwise it shall assert low.
- **Memory is word-addressed**, meaning that each unique memory address gives a full 32-bit word; this is in contrast to memory on MIPS and x86, which are byte-addressed. This was done because word-addressed memory is actually easier to implement.
- Note that the operand order for **`bgt`** and **`beq`**, as applied to the ALU, is different from most other instructions. Further, note that the ALU gives you an "**`isLessThan`**" signal rather than "**`isGreaterThan`**".

Note: Testing the Processor

It is a good idea to test the functioning the processor before modifying or adding to it. Even though the processor is complete, ensure you are able to use it before you delve in. When running on the FPGA board, a keyboard should be plugged into the PS/2 port (not USB port) on the board for input.

We are providing a number of **test programs** for you to verify your understanding of the processor:

- **`test-fibonacci`**: Asks a user for a number N, then prints the N-th Fibonacci number. It computes recursively, so may have memory issues for N>30 or so.
- **`test-give_me_n`**: Asks a user for a number, then prints that number.
- **`test-hello`**: This program prints "Hello" just with immediate values.
- **`test-hello2`**: This program prints "Hello from dmem" from data memory.
- **`test-simple`**: This program doesn't output anything; it mainly just plays with various instructions, and is suitable for simulation.

The HEX files for instruction memory and data memory are provided for the first two[2]. You are expected to write its own assembler tool for this project.

[2] Quoting Dr. Bletsch: “These test programs were just available to me as hex, with the original author and means of creation lost to history. It has been disassembled using the enclosed disassembler, with jumps annotated as comments. This is an interestingly realistic situation, as having an undocumented binary file and needing to reverse-engineer it is not an uncommon problem in industry.”