# Homework #4 – Processor Core Design


Version 1.0 Last updated 2017-10-18

For this homework, you will be building a single-cycle processor in Quartus II using Structural VHDL, downloading the design onto FPGA board, and running the test programs provided for you. You may use when-else, but not other behavioral style VHDL. This problem is not trivial, so please start early. Please submit a Quartus Archive (**`.qar`**) file for this assignment.

*Have fun!!*

## 1. Problem Statement

In this problem, you will build an un-pipelined single-cycle processor similar to the one we discussed extensively in class. A skeleton has been built for you, including many of the essential components that make up the CPU. This 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 for you to implement, which is responsible for interpreting instruction opcodes and setting certain signals correctly so that the correct instruction can execute. You will incorporate these components and add other combinational logic to make the processor functional. The ISA you should implement is listed in part 3. It is a very small and simplified RISC ISA. (It is similar to but differs in a number of ways from the "real" MIPS.) Note that there are instructions called input and output, which will be explained below. The processor module has following inputs and outputs: (do not modify these interfaces)

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

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.

Your implementation of the **`input`** instructions should have the following semantics: on the same cycle that you read the data provided on **`keyboard_in`**, you must assert **`keyboard_ack`** high for that clock cycle and only that clock cycle. Failing to assert it high will make you always read the same input; asserting it high for more than one cycle will destroy unread data in the keyboard buffer.

Your implementation of the **output** instruction should have the following semantics: on the same cycle that you write data to **lcd_data**, you must assert **lcd_write** high for that clock cycle and only that clock cycle. Failing to assert it high will cause no output to be displayed; asserting it high for more than one cycle will display additional gibberish.

## 2. Provided materials

The structure of the provided skeleton project is as follows; you'll just be implementing two of the existing components (plus any components you choose to add):

- **skeleton**: The top-level entity for the finished system; includes your processor as well as the keyboard and LCD.
- **lcd**, **ps2**: Interfaces to the LCD display and PS/2 keyboard.
- **processor**: The processor itself. ***You'll be implementing this component.***
- **control**: An empty module intended to translate the opcode to various control signals needed by the CPU datapath. ***You'll be implementing this component.***
- **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 a the appropriate ***-imem.hex** file.
- **dmem:** Data memory RAM. Implemented as an Altera IP module. To change the program, provide this component with a the appropriate ***-dmem.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.

## 3. The instruction set

Your CPU has 32 general purpose registers: $r0-$r31. Register $r0 is the constant value 0 (i.e., an instruction can specify it as a destination but "writing" to $r0 must not change its value). The register $r31 is the link register for the jal instruction (similar to $ra in MIPS). The user of your CPU may write to it with other instructions, but that would mess up function call/return for them.

| 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:

- Unlike the MIPS, the opcode is only 5 bits, and there is no "func" field for R-type instructions.
- The immediate field in I-Type instructions (bits 16 downto 0) is signed 2s complement. The processor should sign-extend it to the full 32-bit word size. *(note this is 17 bits, strangely.)*
- The instruction memory ROM is fairly small, so your PC can be as small as 12 bits; bits higher than this in J-type instructions (bits 26 downto 0) can be discarded.
- 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.
- The instruction and data memory address spaces are separate (one is a ROM, the other a RAM).
- Static data begins at data memory address zero. Stack data begins at the end of the data memory and grows downwards. There is no preset boundary between the end of static data and the start of the upwards-growing heap; this is a responsibility of the assembly program.
- 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`**".
- There are many aspects of the ISA that don't actually affect your job as the CPU architect at all. As a result, you don't need to worry about:
    - Stack management – the stack is a convention maintained by programmers writing code for your CPU; you don't have to do anything to make it exist. This means that it's up the programmer to decide if one the registers will be used as a stack pointer; you as the CPU designer don't have to do anything special to allow or enforce this.
    - Heap management – same as the stack; it's maintained by the programmers so you don't have to do anything to make it exist. This means that even though the heap is supposed to start right above static data, you as the CPU designer don't have to do anything special to allow or enforce this.
    - The kernel – there's no OS kernel for your CPU, and user programs running on your CPU will have direct access to the I/O devices (keyboard+LCD), so you don't need to worry about inventing syscalls, protected instructions, exceptions, etc.

## 4. Testing

It is a good idea to test thoroughly before you download the design to FPGA board. It is recommended to use ModelSim to test each module. 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 an **assembler** and a **disassembler** for you to generate test programs and to analyze existing binary files. These tools are posted on the course page (below the link to this writeup). These are very limited tools (e.g., no hex values for constants - only decimal integers). We have tested the assembler on the Duke Linux machines. You will have to copy the generated memory image files to your own machine, or you can use the assembler to whatever machine you have, provided you have Python 2.6 or later installed. *Please read the included* ***`readme.txt`*** *for details on these tools.*

Along with these tools, we are providing a number of **test programs**:

- **`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[1]. For the others, you'll need to use the included assembler to produce these HEX files. The output format is the Intel HEX format, which specifies words using a special notation.

To use a given program, update the **`imem`** and **`dmem`** components using the wizard interface to set their initialization file to the appropriate **`-imem.hex`** and **`-dmem.hex`** files.

The **`test-give_me_n-*.hex`** file is selected by default for **`imem`** and **`dmem`**. When this program starts, it displays "Give Me N:", and you can type in a number and press "Enter", and it should show "n=" and the number that you entered. If you then press "reset" the process repeats (you are prompted for another number, which it will then display back to you).

[1] 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...

## 5. Extra Credit

There are several ways to earn extra credit on this assignment, each worth up to 5 points. You may implement as many as you wish, but you will only be credited for **up to two (2)** of them. Options:

- Implement a game or other interesting program frøm scratch on your CPU. The program must be of significant complexity.
- Display PC[15:0] in hex on the seven-segment display using digits HEX3..HEX0.
- Display a "mega-instructions executed" counter in hex on the seven-segment display using digits HEX7..HEX4; this will show a count of instructions executed divided by $2^{20}$ (which approximates $10^6$). This counter should be cleared to zero on reset.
- Allow switching between the PLL-provided clock and a manual "human" clock. When SW0 is set high, the CPU clock should run at full speed. When SW0 is set low, the clock should be tied to KEY3, meaning that each press of the button will advance one clock cycle. To allow this to be observed, tie LEDG0 to the clock and make LEDR17..LEDR0 show the high 18 bits of the current instruction. (Note: because of the physics involved in the actual button, a single press may be registered as more than one cycle; this is okay for our purposes, but in industry, the switch would be processed with "debounce" logic to eliminate these oscillations.)

These are the kinds of things people were doing with the earliest microcomputers, such as the Altair 8800 from 1974, which had a physical "stop/run" switch, a "single step" button, and an array of switches and LEDs to directly view/set the address and data lines of the CPU.

**NOTE: Do not let your pursuit of extra credit endanger the proper functioning of the main CPU!**