# Homework #4 – Processor Core Design

# Due Wednesday, March 29, at 5:00pm

# 150 points

This homework requires you to design and implement the Duke 250/16, a 16-bit MIPS-like, **word-addressed (not byte-addressed)** RISC architecture. (A word is 16-bits.) We have specified the architecture, and you will use Logisim to design a single cycle implementation of this architecture. The architecture's instructions are specified in Table 1.

Submission instructions – please read VERY carefully:

- You must do all work individually, and you must submit your work electronically via Sakai.
- You will submit a Logisim file called hw4.circ. This file is the circuit for your processor.
- You will submit a PDF file called hw4.pdf. This file is your description of your processor, and the grader will use this description to help assign partial credit. (This file is for your benefit!) The file should explain the following issues:
  - What parts of your processor work and which parts do not work. This helps us to find partial credit.
  - For subcircuits (e.g., register file or ALU), explain their interfaces so that we can possibly test them individually.
- All submitted circuits will be tested for suspicious similarities to other circuits, and the test will uncover cheating, even if it is "hidden." Plagiarism of Logisim code will be treated as academic misconduct.
- Logisim implementations must use only the primitive gates (NOT, AND, OR, NAND, NOR, XOR), multiplexers, and D flip-flops. You may also use tunnels and splitters.
- You may not use any pre-existing Logisim circuits (i.e., that you could possibly find by searching the internet).
- Since we use the auto-grading system, you must label your processor's pins as stated in the specification!! Otherwise your circuit will not be considered as correct.

*Have fun!!*

## The instruction set

| instruction | opcode | type | usage | operation |
|---|---|---|---|---|
| nand | 0000 | R | nand $rd, $rs, $rt | $rd = $rs NAND $rt |
| xor | 0001 | R | xor $rd, $rs, $rt | $rd = $rs XOR $rt |
| addi | 0010 | I | addi $rt, $rs, Imm | $rt=$rs+Imm |
| add | 0011 | R | add $rd, $rs, $rt | $rd=$rs+$rt |
| sub | 0100 | R | sub $rd, $rs, $rt | $rd=$rs-$rt |
| shra | 0101 | R | shra $rd, $rs, &lt;shamt&gt; | $rd = $rs shifted &lt;shamt&gt; to right (**arithmetic** shift that preserves sign); shamt is unsigned. Arithmetic shift means that the bits shifted in on the left are the same as the original most-significant bit. |
| shl | 0110 | R | slh $rd, $rs, &lt;shamt&gt; | $rd = $rs shifted &lt;shamt&gt; to left; shamt is unsigned. |
| bgt | 0111 | I | bgt $rs, $rt, Imm | if ($rs>$rt) then PC=PC+1+Imm |
| beqz | 1000 | R | beqz $rs, Imm | if ($rs==0) then PC=PC+1+Imm |
| lw | 1001 | I | lw $rt, Imm($rs) | $rt = Mem[$rs+Imm] |
| sw | 1010 | I | sw $rt, Imm($rs) | Mem[$rs+Imm] = $rt |
| j | 1011 | J | j L | PC = L (upper 4 bits same) |
| jr | 1100 | R | jr $rs | PC = $rs |
| jal | 1101 | J | jal L | $r7=PC+1; PC = L |
| output | 1110 | R | output $rs | print $rs on a TTY display |
| input | 1111 | R | input $rd | $rd = keyboard input |

Table 1: Duke 250/16 Instructions

The formats of the R, I, and J type instructions are shown below: number of bits in parenthesis.

| | | | | | |
|---|---|---|---|---|---|
| **R-Type** | Opcode (4) | Rs (3) | Rt (3) | Rd (3) | Shamt (3) |
| **I-Type** | Opcode (4) | Rs (3) | Rt(3) | Immediate (6) | |
| **J-Type** | Opcode (4) | Address (12) | | | |

Immediate values are 6-bit signed 2's complement, so you must ensure that you sign extend it.

The *input* instruction is nonblocking, which means it will always complete and write something into the destination register. After a read, bits 15-8 of $rd ($rd[15..8]) should always be zero. If valid data was read from the keyboard, $rd[7] should be 0 and $rd[6..0] should be the 7-bit value read. If valid data was not available on the keyboard, $rd[7] should be 1 and $rd[6..0] should be 0. This has the effect that $rd should be the ASCII code read from the keyboard, or 128 to indicate that no data was available. You will use the keyboard input device available in Logisim.

The *output* instruction writes the 7-bit ascii character contained in the low 7 bits of $rs ($rs[6..0]) to the Logisim TTY output device. **Please use the TTY with the following specifications: 13 rows, 80 columns, and falling edge.**

## Registers

There are 8 general purpose registers: $r0-$r7. The register $r7 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. Users of your CPU are also advised to use $r6 as the stack pointer. $r0 is the constant value 0.

## The reset input

The processor has a single input called "**reset**" (please have an **exact** same name). This input resets the state of the computer by doing the following:

1. Reset PC to 0.
2. Clear the TTY display.
3. Clear the keyboard input buffer.
4. Reset the registers in the register file to all-zero.
5. NOTE: the "reset" input does NOT affect instruction memory or data memory.

## Memory layout

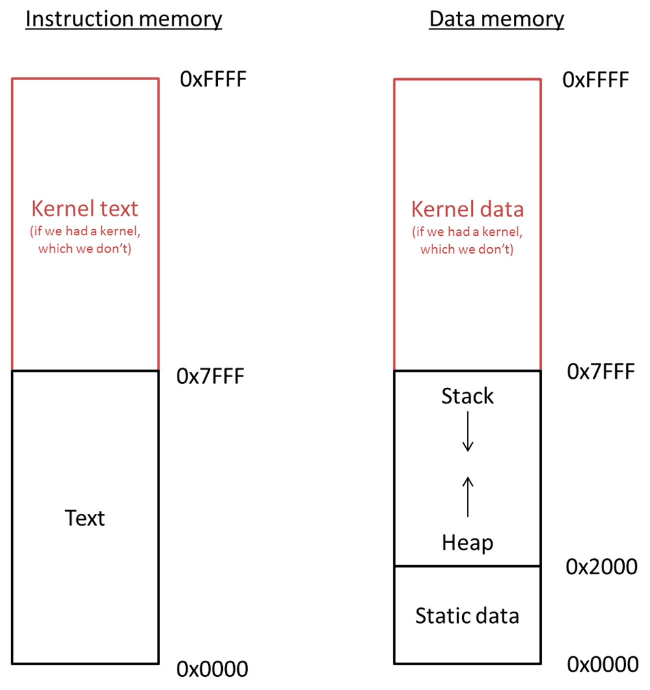

Figure 1: The memory model for your CPU.

The conventions for memory allocation, as performed by the assembler we provide you, are shown in Figure 1. This is what's known as a "Harvard architecture", which simply means that there is a separate memory space for instructions versus data. This maps naturally to the separate "instruction fetch" and "load word" facilities in our CPU's data path. In addition, we reserve the top half of each memory region for the kernel, even though no kernel or operating system will exist for this architecture. This means that, in instruction memory, user programs can have addresses from 0x0000 to 0x7FFF. In data memory, the first 8 Kwords (0x2000 words) are reserved for static data, with the heap starting at address 0x2000 and growing up. The stack starts at address 0x7FFF and grows down. **<u>REMEMBER: this is WORD-addressed, not BYTE-addressed.</u>**

You should use a Logisim ROM memory block for the instruction memory and a Logisim RAM block for data memory. You can edit the values in these memory blocks manually, but you can also right click (control click for Mac users) to open the popup menu that allows you to load an image file. These image files will be generated by the assembler described below.

## The Assembler and Simulator

We are providing an assembler and a simulator for you to generate test programs and to verify your program's behavior. The assembler and simulator are posted on Sakai -> Resources. 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 port the assembler to whatever machine you have.

The simulator is useful for debugging your design. Note that using the verbose flag of the simulator will spit out every instruction executed as well as the correct contents of every register—this is very helpful during debugging.

There are two pseudo-instructions available for use in your programs:

1. **`la $rs, label   # load address`**
2. **`halt`**

The **`la`** pseudo-instruction is converted into several actual machine instructions that have the effect of loading a 16-bit address into the specified register. The **`halt`** instruction is actually a branch that simply branches back to itself, creating an infinite loop (though when run with the simulator, this special branch is detected and causes the simulator to terminate).

**For information on using these tools, see the readme.txt included with it!**

## What you don't need to worry about

There are many aspects listed above that don't actually affect your job as the CPU architect. 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 even though we've said that $r6 is the 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 at 0x2000, 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+TTY), so you don't need to worry about inventing syscalls, protected instructions, exceptions, etc.
- The "Harvard architecture" (separate instruction and data memory spaces) will happen naturally if you simply design the CPU in the way we described in class. If this were a "von Neumann architecture" (a single flat memory space for code+data), then you'd just add some multiplexers

to choose between the instruction ROM and the data RAM based on the high bits of the address.

## Tips for carrying out this project

- You should break this project into smaller manageable chunks. You may want to design separate subcircuits (use the ADD Circuit option from the Project menu) for 1) ALU, 2) Instruction Decode, 3) Next PC computation, and 4) it is useful to have a sign extender. Logisim has some documentation for subcircuits. Note that for subcircuits with many inputs and outputs it gets tedious and Logisim is a little buggy sometimes (this will manifest when you try to connect to the subcircuit input/output within the main circuit).
- Write some very simple test programs that test each instruction or incrementally include more instructions. Start with ALU ops, then memory, then branch and jumps. This will make debugging much easier.
- www.asciitable.com is your very good friend.
- Use the "probe" feature to see what values wire bundles have at different points during execution. You can also use HEX displays to make it very easy to see values (but the circuit area gets large with those…)
- Think carefully about how you route wires around the circuit, keep things as neat as possible else debugging gets very difficult.
- You will use a lot of the splitter wiring component, it can be used to both split off wires and to bring wires together to create a bundle.
- The constant wiring element is your friend, use it where you can…
- There will be a lot of multiplexers. No MUXes should need an enable in your design, so you can set the properties of the MUX to disable that.
- Instruction memory ROM should be set to have 16 address bits and a data width of 16 bits.
- Data memory RAM should be set to have 16 address bits and a data width of 16 bits. (Note, this will actually give you more memory than what the above memory allocation says, but that's currently an assembler limitation. )
- The data memory RAM should be set to have separate load and store ports. You will use the write enable signal, but you can leave the select and load unconnected, that will make it behave as a combinational delay for load instructions.
- Remember that nearly all Logisim components have properties that allow you to change the input and/or output widths, etc. Use that to your advantage.
- There should only be a five clocked items in your design (PC register, register file, data memory, keyboard and TTY). Strong recommendation: clock the register file, data memory, and TTY on the **falling** edge of the clock.
- When debugging, use the single Tick feature or "Poke" the clock to cause it to transition (note: you need two pokes for a full clock cycle).
- When you execute programs with many instructions you can use the simulate feature to have the clock tick at a specified frequency. You'll want to do this for the sample program provided since it executes over 1000 instructions.