Homework #4 – Processor Core Design

Due Wednesday, March 30 at 5 pm
150 points

You must do all work individually, and you must submit your work electronically via Sakai.

The assignment is worth 150 points, and partial credit will be given.

All submitted circuits will be tested for suspicious similarities to other circuits, and the test will uncover cheating, even if it is "hidden."

The project for this course is the design and implementation of the Duke 250/16, a 16-bit MIPS-like, **word-addressed (not byte-addressed)** RISC architecture that has been scaled down and simplified to make it feasible for a class project. I 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.

Table 1: Duke 250/16 Instructions

instruction	opcode	type	usage	operation
add	0000	R	add $rd, $rs, $rt	$rd=$rs+$rt
addi	0001	I	addi $rt, $rs, Imm	$rt=$rs+Imm
sub	0010	R	sub $rd, $rs, $rt	$rd=$rs-$rt
and	0011	R	and $rd, $rs, $rt	$rd=$rs AND $rt
xor	0100	R	xor $rd, $rs, $rt	$rd = $rs XOR $rt
sll	0101	R	sll $rd, $rs, <shamt>	$rd = $rs shifted <shamt> to left; shamt is unsigned
sra	0110	R	sra $rd, $rs, <shamt>	$rd = $rs shifted <shamt> to right (arithmetic shift!); shamt is unsigned
lw	0111	I	lw $rt, D($rs)	$rt = Mem[$rs+D]
sw	1000	I	sw $rt, D($rs)	Mem[$rs+D] = $rt
beq	1001	I	beq $rs, $rt, B	if ($rs==$rt) then PC=PC+1+B
blt	1010	I	blt $rs, $rt, B	if ($rs<$rt) then PC=PC+1+B
j	1011	J	J L	PC = L (upper 4 bits same)
jr	1100	R	jr $rd	PC=$rd
jal	1101	J	jal L	$r7=PC+1; PC = L
input	1110	R	input $rd	$rd = keyboard input
output	1111	R	output $rs	print $rs on a TTY display

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

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 $rt ($rt[15..8]) should always be zero. $rt[7] should be 0 if and only if valid data was read from the keyboard controller, else it should be 1. $rt[6..0] should contain the (7-bit) ASCII representation of the character read from the buffer. Therefore, if $rt < 256, $rt is equal to the ASCII code (since $rt[15..8] will be zero). 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.

There are 8 general purpose registers: $r0-$r7. The register $r7 is the link register for the jal instruction. (You may write it with other instructions, but I'd strongly discourage you from doing this.) Use $r6 as the stack pointer. $r0 is the constant value 0 (i.e., an instruction can specify it as a destination but "writing" to $r0 doesn't change its value).

The processor has a single input called Reset. 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.

NOTE: the Reset input does NOT affect instruction memory or data memory.

IMPORTANT: On this assignment, you may only use the following Logisim elements:

1. Anything from the "Wiring" folder
2. Anything from the "Gates" folder
3. Anything from the "Plexers" folder
4. From the "Memory" folder: "D Flip-Flop", "RAM", and "ROM"
5. From the "Input/Output" folder: "Keyboard", "TTY".
6. The "Text" tool
7. Any sub-circuits you develop from the above

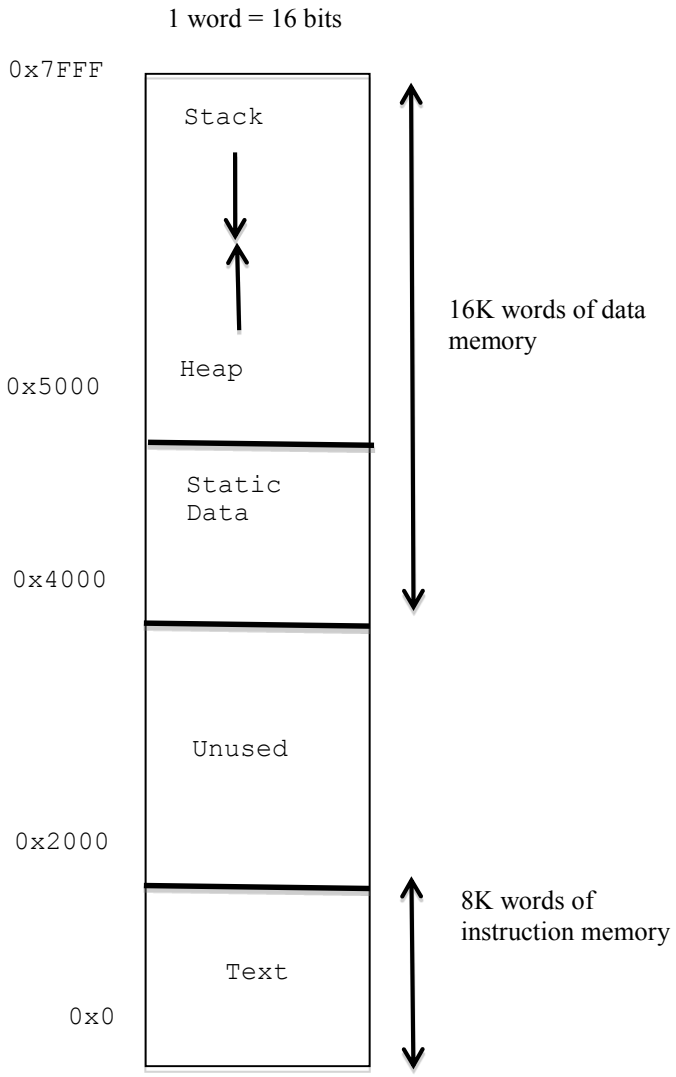

Figure 1: Address Space Layout

The conventions for memory allocation, as performed by the assembler we provide you, are shown in Figure 1. Note that you have an address space of 24 Kwords (each word is 16 bits), divided into 8K words of instruction memory and 16K words of data memory. Addresses between 0 and 0x1fff correspond to the 8Kword instruction memory, and addresses between 0x4000 and 0x7fff correspond to the 16K word data memory (i.e., 0x4000 is the address of the first location in the data memory). Addresses between 0x2000 and 0x3fff do not correspond to locations in physical memory. For those of you wishing to write self-modifying code, you're out of luck. **REMEMBER: this is WORD-addressed, not BYTE-addressed.**

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 available on Sakai (files in directory "asm-sim" under Resources). These are very limited tools (e.g., no hex values for constants - only integers). We have tested the assembler on Linux machines (i.e., teerXX.oit.duke.edu). You will have to copy the generated memory image files to your own machine, or you can port the assembler to whatever machine type 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) ldia $rs, label # load address
2) halt

The ldia 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.

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) Register File, 4) Next PC computation, and 5) I find it 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. (Note, this will actually give you more memory than what the above memory allocation says, but that's currently an assembler limitation.)
- 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 first range of addresses from 0x0 to 0x4000 will be initialized to 0 by the assembler.)
- 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. You are free to use whatever components are available with the default Logisim Library and the register file we provide.
- 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.