Homework #4 – Processor Core Design

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:

    o What parts of your processor work and which parts do not work. This helps us to find partial credit.

    o 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 components specified in the "Logisim restrictions" section later in this document.

- For successful automated grading, your circuit must meet the requirements specified in the "Automated testing" section.

- You may not use any pre-existing Logisim circuits (i.e., that you could possibly find by searching the internet).

Have fun!!

The instruction set

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
not	0011	R	not $rd, $rs	$rd = NOT $rs
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
srl	0110	R	srl $rd, $rs, <shamt>	$rd = $rs shifted <shamt> to right (logical shift: no special treatment of sign bit); 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
bne	1001	I	bne $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 $rs	PC=$rs
jal	1101	J	jal L	$r7=PC+1; PC = L
input	1110	I	input $rt	$rt = keyboard input
output	1111	I	output $rs	print $rs on a TTY display

Table 1: Duke 250/16 Instructions

The formats of the R, I, and J type instructions are shown below: number of bits in parenthesis, with the specific bit numbers shown in brackets (remember that the least significant or rightmost bit is bit 0).

R-Type	Opcode (4) [12..15]	Rs (3) [9..11]	Rt (3) [6..8]	Rd (3) [3..5]	Shamt (3) [0..2]
I-Type	Opcode (4) [12..15]	Rs (3) [9..11]	Rt(3) [6..8]	Immediate (6) [0..5]	
J-Type	Opcode (4) [12..15]	Address (12) [0..11]			

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. If valid data was read from the keyboard, $rt[7] should be 0 and $rt[6..0] should be the 7-bit value read. If valid data was not available on the keyboard, $rt[7] should be 1 and $rt[6..0] should be 0. This has the effect that $rt 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 (i.e., an instruction can specify it as a destination but "writing" to $r0 must not change its value).

Implementation note: Your register file's read ports _must_ use Tri-state Buffers and a Decoder rather than a big Mux (as described in the class notes regarding the register file). Logically, the two approaches are equivalent, but in real implementation, the Tri-state Buffer approach is much faster. Besides, this is a great chance to play with Tri-states. Solutions using a Mux within the register file will be penalized up to 10%.

The reset input

The processor has a single input called "`reset`"; the name must match exactly. 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.
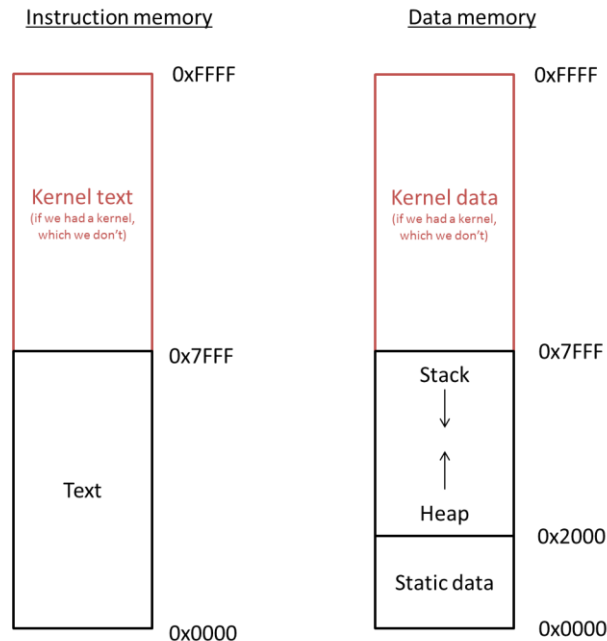
Memory layout

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. **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.

Logisim restrictions

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", and "Button".
6. The "Text" tool
7. Any sub-circuits you develop from the above

The penalty for violating these restrictions can be up to 75% of total score!

Automated testing

An automated self-test tool has been provided. **For the self-test tool to work, your circuit must meet the following requirements:**

- Circuit is called <u>hw4.circ</u> and is stored in the same directory as the test tool and associated data.

- You must name your register file component "`RegisterFile`" (including capitalization).

- You must name your reset input "`reset`" (including capitalization).

- Testing is based on the Probe component. You must place a Probe on each register in your register file named "`r0`", "`r1`", "`r2`", etc.

- Make sure that the default state of all DFFs is 0 (i.e. that you don't leave a DFF inside a register 'poked' to a 1 value when you save). Most of the tests toggle the reset line to ignore this issue, but the `io` test cannot, as that would otherwise reset the keyboard buffer.

- You may use Probes for your own purposes, but only if you leave their label *blank*. The tester filters out unlabeled probes, but any labeled probes other than "`r0`", "`r1`", etc., will throw off the results.

- Configure the TTY with the following specifications: 13 rows, 80 columns, and falling edge.

- You may not use a ROM component for any purpose other than your instruction memory, as the console automation will overwrite every ROM component in your circuit with the instruction data.

- The tool has been tested on the Duke Linux environment, so that is where we recommend you run it. This will mean transferring your circuit to your Duke home directory via SMB (Windows share), SFTP, etc.

The self-test tool is similar to those you've used already. You can run "./hw4test.py" to see a usage message. It produces "*_actual_*.txt" and "*_diff_*.txt" files so you can see your output and the differences between that and what was expected. If you want to run an individual command line test manually, run the hw4test.py tool with the "-v" option to see the exact java command used to execute the test, which you can then use yourself. The assembly source files for the tests are provided in the `assembly-files` directory.

A note on the philosophy behind providing this tester: the goal here is to help you determine any bugs you might have missed and *supplement* your testing effort. Staring at diff files from a test you do not understand will generally NOT help you debug your circuit. It is expected that you'll need to develop your own specific tests using the assembler and simulator 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 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 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 $rd, label    # load address`
2. `halt`

The `la` pseudo-instruction is converted into multiple actual machine instructions that have the effect of loading a 16-bit address into the specified register (specifically, a series of `addi` and `sll` instructions). Specifically, the transformation is that:

```
la $rd, ADDR
```

Will become the following, where the bracket notation indicates bits within ADDR:

```
addi $rd, $r0, ADDR[15..11]
sll  $rd, $rd, 5
addi $rd, $rd, ADDR[10..6]
sll  $rd, $rd, 5
addi $rd, $rd, ADDR[5..1]
sll  $rd, $rd, 1
addi $rd, $rd, ADDR[0]
```

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). Of course, branches are conditional, so to guarantee we loop, a sequence of three instructions is emitted, where one of the branches is guaranteed to be taken:

```
bne  $r0, $r7, -1
addi $r7, $r7, 1
bne  $r0, $r7, -1
```

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

Below is a screenshot of these tools being used to assemble and test an included example program:

```
tkb13@login-teer-03:~/asm-sim $ ls
asm.cpp  example.s  format.h  Makefile  readme.txt  sim.cpp
tkb13@login-teer-03:~/asm-sim $ make
g++ -g -o asm asm.cpp
g++ -g -o sim sim.cpp
tkb13@login-teer-03:~/asm-sim $ ./asm example.s
tkb13@login-teer-03:~/asm-sim $ ./sim -F example.imem.lgsim example.dmem.lgsim
Hello!tkb13@login-teer-03:~/asm-sim $ ./sim -v -F example.imem.lgsim example.dmem.lgsim
Regs: [0000 0000 0000 0000 0000 0000 0000 0000]  0000 1040 addi $r1,$r0,0
Regs: [0000 0000 0000 0000 0000 0000 0000 0000]  0001 520d sll $r1,$r1,5
Regs: [0000 0000 0000 0000 0000 0000 0000 0000]  0002 1240 addi $r1,$r1,0
Regs: [0000 0000 0000 0000 0000 0000 0000 0000]  0003 520d sll $r1,$r1,5
Regs: [0000 0000 0000 0000 0000 0000 0000 0000]  0004 1240 addi $r1,$r1,0
Regs: [0000 0000 0000 0000 0000 0000 0000 0000]  0005 5209 sll $r1,$r1,1
Regs: [0000 0000 0000 0000 0000 0000 0000 0000]  0006 1240 addi $r1,$r1,0
Regs: [0000 0000 0000 0000 0000 0000 0000 0000]  0007 d01d jal 29
Regs: [0000 0000 0000 0000 0000 0000 0000 0008]  001d 72c0 lw $r3,0($r1)
Regs: [0000 0000 0000 0048 0000 0000 0000 0008]  001e 9601 bne $r3,$r0,1
Regs: [0000 0000 0000 0048 0000 0000 0000 0008]  0020 f600 output $r3
HRegs: [0000 0000 0000 0048 0000 0000 0000 0008]  0021 1241 addi $r1,$r1,1
Regs: [0000 0001 0000 0048 0000 0000 0000 0008]  0022 b01d j 29
Regs: [0000 0001 0000 0048 0000 0000 0000 0008]  001d 72c0 lw $r3,0($r1)
Regs: [0000 0001 0000 0065 0000 0000 0000 0008]  001e 9601 bne $r3,$r0,1
Regs: [0000 0001 0000 0065 0000 0000 0000 0008]  0020 f600 output $r3
eRegs: [0000 0001 0000 0065 0000 0000 0000 0008]  0021 1241 addi $r1,$r1,1
Regs: [0000 0002 0000 0065 0000 0000 0000 0008]  0022 b01d j 29
Regs: [0000 0002 0000 0065 0000 0000 0000 0008]  001d 72c0 lw $r3,0($r1)
Regs: [0000 0002 0000 006c 0000 0000 0000 0008]  001e 9601 bne $r3,$r0,1
Regs: [0000 0002 0000 006c 0000 0000 0000 0008]  0020 f600 output $r3
lRegs: [0000 0002 0000 006c 0000 0000 0000 0008]  0021 1241 addi $r1,$r1,1
Regs: [0000 0003 0000 006c 0000 0000 0000 0008]  0022 b01d j 29
Regs: [0000 0003 0000 006c 0000 0000 0000 0008]  001d 72c0 lw $r3,0($r1)
Regs: [0000 0003 0000 006c 0000 0000 0000 0008]  001e 9601 bne $r3,$r0,1
Regs: [0000 0003 0000 006c 0000 0000 0000 0008]  0020 f600 output $r3
lRegs: [0000 0003 0000 006c 0000 0000 0000 0008]  0021 1241 addi $r1,$r1,1
Regs: [0000 0004 0000 006c 0000 0000 0000 0008]  0022 b01d j 29
Regs: [0000 0004 0000 006c 0000 0000 0000 0008]  001d 72c0 lw $r3,0($r1)
Regs: [0000 0004 0000 006f 0000 0000 0000 0008]  001e 9601 bne $r3,$r0,1
Regs: [0000 0004 0000 006f 0000 0000 0000 0008]  0020 f600 output $r3
oRegs: [0000 0004 0000 006f 0000 0000 0000 0008]  0021 1241 addi $r1,$r1,1
Regs: [0000 0005 0000 006f 0000 0000 0000 0008]  0022 b01d j 29
Regs: [0000 0005 0000 006f 0000 0000 0000 0008]  001d 72c0 lw $r3,0($r1)
Regs: [0000 0005 0000 0021 0000 0000 0000 0008]  001e 9601 bne $r3,$r0,1
Regs: [0000 0005 0000 0021 0000 0000 0000 0008]  0020 f600 output $r3
!Regs: [0000 0005 0000 0021 0000 0000 0000 0008]  0021 1241 addi $r1,$r1,1
Regs: [0000 0006 0000 0021 0000 0000 0000 0008]  0022 b01d j 29
Regs: [0000 0006 0000 0021 0000 0000 0000 0008]  001d 72c0 lw $r3,0($r1)
Regs: [0000 0006 0000 0000 0000 0000 0000 0008]  001e 9601 bne $r3,$r0,1
Regs: [0000 0006 0000 0000 0000 0000 0000 0008]  001f b023 j 35
Regs: [0000 0006 0000 0000 0000 0000 0000 0008]  0023 ce00 jr $r7
Regs: [0000 0006 0000 0000 0000 0000 0000 0008]  0008 e140 input $r5
Regs: [0000 0006 0000 0000 0000 0080 0000 0008]  0009 1040 addi $r1,$r0,0
Regs: [0000 0000 0000 0000 0000 0080 0000 0008]  000a 520d sll $r1,$r1,5
Regs: [0000 0000 0000 0000 0000 0080 0000 0008]  000b 1240 addi $r1,$r1,0
Regs: [0000 0000 0000 0000 0000 0080 0000 0008]  000c 520d sll $r1,$r1,5
Regs: [0000 0000 0000 0000 0000 0080 0000 0008]  000d 1243 addi $r1,$r1,3
Regs: [0000 0003 0000 0000 0000 0080 0000 0008]  000e 5209 sll $r1,$r1,1
Regs: [0000 0006 0000 0000 0000 0080 0000 0008]  000f 1241 addi $r1,$r1,1
Regs: [0000 0007 0000 0000 0000 0080 0000 0008]  0010 7280 lw $r2,0($r1)
Regs: [0000 0007 03e8 0000 0000 0080 0000 0008]  0011 2558 sub $r3,$r2,$r5
Regs: [0000 0007 03e8 0368 0000 0080 0000 0008]  0012 3420 not $r4,$r2
Regs: [0000 0007 03e8 0368 fc17 0080 0000 0008]  0013 4528 xor $r5,$r2,$r4
Regs: [0000 0007 03e8 0368 fc17 ffff 0000 0008]  0014 5432 sll $r6,$r2,2
Regs: [0000 0007 03e8 0368 fc17 ffff 0fa0 0008]  0015 6432 srl $r6,$r2,2
Regs: [0000 0007 03e8 0368 fc17 ffff 00fa 0008]  0016 14be addi $r2,$r2,-2
Regs: [0000 0007 03e6 0368 fc17 ffff 00fa 0008]  0017 6411 srl $r2,$r2,1
Regs: [0000 0007 01f3 0368 fc17 ffff 00fa 0008]  0018 a081 blt $r0,$r2,1
Regs: [0000 0007 01f3 0368 fc17 ffff 00fa 0008]  001a 91ff bne $r0,$r7,-1
60 dynamic instructions executed
tkb13@login-teer-03:~/asm-sim $
```

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) 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.
- 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.