《 Computer Architecture – CPU Logisim HW》

1. HW Description

Your task is [illegible] CPU250/16, which is a RISC (Reduced Instruction Set Computing) [illegible] sembles MIPS, but is word-addressed and uses 16-bit words [illegible] dressing. To complete this task, a single cycle implementat[illegible]ure will be designed using Logisim. The specificatior[illegible]sted in the provided Table.

Notes:

- 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).

A. CPU250/16 Instruction Set:

Instruction	Opcode	Type	usage	operation
lw	1000	I	lw $rt, D($rs)	$rt = Mem[$rs+D]
sw	0111	I	sw $rt, D($rs)	Mem[$rs+D] = $rt
beq	0110	I	beq $rs, $rt, B	if ($rs==$rt) then PC=PC+1+B; else PC=PC+1
bgt	0101	I	bgt $rs, $rt, B	if ($rs>$rt) then PC=PC+1+B; else PC=PC+1
j	0100	J	j L	PC = L (upper 4 bits zeroed)
jr	0011	R	jr $rs	PC = $rs
jal	0010	J	jal L	$r7=PC+1; PC = L
input	0001	I	input $rt	$rt = keyboard input
output	0000	I	output $rs	print $rs on a TTY display
add	1111	R	add $rd, $rs, $rt	$rd = $rs + $rt
addi	1110	I	addi $rt, $rs, Imm	$rt = $rs + Imm

sub	1101	R	sub $rd, $rs, $rt	
sll	1100	R	sll $rd, $rs, <shamt>	$rd = $rs << shamt (shamt is unsigned)
srl	1011	R	srl $rd, $rs, <shamt>	$rd = $rs >> shamt (logical shift: no special treatment of sign bit; shamt is unsigned)
and	1010	R	and $rd, $rs, $rt	$rd = $rs & $rt
xor	1001	R	xor $rd, $rs, $rt	$rd = $rs XOR $rt

B. About R, I J Instruction Types:

The R, I, and J type instructions are presented below, indicating the number of bits used for each instruction type within parentheses. The specific bit numbers for each instruction type are also provided in brackets, keeping in mind that the rightmost bit is referred to as 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 represented using 6 bits in signed 2's complement format, and must be sign-extended.

The *input instruction* operates in a nonblocking manner, meaning that it always completes and writes a value into the destination register.

- When data is read, bits 15-8 of $rt ($rt[15..8]) must always be set to 0.
- If valid data is retrieved from the keyboard, $rt[7] should be 0, and $rt[6..0] should represent the 7-bit value that was read.
- If valid data is not available on the keyboard, $rt[7] should be 1, and $rt[6..0] should be set to 0.
- As a result, $rt should contain the ASCII code that was read from the keyboard, or 128 to indicate that no data was available. The keyboard input device available in Logisim will be used.

The *output instruction* writes a 7-bit ASCII character contained in the lower 7 bits of $rs ($rs[6..0]) to the Logisim TTY output device. The TTY should be configured with 13 rows, 80 columns, and operate on the rising edge.

2. Registers

CPU 250/16 has 8 general purpose registers: $r0-$r7

- The register $r7 is the link register for the jal instruction (similar to $ra in MIPS).
  - Although users of the CPU have the ability to write to $r7 using other instru[illegible]uld disrupt function call/return operations.
- The Register[illegible] stack pointer, while $r0 should always hold the constant val[illegible]

Regarding the impl[illegible]portant to note that the read ports of the register file must use Tri-st[illegible]ecoder instead of a large Mux.

3. Reset Input

The CPU features a sole input referred to as "reset", which must be named exactly as such. This input allows for the computer's state to be reset by performing the following actions:

1. Asynchronously reset the program counter (PC) to 0.
2. Asynchronously clear the TTY display.
3. Asynchronously clear the keyboard input buffer.
4. Asynchronously reset all registers in the register file to zero.

Please note that the Reset input will not have any impact on the instruction or data memory. This can be accomplished by linking the reset input to the "clear" or "reset" input pins on the D Flipflops and IO devices that are utilized.

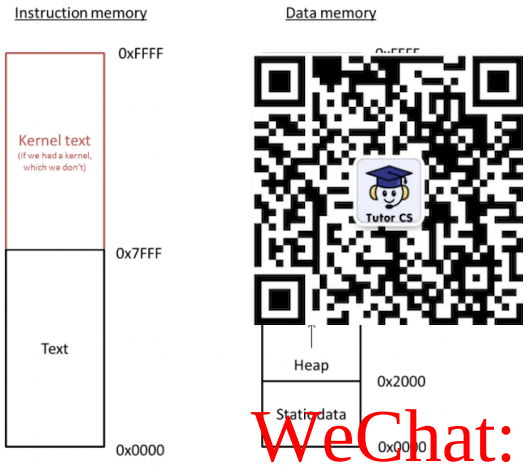
4. Clocking the CPU

Please not that there should only be five clocked items in your design (PC register, register file, data memory, keyboard and TTY).

CLOCK REQUIREMENT:

• Clock the register file, and TTY on the rising edge of the clock. •
Clock the PC register, data memory, and keyboard on the falling edge.

5. Memory Layout



The assembler we provide follows the conventions for memory allocation as shown in Figure 1. This is referred to as a "Harvard Architecture", which simply means that separate memory spaces are designated for instructions and data. This aligns with the distinct "instruction fetch" and "load word" functions present in our CPU's data path. Furthermore, we reserve the top half of each memory region for the kernel, even though no kernel or operating system exists for this architecture. Therefore, user programs can have addresses ranging from 0x0000 to 0x7FFF in instruction memory, while the first 8 kwords (0x2000 words) in data memory are reserved for static data, with the heap starting at address 0x2000 and expanding upward, and the stack starting at address 0x7FFF and expanding downward. Please note that this architecture is WORD-addressed, not BYTE-addressed, meaning that each address corresponds to a complete 16-bit word.

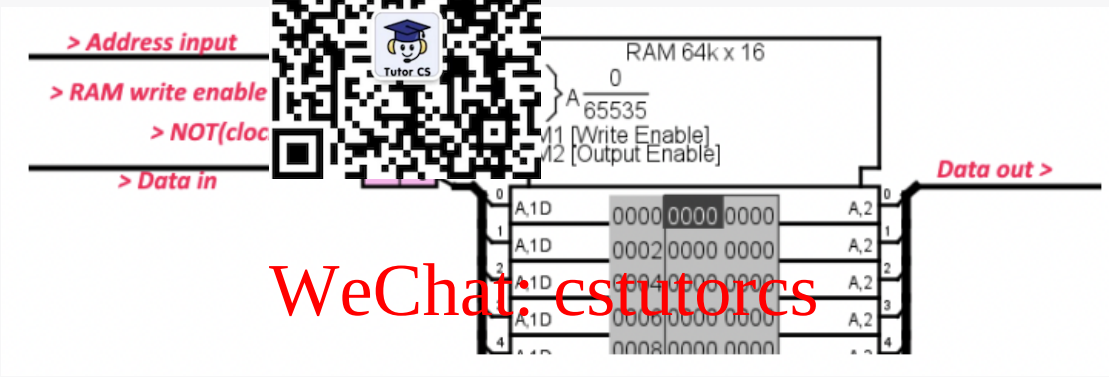
To implement this, you can use a Logisim ROM memory block for instruction memory and a Logisim RAM block for data memory. You may edit the values in these memory blocks manually, or alternatively, right-click (or control-click for Mac users) to access a popup menu that enables you to load an image file. These image files will be created by the assembler discussed later in this document.

Logisim Restrictions: 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”
   **Note: when deploying RAM in Logisim Evolution, you’ll need to couple it with a “memory_latch” circuit we are providing you; see the section “Using RAM in Logisim Evolution” below.
5. From the “Input/Output” folder: “Keyboard”, “TTY”, and “Button”.
6. The “Text” tool

Using RAM in Logisim Evolution

**We will provide you a folder with the automated tester (hwtest.py and associated files), the example programs used by the tester (programs/), and the assembler+simulat



1. Lay down your RAM in your main circuit and configure as follows:
    a. Address and data width should be 16-bit.
    b. b. Triggering should be on high-level instead of on an edge.
    c. c. Databus implementation should be separate read/write ports.

2. Find the memorylatch.circ provided via Folder, and merge the contents your cpu.circ: With your CPU open, use the "File | Merge" option in Logisim Evolution to merge in the memorylatch.circ file. This will add a memory_latch subcircuit.

3. Place a memory_latch instance before your RAM in your main circuit. Instead of sending your data and address directly to your RAM, you should send them to the memory_latch instance (along with a NOT(clock) signal and a write_enable signal) and this latch will generate an "address_latched", "data_in_latched", write-enable latched ("WE_latched") and output-enable latched ("OE_latched") which you should then send to the RAM that is level triggered. You could change the representation to hook into your RAM compactly, as shown above.

What 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 e said that $r6 is the stack pointer, you as the CPU designer dor ng special to allow or enforce this.

- Heap manag he stack; it's maintained by the programmers so you don't ha make it exist. This means that even though the heap is supp 000, 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