**Design and implementation of a simple 16-bit CPU**

**Tatiana Castro Vélez**

**Introduction**

In the Fall semester 2012 Humberto Ortiz-Zuazaga taught Computer Architecture I, and asked the students to work on implementing an 16 bit CPU in Logisim. Let's see if I finally got the spec right.

**Specification**

The idea is to build a simple CPU up from logic gates. I asked students to use only and gates, or gates or inverters. I used RAM for the program memory, but we built everything else up.

The CPU will execute 16-bit instructions.

With 16 registers we can let 4 bits represent the rs, rt and rd registers, and

lw

and

sw

compute the target address as rs+rt.

ldhi

loads an 8-bit immediate into the upper 8 bits of a register.

bz

jumps to PC + imm8.

j

jumps to PC + imm12.

jal

saves PC+1 in r1 and jumps to PC+imm8.

**Sample code**

With the format described above we can write a small program to count to n in 16-bit pseudoMIPS assembly

add $1, $0, $0

addi $1, 10 ; set n to 10

add $2, $0, $0 ; set i to 0

loop:

addi $2, 1

addi $1, -1

bnz $1, loop ; loop is PC-2

We can translate this by hand to the machine language:

binary | hex

0000 0001 0000 0000 | 0100

1000 0001 0000 1010 | 810A

0000 0010 0000 0000 | 0200

1000 0010 0000 0001 | 8201

1000 0001 1111 1111 | 81FF

1011 0001 1111 1110 | B1FE

Loading the instruction memory with these values will allow us to simulate the execution of the program.

**Design**

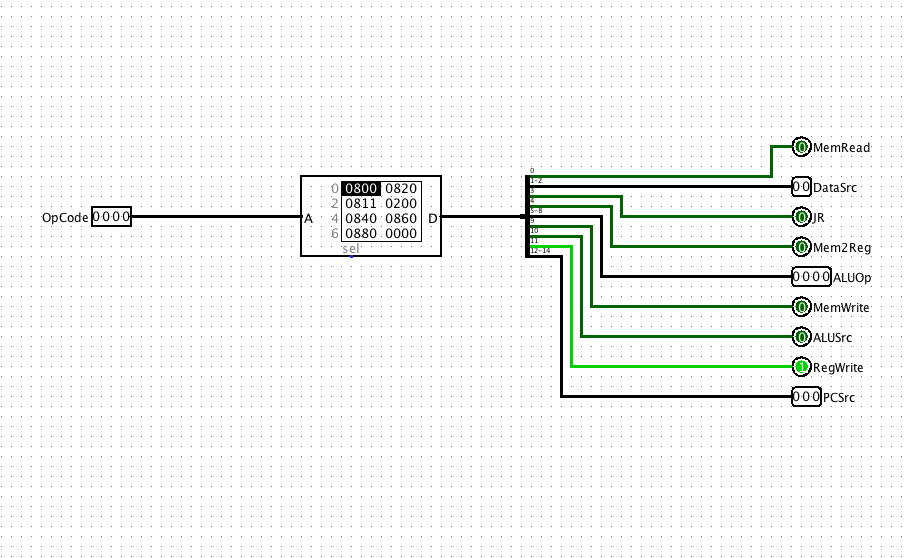
The design is a simplified version of the MIPS design presented in the textbook David A Patterson, John L Hennessy, *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*, 2009. ISBN 978-0-12-374493-7

**Control Table:**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instruction | OpCode | MemRead | DataSrc | JR | Mem2Reg | ALU-Op | MemWrite | ALUSrc | Reg  Write | PCSrc |
| Add | 0000 | 0 | 00 | 0 | 0 | 0000 | 0 | 0 | 1 | 000 |
| Addi | 1000 | 0 | 00 | 0 | 0 | 0000 | 0 | 1 | 1 | 000 |
| Sub | 0001 | 0 | 00 | 0 | 0 | 0001 | 0 | 0 | 1 | 000 |
| And | 0100 | 0 | 00 | 0 | 0 | 0010 | 0 | 0 | 1 | 000 |
| Or | 0101 | 0 | 00 | 0 | 0 | 0011 | 0 | 0 | 1 | 000 |
| Not | 0110 | 0 | 00 | 0 | 0 | 0100 | 0 | 0 | 1 | 000 |
| Lw | 0010 | 1 | 00 | 0 | 1 | 0000 | 0 | 0 | 1 | 000 |
| Sw | 0011 | 0 | 00 | 0 | 0 | 0000 | 1 | 0 | 0 | 000 |
| Bz | 1010 | 0 | 00 | 0 | 0 | 0001 | 0 | 1 | 0 | 011 |
| Bnz | 1011 | 0 | 00 | 0 | 0 | 0001 | 0 | 1 | 0 | 100 |
| J | 1111 | 0 | 00 | 0 | 0 | 0000 | 0 | 0  (no importa) | 0 | 101 |
| Jr | 1110 | 0 | 01 | 1 | 0 | 0000 | 0 | 0  (no importa) | 0 | 010 |
| Jal | 1101 | 0 | 0 (no importa) | 0 | 0 | 0000 | 0 | 0 (no importa) | 1 | 000 (no importa) |
| Ldhi | 1001 | 0 | 10 | 0 | 0 | 0000 | 0 | 0 (no importa) | 1 | 000 |

To build the control we needed the opcode of the address. For the design in logisim, I put a 4 bit pin to cover the opcode of the address and connected it to a RAM with 4 bit width and 15 data bit width. Why 15 data bit width? We need to count how many bits does every instruction have and according to that we enter the data bit width. In this case there were 15 data bit width. Then we connect the memory to a splitter and connect it to output pins according to there respective data bit widths. When you are done with that you have to convert the data you have in your control table to hexadecimal for you to load your information of the control to memory. Once you are done with that, you are done with the control of your CPU.

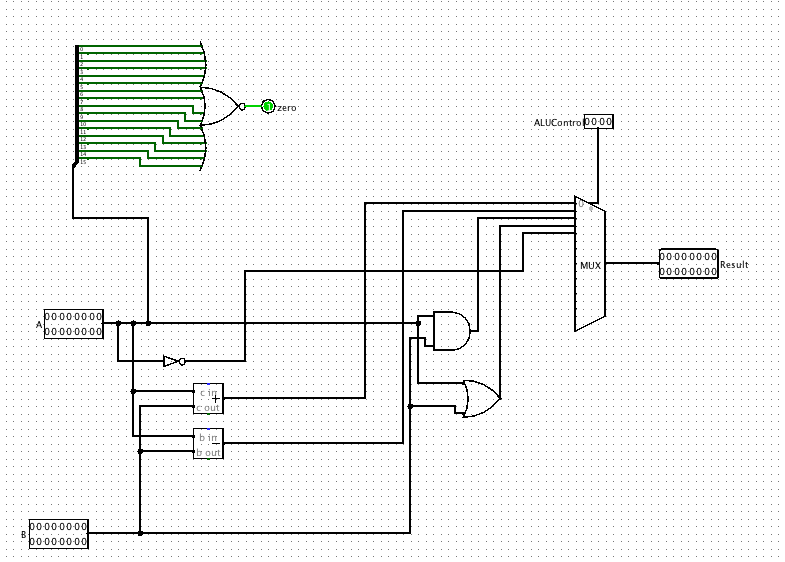
Hexadecimal values:

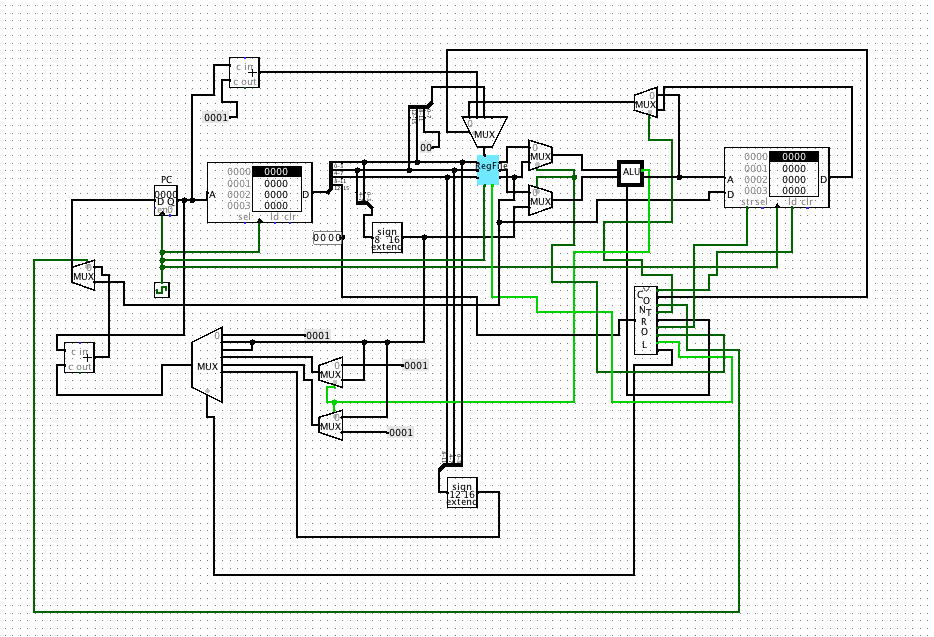
* Add: 800
* Addi: c00
* Sub: 820
* And: 840
* Or: 860
* Not: 880
* Lw: 810
* Sw: 200
* Bz: 3420
* Bnz: 4420
* J: 5000
* Jr: 200A
* Jal: 800
* dhi: 8024

**ALU:**

The arithmetic logic unit (ALU) is the brawn of the computer, the device that performs the arithmetic operations like addition and subtraction or logical operations like AND and OR. The ALU is constructed from four hardware building blocks (AND and OR gates, inverters, and multiplexors).

In my CPU ALU we have the adder and substractor. The ALU Control and have an and, or and nor like the instructions requested. We also have a part that tells us the zero.



**CPU:**

We will be examining an implementation that includes a subset of the core MIPS instruction set:

■ The memory-reference instructions load word (lw) and store word (sw)

■The arithmetic-logical instructions add, sub, AND, OR, and slt

■ The instructions branch equal (beq) and jump (j), which we add last

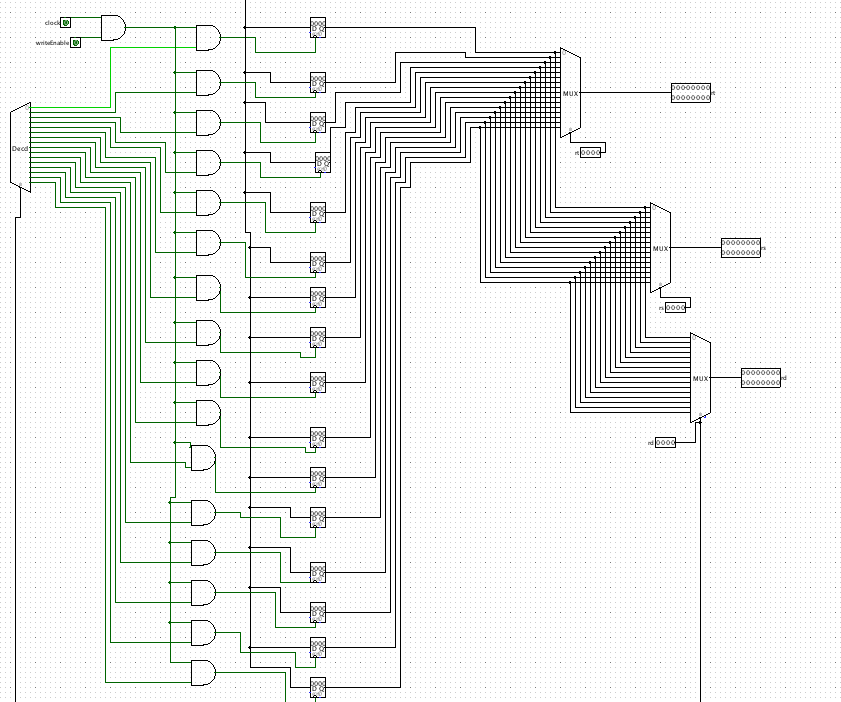
For every instruction, the first two steps are identical:

* 1.  Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
* 2.  Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one regis ter, but most other instructions require that we read two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class. Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction. The simplicity and regularity of the MIPS instruction set simplifies the implementation by making the execution of many of the instruction classes similar.

For example, all instruction classes, except jump, use the arithmetic-logical unit (ALU) after reading the registers. The memory-reference instructions use the ALU for an address calculation, the arithmetic-logical instructions for the opera tion execution, and branches for comparison. After using the ALU, the actions required to complete various instruction classes differ. A memory-reference instruction will need to access the memory either to read data for a load or write data for a store. An arithmetic-logical or load instruction must write the data from the ALU or memory back into a register. Lastly, for a branch instruction, we may need to change the next instruction address based on the comparison; other wise, the PC should be incremented by 4 to get the address of the next instruction.

**Register File:**



A register file is an array of processor registers in a central processing unit (CPU). Modern integrated circuit-based register files are usually implemented by way of fast static RAMs with multiple ports. Such RAMs are distinguished by having dedicated read and write ports.

The instruction set architecture of a CPU will almost always define a set of registers, which are used to stage data between memory and the functional units on the chip. In simpler CPUs, these *architectural registers* correspond one-for-one to the entries in a physical register file within the CPU.

**Decoder:**

* The decoder is a series of AND gates that drive word lines.

There is one decoder per read or write port. If the array has four read and two write ports, for example, it has 6 word lines per bit cell in the array, and six AND gates per row in the decoder. Note that the decoder has to be pitch matched to the array, which forces those AND gates to be wide and short