Project Log of the OZ-1 Processor - December 19, 2008 to January 26, 2009

Ben Oztalay

Friday, December 19, 2008

I read an article about a person who built his own 8-bit processor from scratch, and after some research, I found a Wikibook on microprocessor design (http://en.wikibooks.org/wiki/Microprocessor\_Design) that I read from front to back. The idea of making a simple 8-bit, single-cycle processor seemed feasible and within reach. While definitely not comfortably within my abilities, it was a challenge I thought would be fun and interesting to face. It seemed like all I needed to do was design a simple 8-bit parallel ALU, and instruction set architecture (ISA), and an instruction decoder.

I quickly started jotting down an overview of the processor I intend to build. The list soon morphed as I thought out various issues, such as if I'd need 8-bit instructions, or 16-bit. Other things to consider were what capabilities the ALU should have, how much RAM, if the processor should have separate instruction and data RAM. If the instruction RAM should actually be ROM so it can be removed for simpler programming, then which type of ROM? Should the processor have input and output pins to interact with outside devices? How many bits should the ALU be able to handle? How many programmer-visible registers should it have? What about the flags register, which flags should the processor include?

After many scratches on a piece of scrap paper, I came up with an almost-finalized overview of the processor. It looks something like this:

- 8-bit, single-cycle processor with 16-bit instructions and an 8-bit address bus

- Capable of addition, subtraction, XOR, OR, and AND operations

- Separate data RAM and instruction EEPROM

-Up to 256 bytes of RAM, as well as 256 bytes of EEPROM for instructions

- 3 programmer-visible registers: A, B and the Flags register, but the Flags register will not be editable and will only contain flags C, Z, and NZ

- Only 1 programmer-invisible register, the Program Counter (PC)

- Will have 16 dedicated output pins, no input pins

- Will be made out of wire-wrapped CMOS logic ICs

- The ALU will be an 8-bit parallel ALU

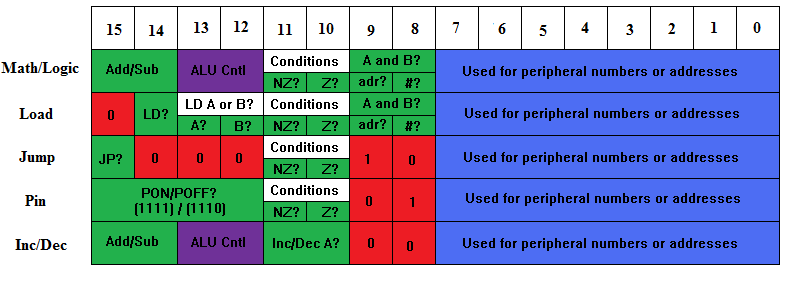
My reasons for choosing these properties were based almost solely on the goal of making a simple, single-cycle processor. Almost every place I could, I substituted performance for simplicity. I don't need this processor to run at blazing speeds. In fact, the clock frequency will most likely be at around 10Hz so I can see the processor at work.

Anyway, my reason for a single-cycle processor was simplicity. I wanted 16-bit instructions so the last byte in an instruction can be used to define an address or number for use with the main function (i.e., ADD A,87 or LD ($80),A). The 8-bit address bus was used because I'd have to have a 16-bit ALU and 24-bit instructions if I went with a 16-bit address bus. Originally, I was going to shoot for all sorts of logic operations in the ALU, but I toned it down for simplicity. I needed to have separate data RAM and instruction EEPROM to allow the processor to be a single-cycle processor. The low amount of registers was to keep it simple, I didn't want to have a register file system. I wanted only output pins, again, for simplicity. I opted for the wire-wrapped CMOS chips over FPGA because something about electronics that's always interested me is really knowing what's going on in a circuit at the logic gate level, and the epitome of a modern circuit is a processor. And for the last point in the overview, I went with an 8-bit parallel ALU because, after drawing up a basic sketch of my proposed ALU, I learned that an 8-bit ALU requires 30 leads to interface with the other parts of the processor (19 inputs, 11 outputs). I figured that was enough, and this is my first processor. It doesn't need large abilities.

After figuring all of this out, I moved on to a basic block diagram, but all fruits of those labors were discarded because I realized I'd need to create an ISA (Instruction Set Architecture) before I get into the block diagram. I struggled with the ISA for an hour or so before deciding it was a good idea to go to sleep.

Saturday, December 20, 2008

I spent all of today's work on the OZ-1 designing an ISA and writing the machine code. After a few hours of on-and-off struggle, I finally came up with an ISA that I'm happy with, proud of, and fits in 2 bytes. Here's the diagram I drew up:



It definitely looks a little nuts, almost every different type of instruction uses different bits in different ways. I did it this way so I could keep the instructions within two bytes. The green sections are for options, and the top half of green sections means, if both bits are 1s, then that option is set. The red sections mean if you're using that type of instruction, the bits that are red must be the 1 or 0 indicated, or the instruction is invalid. Purple is for ALU control.

There's a lot to say about the ISA, I designed it, again, around simplicity. I wanted almost every type of command to be able to have a condition without having a jump to a separate label for them to execute, like many (if not, all) popular assembly language variants. Essentially, this means that you can add two numbers if the Z (zero) flag is set, or the NZ (not zero) flag is set, or do any other command conditionally in that manner, excluding INC A and DEC A. The syntax for the adding example might be: ADD Z,A,B. This means, if the Z flag is set, add registers A and B. It's the same way for the NZ flag. If you want to turn on output pin 5, but only if the NZ flag is set, you'd write: PON NZ,5. So, with this system, the OZ-1 will have 12 main instructions, but a total of 53 instructions with various combinations of Z and NZ and a few other things.

The OZ-1 won't have a direct instruction to tell it to compare register A with a certain number, but rather, the CP command in an assembly code file will be converted to a SUB A,# instruction when assembled. Just thought that might be good to note.

Oh, one last thing, for the ALU, it has three input control leads, which are SUB/ADD, C0 and C1. Here's a truth table for the settings:

SUB/ADD C0 C1 Setting

0 0 0 add

1 0 0 sub

X 0 1 AND

X 1 0 OR

X 1 1 XOR

Anyway, that's all for today. Signing out.

Sunday, December 21, 2008

Alrighty, I was mulling over a few issues after I woke up today. The first was, how on earth am I going to get a 256-byte EEPROM chip? And not only that, it has to be parallel-access. The other was how I'd go about getting a full 2 bytes off of the instruction EEPROM every cycle when it only allows access to one byte at a time. Well, as for the first problem, I posted a question on an electronics forum, DutchForce. For the second one, I decided there were a few ways to go about it, and the simplest way to do it was to have two separate instruction EEPROM modules. The program counter would address them with the same address, so a full instruction might be byte 157 on the first chip and byte 157 on the second chip. I'd load both into a 16-bit instruction register at the same time. The first chip would hold the main chunk of the instructions, the 8-bit opcode. The second chip would hold all of the peripheral numbers and addresses that the instructions would use. I felt this was the best way to go about it, and it doubled my capacity for instructions! Before, the processor would only be able to hold 128 instructions, but now it'll be able to hold a full 256. Like I said before, I don't plan to have this processor do a whole lot. I do plan, however, to build a 16-bit OZ-2. I'm planning to start work on an instruction decoder and rough block diagram today, but seeing as it is December, and it just snowed 8 inches, the driveway calls.

I finalized a design of an instruction decoder later today. I figured the best way to go about it was to have the instruction decoder set certain outputs on the decoder to high depending on what the instruction is telling the processor to do. Then a general 8-bit data bus that connects to every part of the processor that uses data would be controlled with multiplexers using those outputs. If the instruction doesn't use a number or address, the data bus would be disconnected right at the instruction decoder. I came up with this design after having designed an instruction decoder that set the certain pins to high, but had three data bus outputs: one to register A for load instructions, one to RAM for addresses, and one to the ALU. This didn't really work out. The block diagram was rather confusing and inefficient. There were paths crossing each other and general clutter, so I figured there had to be a better way. With the new instruction decoder designed, I plan to draw up a new block diagram.

Also, I've been toiling over how to best do jump instructions if they use a condition that isn't true. Like, ADD Z,A,B, but the Z flag isn't set. Right now, I have it set up so if the condition isn't true, the data bus gets cut off, and the RAM and registers disabled, then the program counter gets advanced one instruction. Essentially, it puts the processor on lockdown until the next cycle. As far as the jump block, I thought about it just setting the program counter to a number specified in the instruction instead of its normal operation of being increased by 1 every cycle.

For instructions that turn output pins on or off, I have a "PON/POFF decoder" block that's hooked up to the data bus and two control lines that are connected straight to the instruction decoder. The pin decoder block takes in a number that's only the first four bits of the data bus, seeing as there are only 16 output pins that I plan to have. If the data bus is carrying a number over 15, nothing will return an error, but the decoder will just ignore the last four bits and the desired result may be different from the actual result.

Monday, December 22, 2008

Well, as I said yesterday, I plan to draw up a new block diagram, which will probably end up looking more like a wiring diagram.

So far today, I've finally settled on a block diagram that I plan to copy on to a computer in something like MS Paint. It's a very simple diagram, hardly any details. There's nothing about control lines for the ALU or any of the other blocks. I'm going to make a new block diagram that'll be much more detailed on the computer, doing that on paper is a bit nuts.

Anyway, about the block diagram I drew up. The general data bus idea has a few exceptions to the rule. First of all, it connects to the ALU so a number can be added to register A, but A is also connected to the data bus, so A needed its own dedicated line to the ALU's other input. Also, many things have several connections to the data bus for different purposes. Mainly, these are separate in/out lines so things like registers A and B can have simpler logic to interface with the rest of the processor. Another thing to note, there's a RAM bus as well that carries data from the RAM to register A and the ALU. One more thing that's prominent about this block diagram is that there's only one visible multiplexer, and that's for the second input on the ALU so it can do things like add a number from RAM or from register B, or just a plain number straight off the data bus. Much of the control logic for the data bus will be done on a relatively internal level inside each block. In reality, there are going to be dozens of lines (actually, just more than two dozen) streaming out of the instruction decoder to tell each area of the processor whether it should be connected or disconnected from the data bus, and which task the ALU should be performing, as well as where it should be getting its data from.

There's also going to be pretty much one line for each block on the diagram coming from the condition block. Remember how I said above that the system I thought of for conditional statements was to put the processor on lockdown until the next cycle? Well, that's what all of those lines would be for. The way I think of it is that each block needs permission from the condition block and instruction decoder to gain access to the data bus. And if the condition block says no, it says no to the whole processor. I'm going to need to figure out some sort of reset logic for the condition block, though, so after the false instruction has passed, the processor continues as it would. Otherwise, Z might not be set, and the whole processor just stops.

So that's where I am as of now. In the future, I'm shooting for designs of the internals of each block, including control inputs from the condition block and instruction decoder and inputs and outputs for the data bus. Before doing that, though, I'm going to make the more extensive version of the block diagram, which will basically be a segmented wiring diagram.

Thursday, January 8, 2009

Through Christmas break, I've been working on internal diagrams of each of the blocks in the block diagram, and I made it through all of them except the condition block, which is fairly simple, I just need to figure out how many lines go where to block everything from the data bus. Then some control logic to lock up the processor when only certain conditions are true. It's just a matter of actually drawing the diagram, which I will some time tonight. Unfortunately, school hit me like a ton of bricks, so I haven't had much time since last Saturday to work on anything that isn't related to school.

Anyway, I had a bit of a breakthrough with single-cycle processor architecture. The new architecture I thought up is based largely off of the OZ-1's architecture, but rearranged to produce a much more efficient system. My idea is to take the data bus arrangement and streamline it. At the top of the "pipe", so to speak, would be 8 bits from the instruction register going to the instruction decoder, and 8 bits into the data bus. The instruction decoder would then control the branches the data bus would take, guiding it to where it's needed. This sounds a lot like the OZ-1's architecture, and the details haven't been worked out yet, but I would design it so the operation that would take the longest would be at the start of the "pipe", and the one that would take the least amount of time would be at the end, so less time can be spent sending data to the longest operation. This would cut down on minimum cycle time. The various operations and related blocks would be organized into groups, i.e., if it's a register operation, branch here, then if it uses register A, branch to here, as opposed to there. I just thought it up on the spot, but I think what I'm going to do is keep everything else, like the instruction decoder and the majority of the blocks I’ve designed, and the ISA of the OZ-1, then just reshape its architecture. This would require a new block diagram, and a lot more thinking, but I want to make a processor that's actually capable of real tasks. After designing this processor as an 8-bit processor, I'm not going to build it, or even get up to arranging an actual circuit diagram, but I'm going to redesign it as a 16-bit processor. Also, I'll add input lines in addition to the already-existing output lines. It's basically going to get a rather beefy upgrade. Also, I'll add more programmer-visible registers, including C and D registers, and organize them in a register file for easier access. As for now, though, much work remains to be done on the 8-bit version of this architecture. All of my current plans are on paper, drawn in pencil, and I plan to keep them, but they won't be of much use any more, at least the ones pertaining to physical arrangement and basic architecture.

Saturday, January 17, 2009

So, mainly I've been working on internal block diagrams for the more extensive block diagram, so to speak. I haven't gotten much further on the new architecture I was hoping to work on; we just finished midterms at school. Anyway, I jotted down a register file system in my notebook, not much else.

One major thing that has happened, though, is I found an amazing program called Logisim. It's a logic circuit simulator, and it's fantastic. It's very extensive, and very easy to use. Most other logic circuit simulators I've come by have been choppy, unpolished, and difficult to use. This one is very intuitive, kind of like how a logic circuit simulator should be. Anyway, moving away from a potential commercial for the program, the most useful thing I've found it does, to me, at least, is let you create what're called subcircuits. For example, I drew up my instruction decoder and debugged it in a circuit called "Instruction Decoder". Then, in the main circuit for the OZ-1 project, I can create an instance of that circuit in block form. It labels all of the inputs and outputs on the block according to their placement and associated label in the subcircuit. It's simply amazing.

I decided to draw up a diagram for an 8-bit register. The program has built-in ones, but I wanted to try it for myself. I made a subcircuit of an edge-triggered D flip-flop, then made 8 of those subcircuits in the register circuit, and wired up some AND gates to the outputs for a "read" input. The write input was hooked up to the enable lines on the flip-flops. I tried out this system, and it didn't quite work. I told the instruction decoder to load register A with a specified number, then wired it up, but unfortunately, race conditions due to the edge-triggered flip-flop made it unfeasible for the register to take input while the enable line is going from low to high. To solve this, I replaced all of the edge-triggered D flip-flops with D latches. D latches aren't edge-triggered. While the enable line is high, their outputs can change at any time according to their inputs. I found this to be the optimal solution, seeing as it wouldn't create any sort of race condition. This worked out great, when I started the simulation the register was loaded with what was on the data bus.

After that, I wanted to try out creating the ALU. Everything in the ALU I drew up myself. I enjoy working with the very basis of circuits; I'm not really a fan of just pretending some sort of magic goes on behind the scenes to make it work. So, I drew up a few subcircuits: a 1x2 multiplexer, a 2x4 multiplexer, and a full adder. The ALU was just a matter of wiring these up correctly. I gotta say, the diagram is a little intimidating because everything in the ALU is in bus format (meaning the data comes in 8-byte chunks, so 8 separate wires are necessary to get the job done correctly). The ALU worked beautifully. I hooked up the ALU to the unofficial data bus and the correct lines on the instruction decoder. I then reset the simulation and started it with the LD A,# instruction, then stopped it and set the instruction to an ADD A,# instruction. It worked perfectly, when the ADD A,# instruction came on, the enable line to the register turned off and the read line turned on, feeding into the ALU. The data bus didn't affect register A because the write line was low, and instead went to the other input on the ALU. The output lines on the ALU read 116. This was great! Register A was holding 110, and the data bus was set to 6. The only thing I didn't do was have the ALU's output be stored in A. This was because I had only drawn up the register, not register A's diagram. There's a difference because register A's diagram involves 3 separate 8-bit inputs that go through a 24 to 8 multiplexer into register A, then the register's output gets demultiplexed into 3 separate 8-bit outputs again. Without the 3 separate inputs, there's no way for the register to differentiate between input from the data bus and input from the ALU.

So, as the detail of my design increases, I've been trying to work out how to take care of race conditions within the processor. When you think about it, if I add register A and a number, then store the result in register A all in one cycle, that's probably going to cause issues. This would mean that the read and write inputs would be enabled, essentially turning register A into a buffer. Questions then arise, such as, would register A's value be changed to the ALU's output before the operation was even complete? Would a loop be created, running the addition operation until the next cycle, instead of just once? What would happen, and how would I go about avoiding it? These are pretty fundamental questions and problems regarding a single-cycle processor.

Monday, January 26, 2009

So, a lot has happened since I last updated this log. Much progress has been made using Logisim. The current state of the processor is: registers A and B and their instructions fully operational, and output pins and RAM pending. Also, everything conditional is working, and the jump instructions work. I have since changed the ISA to exclude INC and DEC instructions, as I had the revelation that they were simply ADD or SUB instructions and purely based on software/programming. I’ll need to run back through and go over the various issues I ran into in the last 9 days.

So, the first thing I did was wire up register A and the ALU. Many of the ensuing problems were caused by unfamiliarity with Logisim. I switched from designing my own registers to using Logisim’s built-in registers, which weren’t plagued by errors due to race conditions within the flip-flops. Then, I had to make them write on the falling-edge so that at the beginning of every cycle that involved writing, the values were set up to write, then when the CLK signal fell, they would write and that would be the end of the cycle. This solved race conditions with writing to registers and various other related issues, mainly ALU operations and storing their results in register A.

Then, I tried clocking the ALU by adding registers to the inputs and outputs and writing them on the rising edge, but that simply didn’t work out. There wasn’t much of a point in clocking the ALU anyway because everything else was clocked.

After getting the ALU and register A set up, I made a new jump system for controlling the program counter. Basically, I lined up 8 full adders with the current program counter value on one input and 1 in the other. The result of this was multiplexed with what was on the data bus, and depending on whether or not the instruction was a jump instruction, one of the results would get through to the program counter, which wrote on the rising edge.

Those were the main issues I came across, and they caused things like register A magically being loaded with whatever was supposed to be on the data bus right at the beginning of the instruction (which was really, really weird). Also, register A getting added to twice in one instruction, instead of just once, causing data corruption. One of the most helpful tools in solving all of this was the logging capabilities of Logisim. I could log the various values of certain inputs and outputs as they changed with the clock signal to see what was going on.

So, at this point I could load numbers in to register A, add to and subtract from register A, jump to various addresses in the instruction ROM, and do everything conditionally. The next step was to add register B, which would greatly expand on the processor’s capabilities.

This went much smoother than expected. Fortunately, the system I had set up with the instruction decoder allowed me to basically just hook up all of the wires and run a simple test program. I was quite pleased. One thing I noted that had to be done was to add controlled buffers on the data bus in various places. This was to protect outputs from being inputted to, like when register B is loaded from register A, which uses the data bus as a way to connect the two. The next step was to connect register B to the ALU, which required multiplexing the data bus and register B through to the second input of the ALU. I designed an 8-bit 3 to 1 multiplexer, which wasn’t exactly a multiplexer. Instead of having control lines that would be decoded internally, they were direct. This means that three data inputs would have 3 control lines. If none of the lines are on, no data gets through. Two control lines being on would cause data corruption. Only one control line is supposed to be on at a time. I found that this was the best way to handle the multiplexing using the signifier system that I set up for the instruction decoder. In short, various signifiers are turned on to tell the processor what the instruction means. This shortens the instruction length, which, in this system, is pretty much the instruction decoder’s sole purpose.

I did catch a few snags going in to this. First of all, register A was outputting to the data bus through a controlled buffer that originally only turned on if the instruction didn’t use the data bus, meaning only registers A and B are in use. This cause an error because if I tried to load A with B, A would try to output to B while B is attempting the same for A. To fix this, I used an AND gate and made sure the buffer only turned on if the data bus wasn’t being used and register B was being loaded from register A. The OZ-1 will only be able to load register B through register A, and only be able to use RAM through register A. I realize that these are large limitations on a processor’s functionality, but I will update and fix these limitations when I design the 16-bit OZ-2. I plan to make the OZ-2 many times more versatile.

Anyway, that was a good summation of the last 9 days of work on the OZ-1. I’m very close to being done with the design in the virtual world, and plan to stop there with the OZ-1 when it’s done. For the OZ-2, I will be building it.

So, enough about the past work on the OZ-1, and on with a little talk about the work I plan to be doing soon. The last two things left in the processor are the input/output pins and access to RAM. I plan to do the pins first. My idea for the pins is not so much and individual control, but treating them as more of an output port. They would be connected to an 8-bit register (so, 8 output pins), and writing the data bus to that register only when the instruction is an output instruction. The pins wouldn’t reset themselves after the cycle; they’d stay on until the program turned them off. I’m not sure what I’d do about input pins, but I have an idea. The system would work kind of like the condition block. Basically, the pins would input to a block that would have 8 other inputs to tell it which to test. If the program is testing pin 4, and pin 4 is on, the block would output a 1. If pin 4 is off, it would output a 0. This would then communicate with the rest of the processor. I don’t plan to add input pins, though, to the OZ-1. It would require a restructuring of the ISA.

Well, today was certainly a good day. I finished the OZ-1. The output pin system works beautifully, although my one regret about it is that you can’t set output pins based on things like register A. It can only be through a fixed number in the instruction that goes to the data bus. The RAM turned out quite well after fixing one minor wiring error. It can be written to and read from, and numbers can be added from it to register A.

Having finished the processor, the next few days will be spent finalizing the ISA and a block diagram, as well as other diagrams. The nomenclature of the diagram in Logisim will be checked and made consistent. Then I’ll take screenshots of the entire processor and each part for future use and documentation.

I figure that here is where I should discuss the project, its benefits, strengths and weaknesses of the processor, etc. I started the OZ-1 processor 38 days ago with a rough outline of the future processor’s statistics and a Wikibook of microprocessor design experience and some basic TI-83+ ASM under my belt. I knew that I wouldn’t be making anything special, and that I’d be making another, but that an 8-bit processor was a good place to start. That’s what Intel started out with, isn’t it? Anyway, I thought this project was great for me. It was fun, and I learned a ton about digital circuitry and logic, processors, and how computers work. I feel much more experienced in the field of digital circuitry now that I’ve finished this project. I’m proud of myself because I came up with 95% of the fundamental concepts behind the OZ-1. While they may not have been original, the fact still remains that I thought of them at some point. I’ll try the same for my next processor, a 16-bit OZ-2. It will be much more versatile and useful. I plan to actually build the OZ-2 out of CMOS logic chips, as I had planned for the OZ-1. I’ll write more about the OZ-2 later.

The OZ-1 certainly is not a very useful processor. I’m going to be frank about it. It only has 2 registers, a very low capacity to keep their values around unless the programmer is very careful (or me, who designed the thing), can hold 256 instructions, and has 256 bytes of RAM. All in all, not very special. Also, it’s a single-cycle processor, which limits its speed, but I found that making it a single-cycle processor was a bit of a challenge in and of itself. I had to figure out how to divide the cycle into two parts, the high and the low clock cycle halves, and how to make everything work off of those two parts in one cycle. Also, the OZ-1 doesn’t even have input pins so other things can communicate with it. I suppose the program counter could be run to two blocks of RAM in a computer, and the instructions run off of there, but still, it would not be very useful or change a whole lot. The output pin system is extremely limited, it can only output based on fixed numbers on the data bus from the instruction. It would be rather easy in concept to fix this and make it more versatile, but in practice, I would have to rewire pretty much everything. The wires are packed pretty tight in Logisim. Also, the ALU is limited to only run operations on register A and one of three other options: a number off of RAM, register B, or the data bus. While it may seem alright, the processor is unable to add register B and the data bus, or a number off of RAM and the data bus. Similar to that, the RAM can’t be loaded with register B, or even something off the data bus. Everything would take twice as many cycles to complete than it should, and register A is almost guaranteed to be destroyed in the process. All of these limitations I plan to correct in the OZ-2.

About the OZ-2, as stated above, it will be 16-bit. That means, it will be able to hold 64 thousand instructions, have 64 kilobytes of RAM, and be able to run operations on numbers up to 65,356.

A big thing about the OZ-2 is that it’ll be many times more versatile than the OZ-1. The OZ-1 could only use specific things for specific operations, whereas the OZ-2 will be able to use almost any medium for holding a number to run any operations that involves numbers.

Also notable will be the OZ-2’s register file of 4 registers: A, B, C, and D. They will still be 8-bit registers, but I hope to be able to implement double-registers, like AB and CD in order to make it easier to run operations on 16-bit numbers. If that doesn’t work out, they’ll end up being 16-bit registers, which seems much more practical in the first place. The flags register will hold more than three flags. I will model the flags off of the flags in the TI-83+ calculator.

The ALU will be more complex by being able to run shift operations. I suspect the instructions may end up being anywhere from 24 to 32 bits long to accommodate the relative complexity of the OZ-2. Oh! I almost forgot. The OZ-2 will have input pins as well as output pins. Two bytes of input pins will be a dedicated data port, as well as two bytes of output pins as a dedicated data port. In addition to those, there will be 8 individually controllable output pins and 8 individually visible input pins. I think that’s about it for the OZ-2. I hope to start work on it from scratch, using the OZ-1 as guidance, after I’m done finalizing the diagrams of the OZ-1.