Tick: A Simple Reversible Processor
6.371 Project Report

Michael Frank
Scott Rixner

May 12, 1996

1 Introduction and Motivation

The Reversible Computing Project in the MIT AI Lab aims to build a CPU (called *Pendulum*) that is *reversible*, meaning that the processor does not discard any information during its execution, and can be run in reverse.

The goal of the eventual Pendulum chip is to use its reversibility to perform adiabatic charge recovery, and thus greatly decrease the amount of energy that is dissipated during its operation. But real charge recovery requires non-standard circuit techniques, special power supplies, and additional circuit complexity which would put it beyond the scope of a 6.371 project.

The Pendulum's ISA and block-level datapath was designed and simulated at the block-functionality level earlier by another grad student, Carlin Vieri. See http://www.ai.mit.edu/~cvieri/reversible.html for more information about the original Pendulum and about the Reversible Computing Project.

For this class project, we decided to design a simplified version of Pendulum, called *Tick*, using standard CMOS circuit techniques rather than charge-recovery logic. This allows the design to be greatly simplified. But the Tick chip should still be useful as a baseline for comparison with an energy-saving version to be designed later. The Tick chip is capable of actually reversing its execution, so that a program can be run forwards and then backwards to return to its initial state, actively demonstrating the fact of the instruction set's reversibility.

We should be clear that Tick is not actually reversible at the circuit level; we throw information away all over the place internally due to our use of standard CMOS logic circuits. But a legal Tick program will conserve all programmer-visible information.

In order for Tick to be comparable to and reusable in our group's later processor work, we designed Tick for the same "HP14" half-micron MOSIS process that our group will probably use for the actual Pendulum processor. HP14 is made for a 3.3V power supply and has three levels of metal. We used Cadence software available at our lab to perform the design work and simulations.

2 Task Specification

Our first job was to define Tick's functionality by scaling down the Pendulum ISA specification to fall within the scope of what we might accomplish. Pendulum is a 32-bit RISC-style processor with 32 registers and a simple MIPS-like instruction set.

We decided to make Tick have 16-bit instructions and 8-bit wide data, because 16-bit instructions allowed us to maintain most of the functionality of the original Pendulum instruction set, by just reducing the number of registers to 8, and decreasing the size of the various immediate and offset fields. The 8-bit wide paths for addresses and data allow room to actually encode small but interesting programs. (A smaller word size, such as 4 bits, would only have allowed for 16-instruction programs, which can't do very much.)

R-type	`opcode`	`rz`	`ra`	`rb`	`0`
	6 bits	3 bits	3 bits	3 bits	1 bit
I-type	`opcode`	`rz`	`ra`	`immediate`	
	6 bits	3 bits	3 bits	4 bits	
B-type	`opcode`	`rz`	`ra`	`offset`	
	6 bits	3 bits	3 bits	4 bits	
J-type	`opcode`	`0`	`0`	`address`	
	6 bits	2 bits		8 bits	

Table 1: Tick Instruction Formats

	`0`	`do/undo`	`function`			`1`	Register ops w. immediate equivalents
bit	5	4	3	2	1	0	
	`0`	`do/undo`	`function`			`0`	Register ops w/o immed. equivs.
bit	5	4	3	2	1	0	
	`1`	`do/undo`	`function`			`1`	Immediate ops.
bit	5	4	3	2	1	0	
	`1`	`fwd/rev`	`branch type`			`0`	Branch/jump instrs.
bit	5	4	3	2	1	0	
	`1`	`fwd/rev`	`1`	`special`		`1`	Special instrs.
bit	5	4	3	2	1	0	

Table 2: Opcode Formats

2.1 The Tick Instruction Set Architecture

Instructions come in several formats, shown in table 1. The most significant 6 bits of each instruction contain the instruction opcode. R-type instructions perform register-to-register operations. I-type instructions perform register operations with an immediate operand. B-type instructions perform conditional relative branches, branch-and-link, and jump-register. J-type instructions perform absolute jumps.

The RZ field generally contains the index of the destination register for the operation, and RA and RB refer to the source registers. (RZ is required by some instructions to be the same as RA in order to preserve reversibility.)

Opcodes themselves have a regular format to make instruction decoding earlier. (See table 2.) Bit 5 is always used to indicate the direction of the instruction for instructions that come in both forwards and reverse versions. The function code for a given operation is the same for both its immediate and register-to-register forms. There are more details of the function and branch-type encodings that were also designed to make decoding easier, but for conciseness we will not go into those here.

Table 3 shows the complete Tick instruction set, together with descriptions of the operations performed when instructions are executed in the forward direction. The last column shows a precondition that must be satisfied to ensure reversibility. Additionally, reversibility generally requires that the RZ register be a distinct register from both RA and RB, unless stated otherwise.

Tick does not actually enforce these preconditions; however, they are still important for us because they often affect the behavior of the instruction when it is run in reverse. For example, when run forward, UAND (undo AND) just clears Z, but when run in reverse it sets Z to the bitwise AND of A and B.

The reverse branch and jump instructions (prefixed by R) require more explanation than would fit in the table. When the processor is running in reverse, the reverse branch and jump instructions control the flow of execution. To ensure reversibility, each forward branch or jump instruction must be paired with a corresponding reverse branch or jump instruction (and vice-versa) in such a way that the previous PC value can always be determined from its current value. Ensuring that this condition holds is the job of the programmer or compiler of Tick programs.

2.2 External Interface Specification

With the instruction set outlined above, the next part of the task specification consists of the pin interface between the Tick processor and the outside world.

Inputs to Tick include power (3.3V) and ground, a single externally-generated clock signal, an active-low "reset" signal, and a "direction" input, which goes high to indicate that the processor should run in reverse rather than forwards.

To make the project more feasible, we decided that the Tick CPU chip should have its memory handled externally, rather than including it on-chip (even though it is only 256 words). This has the added benefit of allowing the chip to be tested by loading inputs and outputs into memory using external circuitry that can be built later.

Tick uses separate instruction and data memories. Addresses to each are 8 bits wide. Data words are 16 bits wide for the instruction memory, and 8 bits wide for the data memory. There are separate read and write buses to data memory, since the interaction with data memory is defined as being a simultaneous read and write.

Tick produces an "Iread" signal that tells the instruction memory to fetch an instruction from the given address when Iread is high during a rising clock edge. There is no write to instruction memory.

Tick also produces an "exchange" signal that, when high during a rising clock edge, tells the data memory to read out the data value at the given address, while simultaneously writing a new value into that same address. The external memory unit is expected to perform the latching necessary to keep the old data value active on the read lines during the entire cycle, even after the old value has been overwritten by the new value in the memory itself.

Instruction				Opcode	Result	Precondition
XOR	rz	ra	rb	000001	rz= ra XOR rb.	rz is ra
ADD	rz	ra	rb	000011	rz= ra + rb (mod 256).	rz is ra
AND	rz	ra	rb	000101	rz= ra AND rb.	rz=0
OR	rz	ra	rb	000111	rz= ra OR rb.	rz=0
SLT	rz	ra	rb	001001	rz= (ra < rb).	rz=0
UAND	rz	ra	rb	010101	rz= 0.	rz= ra AND rb
UOR	rz	ra	rb	010111	rz= 0.	rz= ra OR rb
USLT	rz	ra	rb	011001	rz= 0.	rz= (ra < rb)
NEG	rz	ra		000000	rz= -ra.	rz is ra
RLV	rz	ra	rb	000010	rz= ra, rot. left rb.	rz is ra
NOR	rz	ra	rb	000110	rz= ra NOR rb	rz=0
SRLV	rz	ra	rb	001000	rz= ra >> rb	rz=0
SUB	rz	ra	rb	001011	rz= ra - rb (mod 256)	rz is ra
SLLV	rz	ra	rb	001100	rz= ra << rb	rz=0
SRAV	rz	ra	rb	001110	rz= ra/2	rz=0
UNOR	rz	ra	rb	010110	rz= 0.	rz= ra NOR rb
USRLV	rz	ra	rb	011000	rz= 0.	rz= ra>>rb
USLLV	rz	ra	rb	011100	rz= 0.	rz= ra<<rb
USRAV	rz	ra	rb	011110	rz= 0.	rz= ra/2
XORI	rz	ra	immed	100001	rz= ra XOR immed	rz is ra
ADDI	rz	ra	immed	100011	rz= ra + immed mod 256	rz is ra
ANDI	rz	ra	immed	100101	rz= ra AND immed	rz=0
ORI	rz	ra	immed	100111	rz= ra OR immed	rz=0
SLTI	rz	ra	immed	101001	rz= (ra < immed)	rz=0
UANDI	rz	ra	immed	110101	rz= 0	rz= ra AND immed
UORI	rz	ra	immed	110111	rz= 0	rz= ra OR immed
USLTI	rz	ra	immed	111001	rz= 0	rz = (ra < immed)
BEQ	rz	ra	offset	100000	PC+=offset if rz=ra	
BNE	rz	ra	offset	100010	PC+=offset if rz!=ra	
BLTZ		ra	offset	100100	PC+=offset if ra<0	
BGEZ		ra	offset	100110	PC+=offset if ra>=0	
JR	rz			101000	PC = rz, rz = 0	
BAL	rz		offset	101010	rz = PC, PC+=offset	rz = 0
J			addr	101100	PC = addr	
RBEQ	rz	ra	offset	110000	(See text.)	
RBNE	rz	ra	offset	110010		
RBLTZ		ra	offset	110100		
RBGEZ	rz	ra	offset	110110		
RJR	rz			111000		
RBAL	rz		offset	111010		
RJ			addr	111100		
START				111011		
FINISH				101011	Do not increment PC.	
EXCH	rz	ra		111111	rz < − > MEM[ra]	

Table 3: Tick Instruction Set

That concludes the description of Tick's external interface. The focus of our project was functionality rather than speed, so meeting a particular target clock rate is not part of the specification. In any case, we plan to run at least at slow as memory.

3 Organization

Figure 1 shows a block diagram of the Tick datapath, with the various control lines omitted for clarity. The primary elements are the controller unit, the register file, the ALU (arithmetic-logic unit), and the PC register. The PC updating path also includes an adder for branches, an incrementer/decrementer for normal forward/reverse PC motion, and a PC controller unit that determines whether branches are taken and controls the multiplexer that feeds into the PC register.

Tick's datapath is really only a trimmed-down version of the original Pendulum datapath (see figure 2). A number of simplifications were possible due to the fact that Tick's components are not actually required to perform charge recovery. Additionally, we are not concerned with pipelining as Carlin was in his original design. So, for example, Carlin's ALU was required to have three writeback paths to the register file in order for the flow of information through the ALU to be pipelined without letting information be thrown away. We, however, just destructively modify the ALU inputs on each cycle, so the only output we need from the ALU is the actual result to go back into the Z register.

4 Detailed Discussion of Design

4.1 The Design Process

After defining the task specification in the form of the instruction set, the first step in the actual chip design process was to get the entire block-level diagram for Tick to simulate correctly using functional descriptions of each block's behavior (written in the Verilog language). This allowed us to quickly ensure that the overall architecture and datapath was consistent before beginning detailed schematics. It also provided a framework within which the transistor-level schematics for the individual components could be tested in the context of the rest of the processor's already-verified correct functionality. This eliminated the need to come up with custom test vectors for each individual component; instead we just ran test programs on the full simulated CPU as each new transistor-level schematic was designed and substituted for a component's functional description. This worked quite well.

Unfortunately, getting the functional simulation working took longer than expected (about a week), largely because we were still learning how to use the Cadence design and simulation tools and the Verilog functional specification language. Nevertheless, we believe it was a valuable step, because it enabled us to work the kinks out of the architecture very early, before doing a lot of time-consuming wiring that might have had to be redone. During this functional verification process, we made many incremental simplifications of the original Pendulum block diagram, thereby avoiding a lot of extra wiring work we might have done had we instead started right away on schematics.

After getting the functional design working, the next stage was to fill in the transistor-level schematics for each component. To save time, we adopted a strategy of defining standard cells for logic gates, flip-flops, MUXes, etc., and using those everywhere we could, as opposed to having any transistor-level detail at the top level of each component.

Additionally, after building the schematics for each component, we ran SPICE simulations in order to get a rough idea of the timing requirements and make sure that the various delays were acceptable. Even though speed was not our main concern for this project, it was important to measure delays because of our clocking scheme, which was based on a single-phase clock controlling various rising-edge-triggered Master-Slave flip-flops. If contamination delays through the circuitry were too small, then the output from one flip-flop might reach the next flip-flop ahead of the clock signal, and so the input to the second flip-flop might change before the hold time had passed, thus corrupting its output. Fortunately, we detected no such problems in our designs. The schematic design stage took about a week and a half.

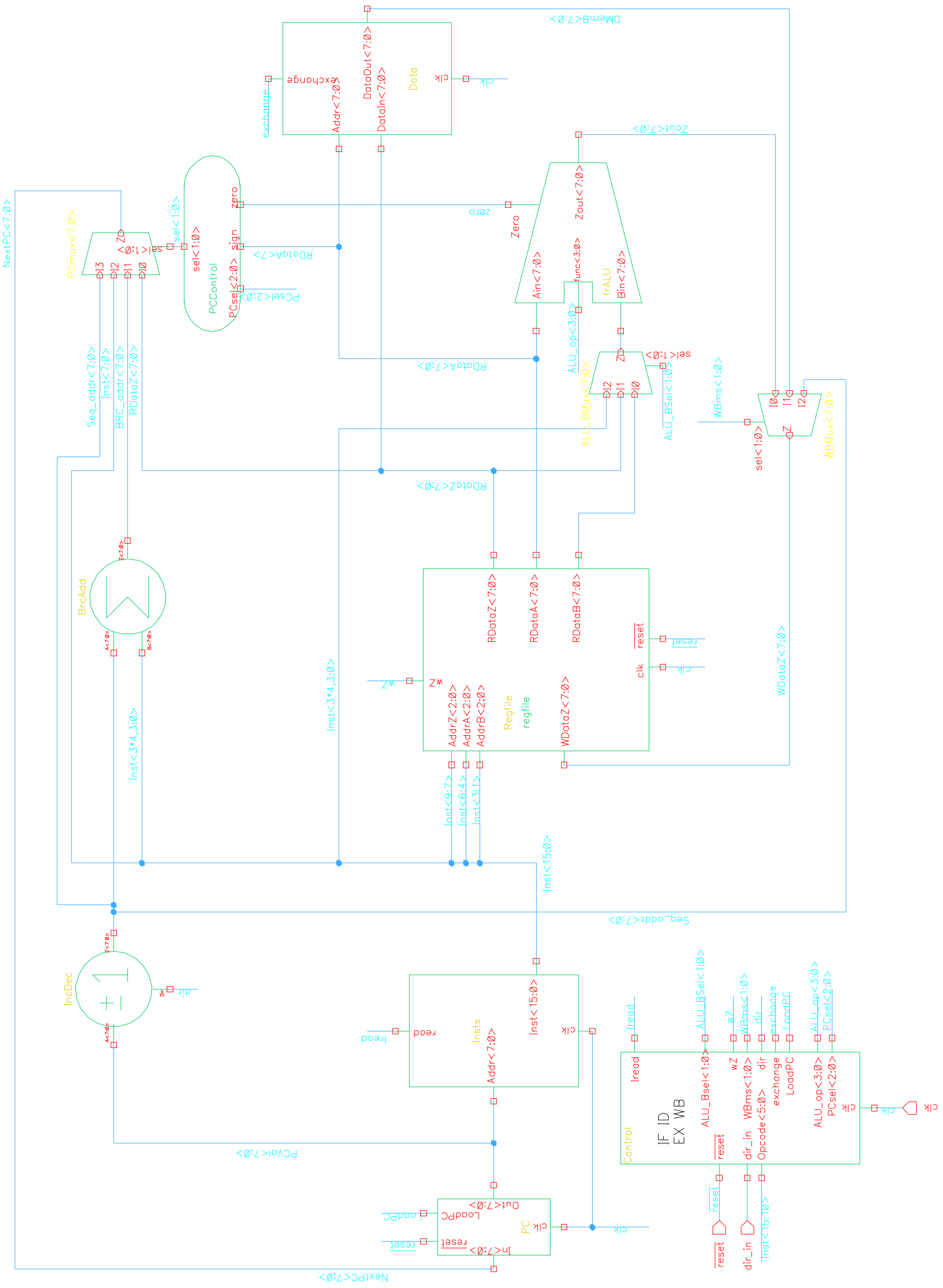


Figure 1: Schematic diagram of the Tick datapath. (Control lines are disconnected for clarity.)

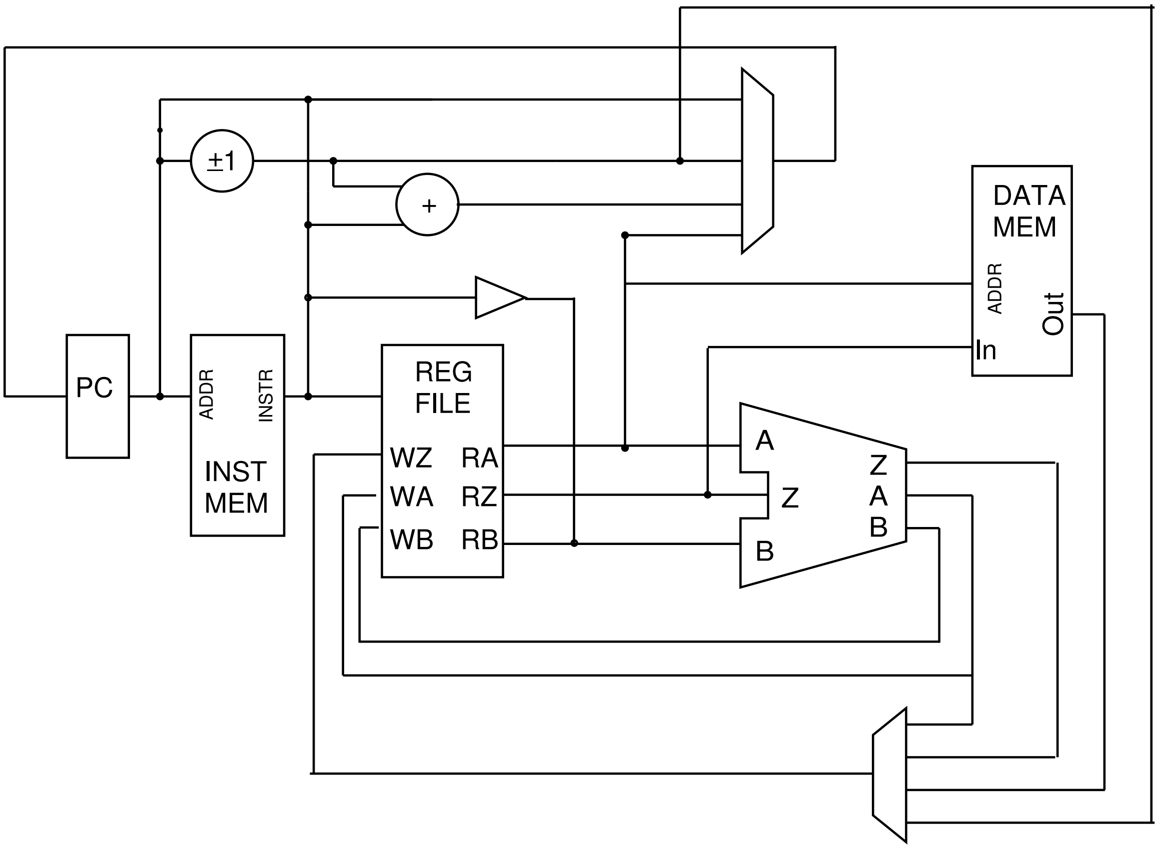


Figure 2: Simplified schematic of the datapath for the original Pendulum (courtesy Carlin Vieri). The controller and control lines are not shown.

The final stage was layout. Layouts were not simulated directly, but rather checked by comparison against the schematics using a layout-versus-schematic (LVS) checker. Unfortunately, this strategy had some problems. The first was that often we would make layouts that were equivalent to the schematics at the logical level, but different at the circuit level. For example, the inputs to the 6-transistor fully-restoring static CMOS XOR gate are asymmetric, so if we connected the inputs differently in the layout and the schematic, this would cause an LVS mismatch between them. So, exact circuit-level correspondence was really a more conservative check than we needed, given that we were using only standard static CMOS gates. A better approach might have been to simulate the netlist derived from the layout and just make sure the results were correct. Many of our LVS errors were due to such trivial mismatches rather than to actual logical flaws in the layouts. However, we did not have time to find out if there was another way to verify our layouts, so we just stuck with the simple LVS approach.

It would also have been nice to have been able to do some simulations that took into account the capacitances and resistances of the actual wires we laid down during layout. The tool was probably capable of this, but again, we did not have time to look into how to do it. However, we were careful to avoid features such as long polysilicon lines in our layout, so we do not anticipate that the wiring would have caused any great problems for us. And again, we were not concerned with speed in this design—our primary constraint was rather just to get the raw functionality working within the limited time frame of the project. In retrospect, the choice of project was really a bit too large for two people to do a very refined job on it given the limited time available. However, we could not really have known in advance how long it would take, given that this was our first chip. Let us just say we have a newfound respect for the people who design complex real microprocessors with the additional constraints of trying to make them very fast and very small.

4.1.1 A Major Problem

We did, however, encounter one major problem resulting from our attempt to work within a fabrication process that was relatively novel and untested (at least from the point of view of students in our lab). Namely, we found out at the last minute (earlier today) that the design rule checker we have been using did not quite check all of the rules. Namely, it did not check for minimum separation between body plugs and diffusion in the case when the two were not actually abutting. We have a new version of the checker now, and it turns out that our current layout contains 617 violations of this constraint.

We can't in general just fix the violations by making tiny local changes, because in many cases the surrounding cells are packed together pretty tightly, so fixing the violations will require slightly spreading out a large amount of wiring and components. Alternatively, we could redesign the cells that have problems to try to fix the problems without actually taking more space or moving their pins; in that case, at least, we would not have to change the wiring of higher-level cells. We will probably take this approach (although there is not time to do it until after we have turned in this report).

4.2 Division of Labor

We worked closely together on the instruction set definition, choosing which Pendulum instructions to implement and making up their new 16-bit instruction encodings in discussions together. Then we broke up to work on the design.

Scott took the ALU, Register File, miscellaneous adders, and some of the basic logic gates. Mike did the controller, the program counter, the Master-Slave Flip-Flop, and some of the basic logic gates. We took turns working on the MUX design, which went through several revisions due to our feeling that our early versions were too large. Mike wired up the PC updating path, and Scott wired up the rest of the top-level design (the datapath and the control lines).

Each of us primarily worked with our own cells at all the different stages of design, from the original Verilog functional specification to the schematics to the layout, with all the corresponding stages of testing, although we often helped each other out with learning the tools and tracking down bugs in our designs. We used each other's basic cells quite frequently and managed to almost completely avoid unintentional duplication of work, although a few cells were first done by one person and then revised later by the other person to make them smaller or provide an alternative layout style.

Mike did most of the report writeup and preparation of figures, and Scott wrote the sections on the ALU, register file, testing, and conclusion.

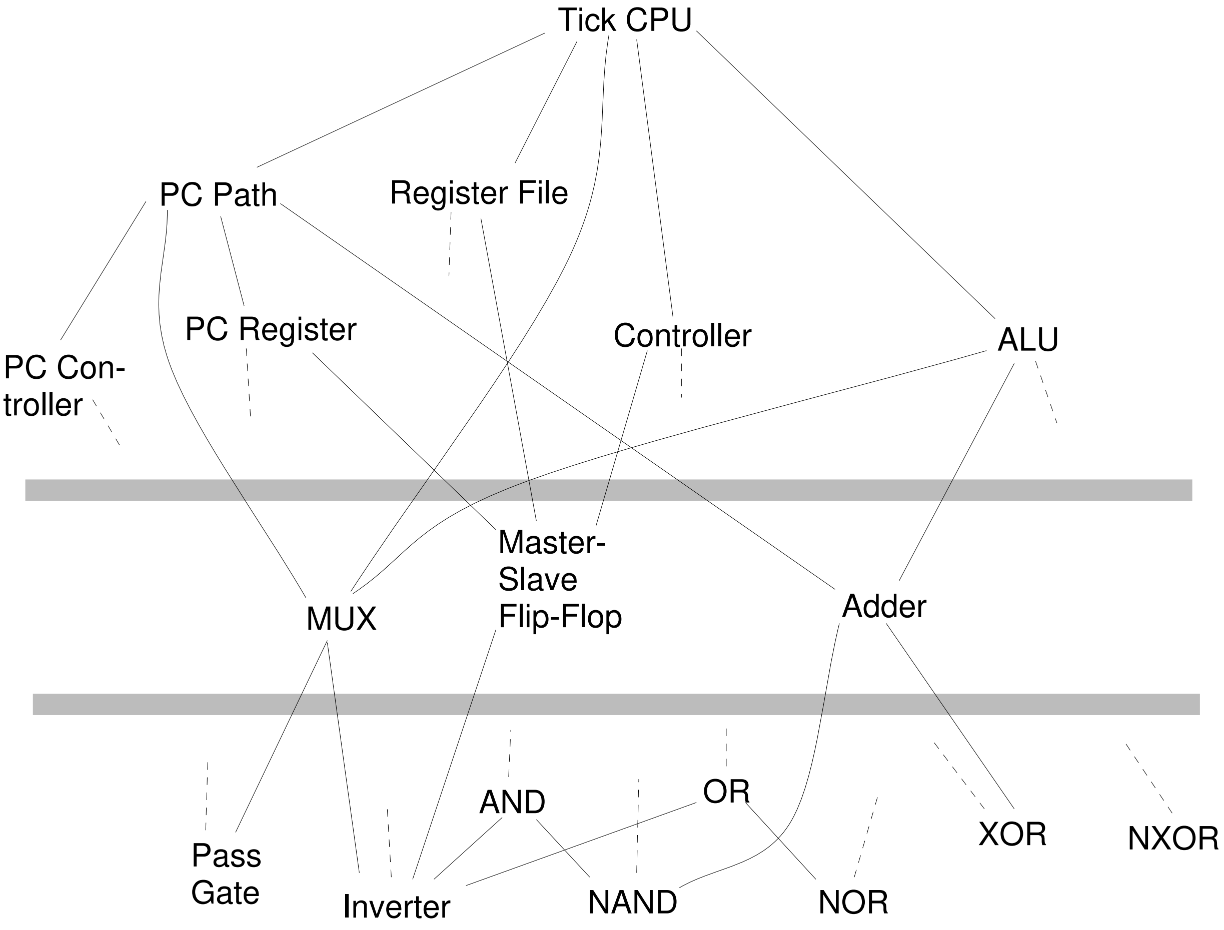


Figure 3: Partial Hierarchy of Cells in the Design

4.3 Discussion of Particular Cells

Figure 3 shows most of the main cells in their design and some of their inclusion relationships with each other. We chose a design style in which almost everything was built out of low-level logic gates, flip-flops, and MUXes, with no individual transistors appearing in the higher-level cells. Everything was done using fully-restoring, static CMOS circuit techniques, for simplicity, except for the barrel shifter (see section 4.3.6).

In layout we used a regular style in which all our gates were built from a strip of P-diffusion and a strip of N-diffusion with a minimum separation between them, so that multiple gates could be laid out beside each other in a long, uniform strip together.

4.3.1 Logic Gates

Figure 4 shows our inverter and our two-input generalized inverters (NAND and NOR). Note the layout style using parallel horizontal strips of diffusion with polysilicon strips going vertically. Also,

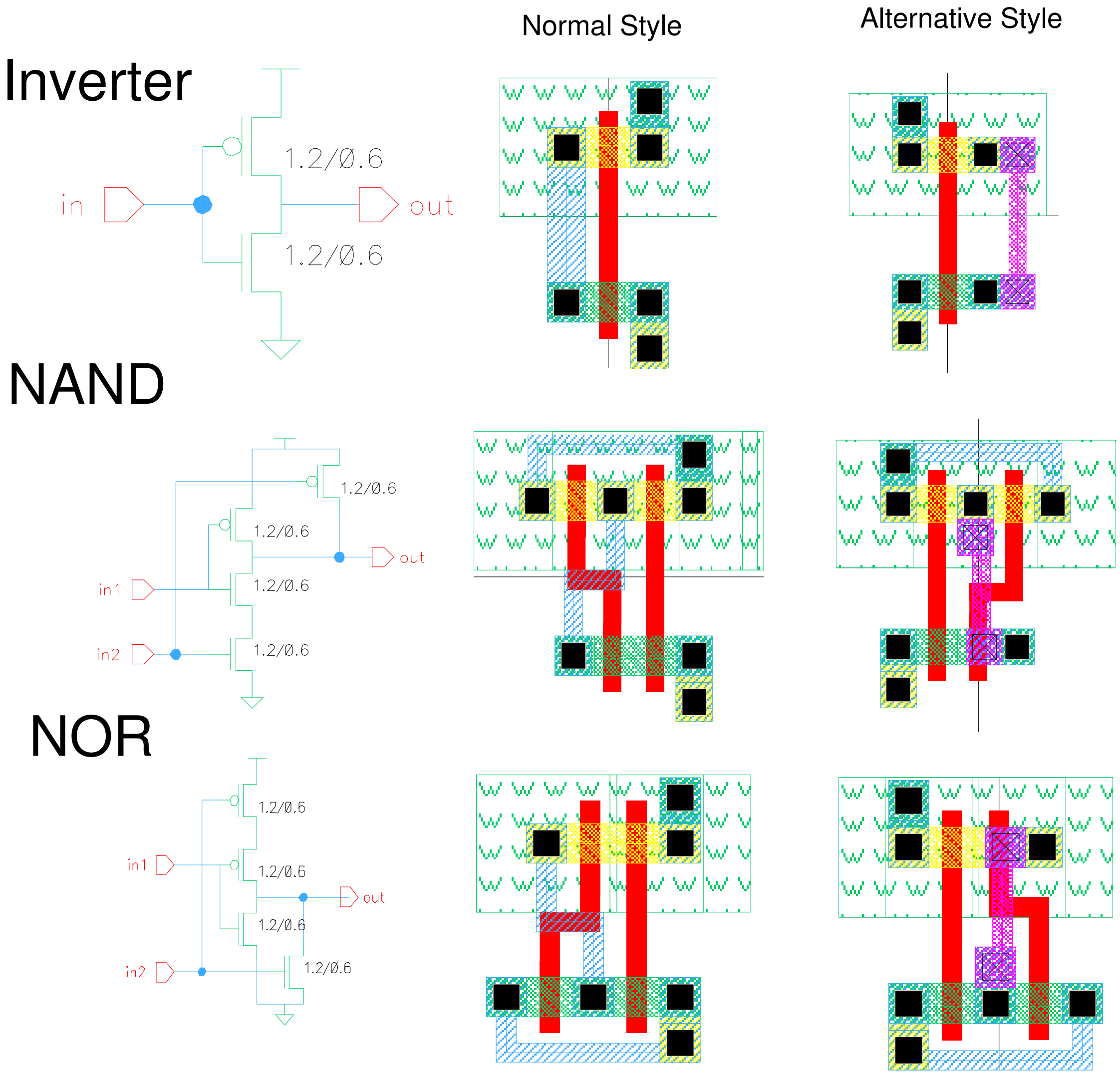
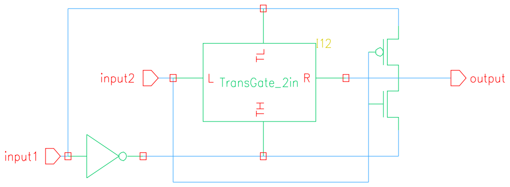
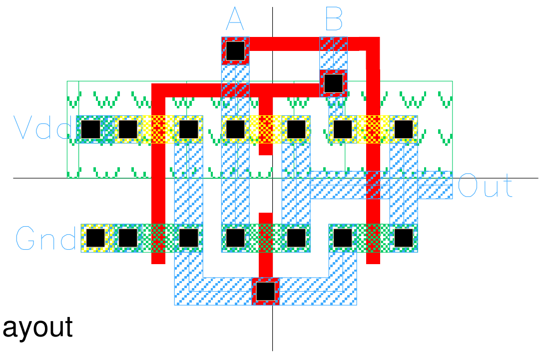


Figure 4: Inverter and generalized inverters.

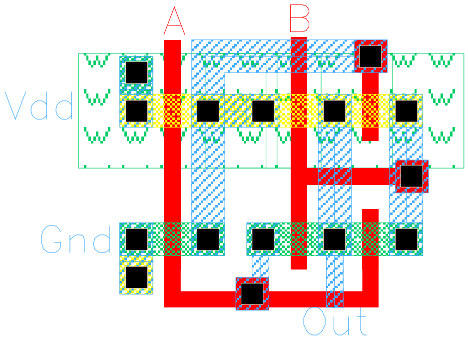
XOR



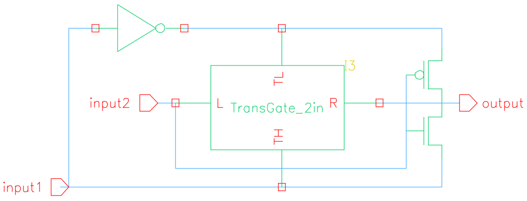
Earlier, larger layout.



Later, more compact layout



XNOR



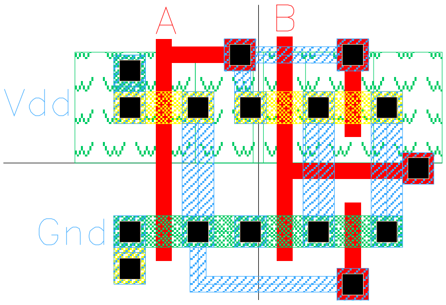


Figure 5: Exclusive-OR and negated XOR gates. The Transgate-2in block is an icon for a standard CMOS transmission gate (pass gate), which we use in many places. (We later made a better icon but haven't updated all our schematics to use it yet.) To transmit, TH should be high and TL should be low.

note that the body plugs stick up and down for easy connection with horizontal power and ground wires in metal1. We played around with a slightly larger alternative style (layouts shown to the right) in which the source/drain connection ran in metal2, so that metal1 wires could pass horizontally through the middle of the gate, while metal2 ran vertically to inputs and outputs. This alternative style was used for part of the CPU controller.

Figure 5 shows our XOR and XNOR (not exclusive-OR) gates. These are the 6-transistor fully-restoring static CMOS gates we learned about in problem set 3. Our first attempt at laying out XOR was somewhat large and did not fit together well with our generalized inverters, so we later made a smaller version whose style was also used for the XNOR.

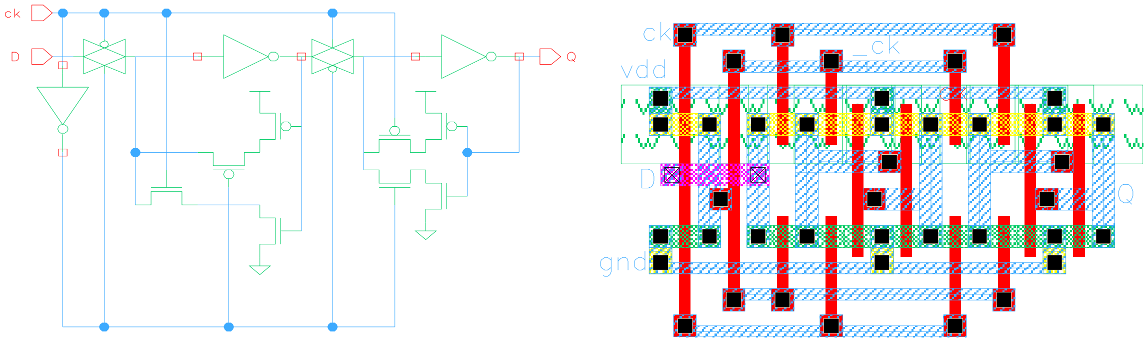


Figure 6: Master-slave flip-flop. (This uses our newer transmission gate icon.)

4.3.2 Master-Slave Flip-Flop

We used a single-phase clocking scheme with all latching done by having the clocks controlling the rising-edge triggered Master-Slave Flip-Flop shown in figure 6. These flip-flops were made out of ordinary recirculating latches, except that the normal inverters in the feedback path were replaced by clocked inverters in order to eliminate two pairs of contacts.

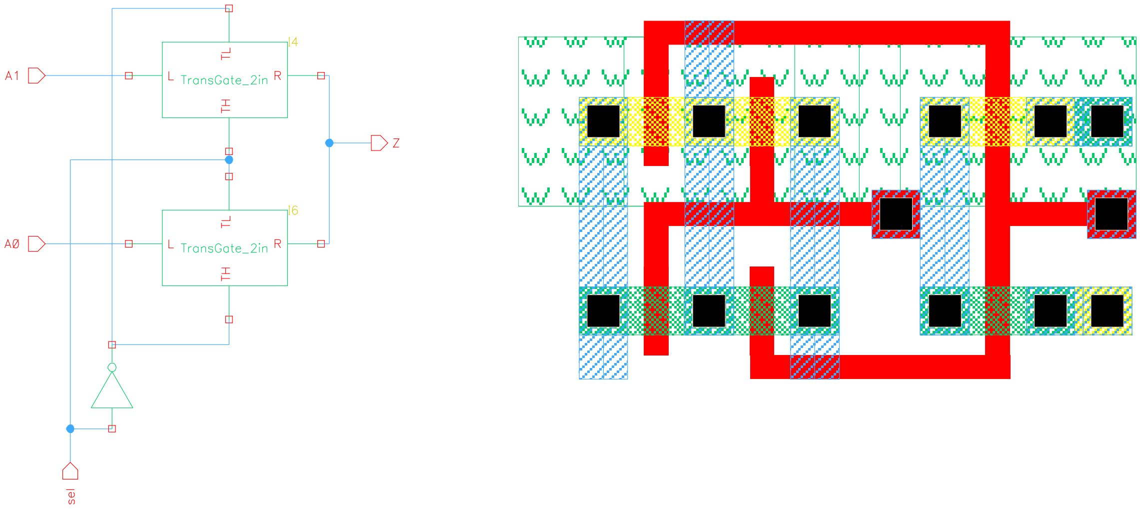


Figure 7: Two-to-one multiplexer cell. Again, TransGate2in is a normal transmission gate.

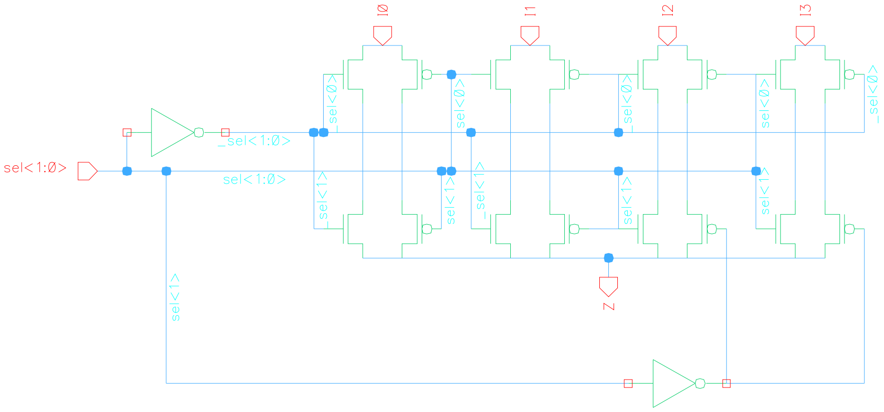


Figure 8: Early version, four-to-one multiplexer cell.

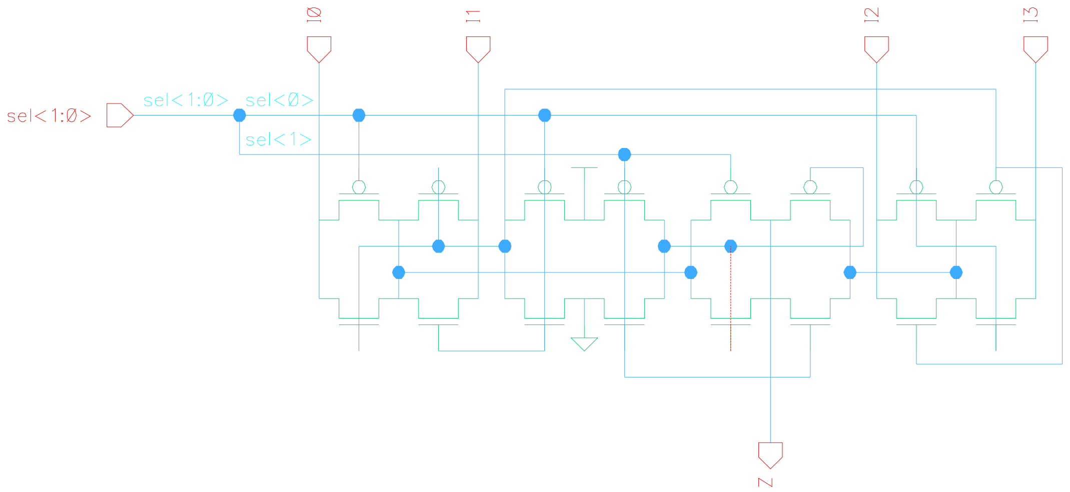


Figure 10: Later version, four-to-one multiplexer cell.

4.3.3 Multiplexers

Our designs used arrays of two-to-one (figure 7) and four-to-one multiplexers in several places. Sometimes we needed three-to-one multiplexers; in such cases we just used the four-to-one MUX and grounded one of the inputs.

The 4-to-1 MUX was one of the first cells to be laid out. At that point, the style of using a row of PFETS and a row of NFETS for each cell had not been established, so the original 4-to-1 MUX was somewhat irregular. After using the layout of our first four-to-one MUX for a while, we decided it was too large, and realized that it could be shrunk by eliminating one of the pairs of pass gates. So we made a new version and began using it in later layout. Figures 7-11 show the evolution of our 4-to-1 MUX. Figure 12 shows an array of eight 4-to-1 MUXes which was used in several places.

Unfortunately we realized there was a bug in the second version of the 4-to-1 MUX (some of its inputs were switched) only after including it in larger layouts, but we managed to fix it by rewiring it internally while leaving its I/O terminals in the same place. So its design (fig. 11) is a little more awkward than it would be if we could do it over again. The error was in the schematic as well as the layout, which was why the layout-versus-schematic checker did not catch it. This experience taught us the hard way that layout-schematic equivalence is not enough for correctness; one should also simulate each piece after any change to its schematics, to make sure it really does what you want, before using it in layout.

The 2-to-1 MUX is quite straightforward, and did not suffer from the problems that the 4-to-1 MUX did. In fact, in some places, the 2-to-1 MUX was replicated 3 times to effectively produce a 4-to-1 MUX, as it could easily be placed in such a manner to use the available space, rather than taking up the fixed space of the 4-to-1 MUX.

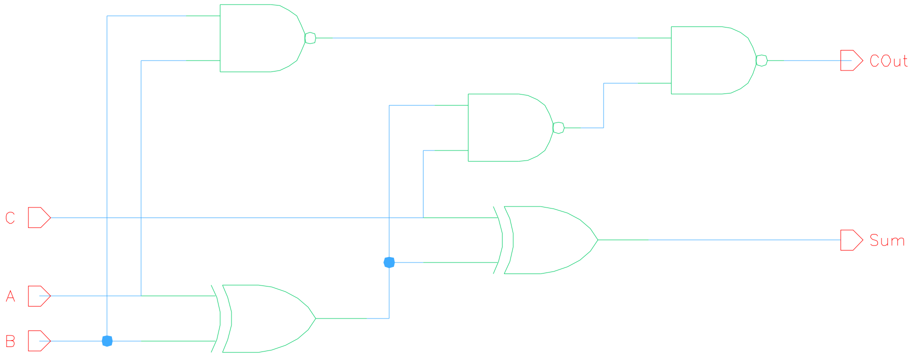


Figure 13: Full-adder cell schematic.

4.3.4 Adders

The adder, which is used in both the ALU and the PC path, is an 8 bit ripple carry adder. 8 full adders (built out of XOR and NAND gates) are connected in the obvious way to form a simple ripple carry adder. The carry out is ignored, as overflow is not handled in this machine (overflow is not a reversible operation).

4.3.5 Controller

The CPU controller finite-state machine (created by Mike) was one of the most time-consuming parts of the design, due to the sheer amount of logic that was needed to define all of the control signals based on the instruction opcodes and current state.

We decided that Tick needed four clock cycles to process each instruction (as opposed to the five that were used in the original Pendulum). The four cycles were called "stages" and were:

1. *Instruction fetch.* The instruction memory fetches the instruction at the current PC address.

2. *Instruction decode.* The controller uses the just-fetched instruction opcode to set up control lines for the next stage, while the register file fetches values.
3. *Execution stage.* The ALU performs an operation, or we read and write values to memory, while the next PC value is computed.
4. *Writeback stage.* The result of the ALU operation or memory exchange is written back to the register file, and the new PC value is clocked into the PC register.

The controller needed two bits of state to keep track of what stage it was in. It also needed one more bit to keep track of the current direction of execution (forwards or reverse). The external signal could not just be used directly for this, because the processor must only switch at certain times to ensure correct reverse execution, in a transactional fashion. Direction-reversal happens during writeback, during which we must refrain from updating the PC so that when we start running in reverse, the first instruction we will execute in reverse is the current one (rather than the next one).

Thus the controller needed to be an 8-state FSM. The first stage of creating it was to write down the logic formulas for all the control outputs and next-state bits in terms of the inputs (opcode, direction bit, reset signal) and current-state bits. Below are all the formulas in a C-like notation.

```
----------------------------------------------------------------------
Intermediate variables.

old_dir = StateIn<2>
stage<1:0> = StateIn<1:0>

undo = old_dir ^ Opcode<4> [Wrong dir for a "doing" op.]
immed = Opcode<5> & Opcode<0>
logari = !Opcode<5> | immed;

brj = Opcode<5> & !Opcode<0>
bcmp = brj & !Opcode<3> & !Opcode<2>
bchreg = brj & Opcode<3> & !Opcode<2>
balrbal = bchreg & Opcode<1>

selfrev = !Opcode<3> & !Opcode<2> & !Opcode<1>          [xor, neg]
invert = !Opcode<2> & Opcode<1>                      [add/sub, rrv/rlv]
op3 = Opcode<3> ^ (undo & invert)    [Flip MSB if reversing invertable.]

fzero = (logari & undo & !selfrev & !invert) | (brj & !bcmp)

seq = !brj | undo

exinst = Opcode<5> & Opcode<4> & Opcode<3> & Opcode<2> & Opcode<1> & Opcode<0>
stfin = Opcode<5> & Opcode<3> & !Opcode<2> & Opcode<1> & Opcode<0>

exec = stage<1> & !stage<0>

dobchreg = bchreg & !undo

pause = ((old_dir ^ dir_in) & !dobchreg) | (stfin & !undo)
poll = exec & pause
----------------------------------------------------------------------
These outputs are combinatorial & independent of the current stage.

ALU_Bsel<1> = immed
```

```
ALU_Bsel<0> = bcmp

ALU_op<3> = fzero | (logari & op3)
ALU_op<2> = fzero | Opcode<2>
ALU_op<1> = fzero | (logari & Opcode<1>)
ALU_op<0> = fzero | Opcode<0> | bcmp

PCsel<2> = seq | Opcode<3>
PCsel<1> = seq | Opcode<2>
PCsel<0> = seq | Opcode<1>

WBms<1> = balrbal & !undo
WBms<0> = exinst

----------------------------------------------------------------------
These outputs depend on the stage and the reset lines.

Iread = !_reset | (stage<1> & stage<0>)
exchange = _reset & exinst & !stage<1> & stage<0>
wZ = _reset & exec & ((logari & !stfin) | dobchreg | exinst)
LoadPC = _reset & exec & !pause
dir = _reset & ((poll & dir_in) | (!poll & old_dir))
StateOut<1> = _reset & (stage<1> ^ stage<0>)
stateOut<0> = _reset & !stage<0>
----------------------------------------------------------------------
```

After defining the logic equations, the next step was to implement them in schematic form and use it in our CPU simulations to make sure it worked correctly in the context of the whole processor. Regular non-inverting logic gates (AND and OR) were used to do this in the most straightforward way, since the concern was to put it in a form that would be easy to debug, and just get it right, rather than to try to be optimal the first time. The result was the mess of spaghetti shown in figure 16. It didn't look pretty, but it worked the first time, except for one bug that was due to a error in the original logic formulas (rather than in the schematic).

This irregular logic would obviously be tricky to lay out correctly, so next came a very difficult decision: to implement the logic as it was, or turn it into a PLA that could be laid out by tiling regular cells. To gauge how difficult the PLA would be, Mike wrote a simple C program to automatically generate a PLA description from the above formulas, by first generating the truth table and then repeatedly combining logically adjacent rows, as suggested in the course notes. A separate program was used to verify the PLA's correctness.

Unfortunately, it turned out that expressing the whole controller logic in PLA form would require a PLA that was 124 rows high. A small amount of row folding might have been possible, but probably not much, since the connections in the AND plane were pretty dense. If we had a program to automatically convert the PLA description to schematics and layout, it might have made sense to do it, just so that we would have a circuit that was guaranteed to be correct. But if we had tiled the PLA by hand, it would have been very time-consuming, and hard to get it right, and it would probably have taken more area than implementing the original logic. So it was decided to leave the original spaghetti schematic alone, and make a directly-corresponding spaghetti layout, shown in figure 17. There are some quirks and ugliness in the layout, but mostly it just uses the simple style of rows of gates between rows of interconnect, so it probably could have been worse. It passed the layout-versus-schematic checker after a bit of playing around with the order of inputs to the XOR gates.

The register to hold the current state was implemented with three of the Master-Slave Flip-Flops described earlier.

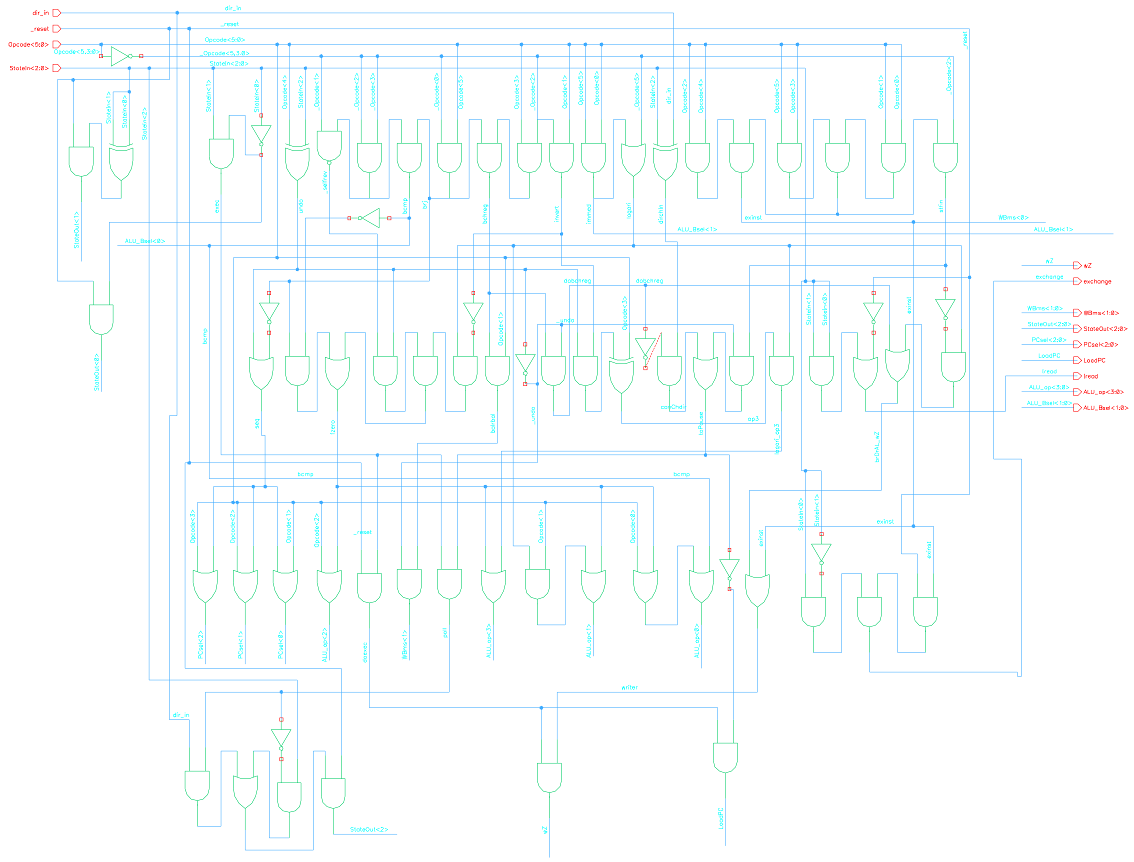


Figure 16: CPU controller combinational logic.

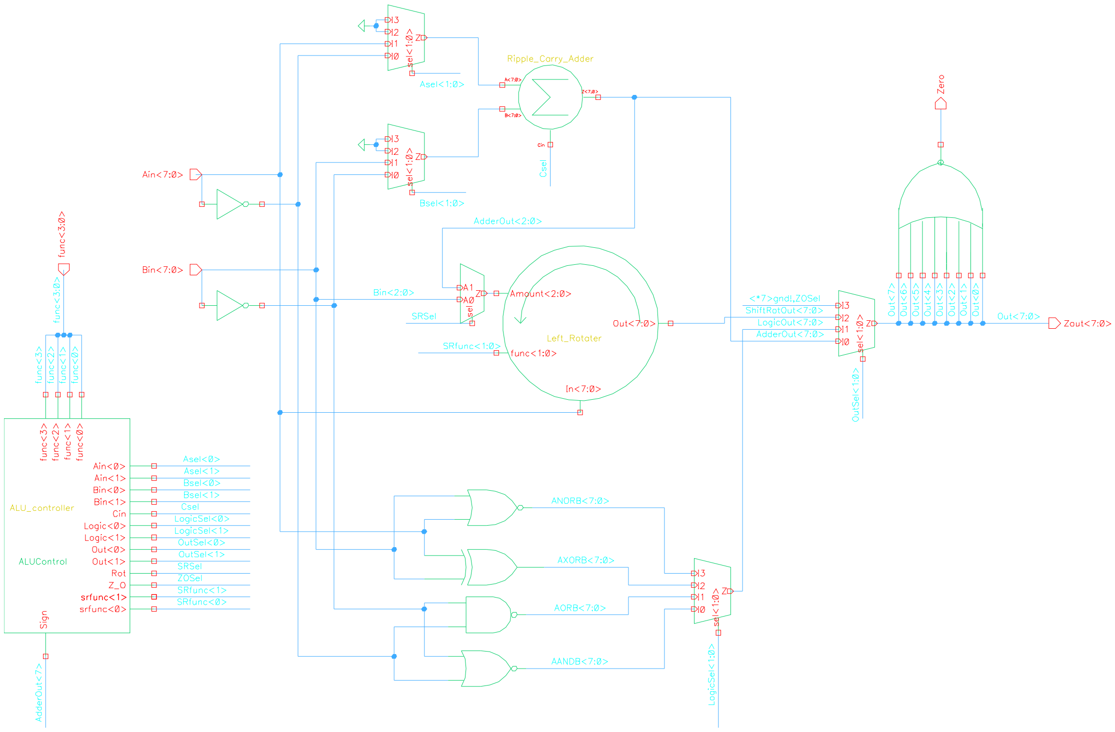


Figure 18: ALU schematic.

4.3.6 ALU

The ALU has 14 possible functions that it can compute, based on the 4-bit "func" input. The operations are as follows:

Logical	Arithmetic	Shift/Rotate
XOR	ADD	RLV
AND	SLT	RRV
OR	NEG	SRLV
NOR	SUB	RLLV
	ZERO	SRAV

These functions correspond exactly to the instructions of the same name, except for "ZERO", which just outputs a zero, and is used to implement the behavior of the various Undo instructions. In a real charge-recovering ALU, the Undo operations would need to do actual computational work in order to clear the result register without dissipating energy, and there would need to be a different ALU operation for each undo operation.

The functionality of the ALU was broken down into three major categories, logical operations, arithmetic operations, and shift/rotate operations. This led to a simpler modular design of the ALU. In addition to these three major modules, there is a simple ALU controller, which decodes the incoming function bits and outputs the appropriate MUX select bits.

The ALU will compute many of the given operations for a given instruction, and actually only output the appropriate result, as the others are discarded by MUXes. In order to aid in conditional branching, the ALU has a "zero" output, which is high when the result is zero. This allows the PC controller to decide whether or not the branch will be taken.

The logical operations are computed simply by arrays of logic gates. Compared with the rest of the ALU, this is extremely compact, with most of the space dedicated to the MUX that controls which logical output will be fed to the overall output MUX of the ALU.

The arithmetic operations are computed by an 8-bit ripple carry adder. Both the A and B inputs of the adder are fed by 3 input MUXes in order to allow for the full range of operations. (The 3 input MUXes are actually 4 input MUXes with the fourth input grounded to reduce the number of cells created in the layout stage.) A ripple carry adder was chosen over a faster method because of its simplicity. Since the memory latency will control the cycle time anyway, there is no need for a faster adder. This is especially true since the data is only 8 bits so the carry will not take that long to propagate through the adder.

The shift/rotate operations are all computed with a left rotator. Regardless of what the operation is, the data is run through some logic to change the input in anticipation of the fact that it will be rotated left. Then the resulting data word is rotated left by the specified number of bits if the operation is a left rotate or left shift. If the operation is a right rotate or right shift, the data word is rotated left by the twos compliment of the specified number of bits (as computed by the ripple carry adder). This allows a single barrel shifter to be used to handle all of the shift and rotate operations. In hindsight, this was not as clever an idea as it first seemed. When the layout was complete for the rotator, it was apparent that it would have been more compact to have at least 2 rotators (left and right), and possibly even 3 additional shifters.

The ZERO operation is performed simply by grounding one of the inputs of the output MUX. Since the SLT instruction can actually result in zero or one, the low order bit of this input is controlled by the ALU controller, so that ZERO and SLT can share the upper 7 bits of their output, and the output MUX can remain a 4 input MUX.

The “zero” ALU output bit is simply computed by having all 8 bits of the ALU output feed an 8 input nor gate.

ALU Layout. Most of the space in the layout of the ALU is taken up by buses. Now that it is complete, it is obvious that the layout is not as good as it could have been. By positioning the 3 major units in a way such that the buses could have been shared easily, the area could have been significantly decreased.

As mentioned earlier, the rotator design leads to a large layout. Most of the area is taken up by the logic that feeds the rotator with the appropriate input bits. The rotator itself is a simple barrel shifter using NFETS. This is the only place in the design where static CMOS was not used, and a buffer was placed on the output to bring the output voltage back up to Vdd when the output is supposed to be high.

The rotator dwarfs the other units in the ALU. The controller is a simple string of logic gates that computes all of the control signals necessary for the ALU to operate correctly. It is fed by a bus of all of the function bits and their complements that runs along all of the logic.

The ripple carry adder is smaller a straightforward concatenation of 8 full adders, and is fed by 2 sets of 4 input MUXes, which end up being almost as large as the adder itself.

The logical functions are computed by a block of logic gates combined with a MUX that is replicated 8 times to compute the complete logical result. This lays out nicely, as the blocks can fit beside each other and feed the logical output bus below them while taking their input from the bus above.

The outputs of these three units are bussed to the ALU output MUX where the correct result is selected and attached to an output bus.

4.3.7 PC Path

The PC path was straightforward, and involved using an array of flip-flops for the PC register, a couple of the 8-bit adders already used in the ALU to compute sequential PC addresses and branches, and an 8-bit 4-to-1 MUX to select the next PC value. The MUX was controlled via a small group of logic gates, which use the sign bit of register A and the “zero” bit from the ALU to decide whether to take conditional branches.

Unfortunately the PC path layout came out somewhat irregular (non-rectangular) in shape, and if we had time to do it over we would arrange its sub-cells somewhat differently.

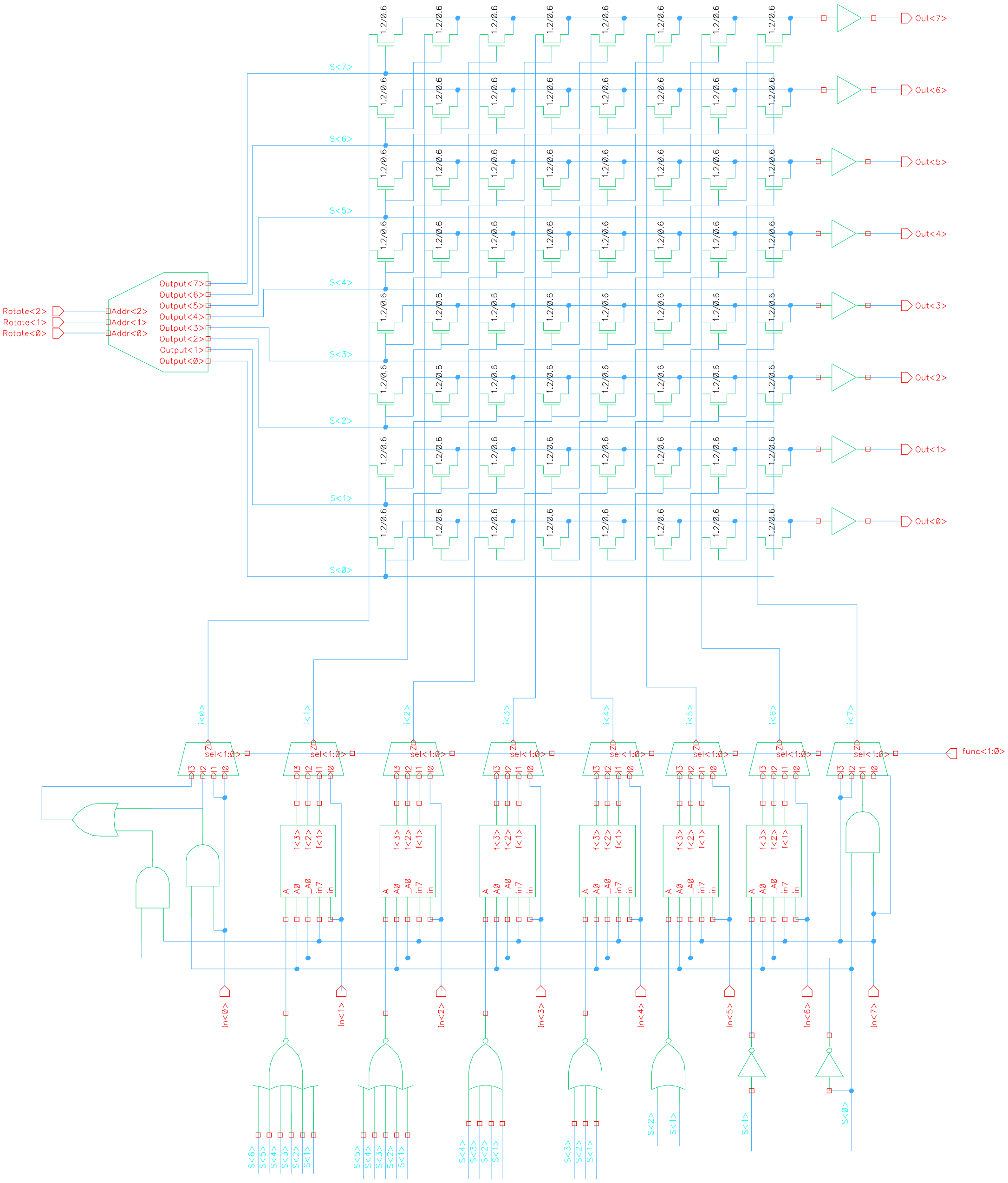
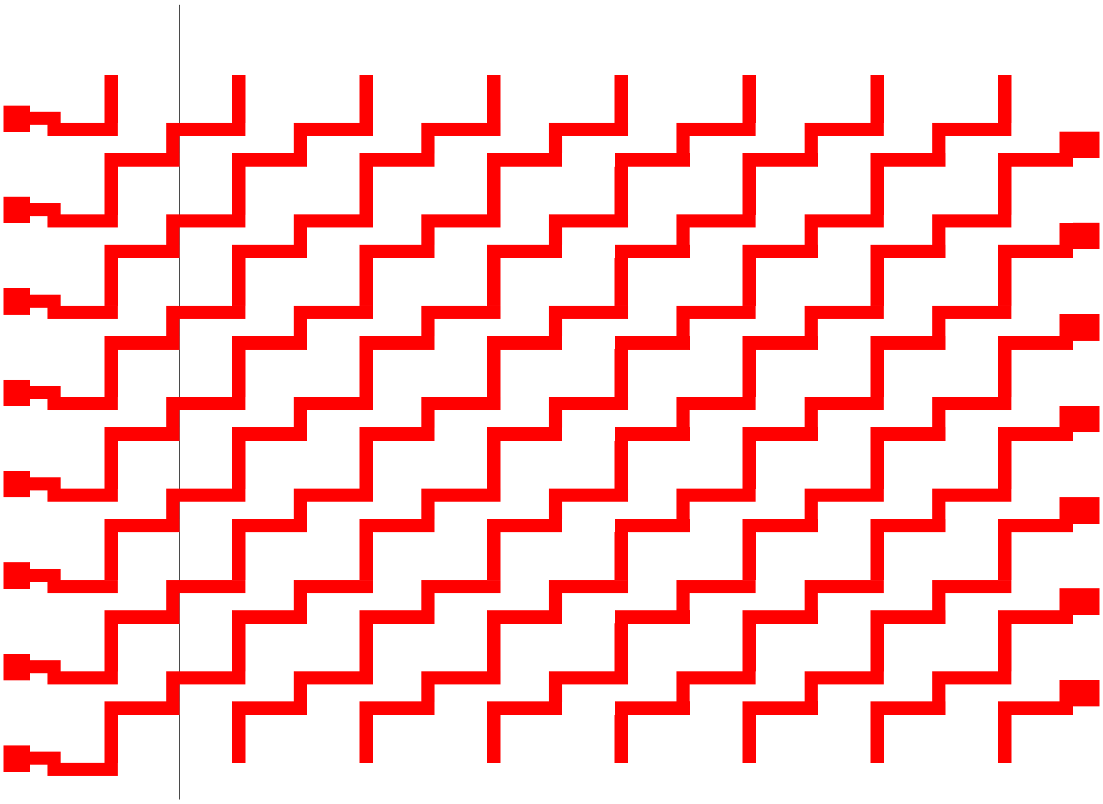


Figure 19: Schematic for Rotator/Shifter.



state. It was especially interesting to watch the sorting program "un-sort" a list of numbers and return them back to their original scrambled state, using the extra "garbage data" that had been unavoidably produced by the initial reversible sort. Everything worked correctly.

Here is the assembly code for the sort program, which is the most interesting.

5.1 Sort Program

```
// Annotated Tick assembly code for an insertion sort
// that uses Hall's trick to produce only O(n log n)
// garbage bits despite O(n^2) comparisons.
//
// Assumes initial state:
//    * All registers 0.
//    * All data memory 0.
//    * PC at 0 going forward.
//
// Algorithm:
//
//   Initialize array m to hold unsorted list.
//   for(j=2;j<=n;j++)
//     loop begin (i = j)
//       i--;
//       swap(m[i],m[i+1])
//     loop end (i=1 or m[i] > m[i+1])
//     swap(m[i],m[i+1])
//     push(i,garbage_stack)
//
// Initial unsorted list:  89 14  7 23 74 46 30 52 88  8
//                sorted:   7  8 14 23 30 46 52 74 88 89
//
// dec    bin       oct   hex
// ---  ---------   ---   ---
// 89   1 011 001   131   59
// 14       1 110    16   0E
//  7         111     7   07
// 23      10 111    27   17
// 74   1 001 010   112   4A
// 46     101 110    56   2E
// 30      11 110    36   1E
// 52     110 100    64   34
// 88   1 011 000   130   58
//  8       1 000    10   08


//----------------------------------------------------------------------

// Initialize memory to initial array.

start            // 0:

addi    $1 $1 3   // 1: $1 = 3, amount for shifting big values into $0.
addi    $0 $0 1   // $0 = 1
rlv     $0 $0 $1  // $0 = 8
addi    $0 $0 2   // $0 = 10, size of array.
```

```
exch    $0 $2      // m[0] = 10.
addi    $2 $2 1    // $2 = 1, pointer to next free memory cell.

addi    $0 $0 1    // 7: $0 = 1.
rlv     $0 $0 $1   // $0 = 8.
addi    $0 $0 3    // $0 = 11.
rlv     $0 $0 $1   // $0 = 88.
addi    $0 $0 1    // $0 = 89.
exch    $0 $2      // m[1] = 89
addi    $2 $2 1    // $2 = 2.

addi    $0 $0 1    // 14: $0 = 1.
rlv     $0 $0 $1   // $0 = 8.
addi    $0 $0 6    // $0 = 14.
exch    $0 $2      // m[2] = 14.
addi    $2 $2 1    // $2 = 3.

addi    $0 $0 7    // 19: $0 = 7.
exch    $0 $2      // m[3] = 7.
addi    $2 $2 1    // $2 = 4.

addi    $0 $0 2    // 22: $0 = 2.
rlv     $0 $0 $1   // $0 = 16.
addi    $0 $0 7    // $0 = 23.
exch    $0 $2      // m[4] = 23.
addi    $2 $2 1    // $2 = 5.

addi    $0 $0 1    // 27: $0 = 1.
rlv     $0 $0 $1   // $0 = 8.
addi    $0 $0 1    // $0 = 9.
rlv     $0 $0 $1   // $0 = 72.
addi    $0 $0 2    // $0 = 74.
exch    $0 $2      // m[5] = 74.
addi    $2 $2 1    // $2 = 6.

addi    $0 $0 5    // 34: 5
rlv     $0 $0 $1   // 40
addi    $0 $0 6    // 46
exch    $0 $2      // m[6] = 46.
addi    $2 $2 1    // 7

addi    $0 $0 3    // 39: 3
rlv     $0 $0 $1   // 24
addi    $0 $0 6    // 30
exch    $0 $2      // m[7] = 30.
addi    $2 $2 1    // 8

addi    $0 $0 6    // 44: 6
rlv     $0 $0 $1   // 48
addi    $0 $0 4    // 52
exch    $0 $2      // m[8] = 52.
addi    $2 $2 1    // 9

addi    $0 $0 1          // 49: 1
```

```
rlv     $0 $0 $1        // 8
addi    $0 $0 3         // 11
rlv     $0 $0 $1        // 88
exch    $0 $2           // m[9] = 88.
addi    $2 $2 1         // 10

addi    $0 $0 1         // 55: 1
rlv     $0 $0 $1        // 8
exch    $0 $2           // m[10] = 8.

addi    $2 $2 -8        // 58: 2
addi    $2 $2 -2        // 0
addi    $1 $1 -3        // 60: 0

//
// Now all registers should be clear again and memory should
// be loaded up.  We can start the real program now.
//
// Register allocation:
//      0       location just past end of memory array.
//      1       pointer to free memory.
//      2       j, the position of the next unsorted item.
//      3       i, the current position during insert.
//      4       2, the first value of j.
//      5       value to be inserted
//      6       t2, temp value #2
//      7       1, final value of i

exch    $0 $2           // 61: $0 = mem[0] = size of array to sort. (10)
addi    $0 $0 1         // 62: $0 = location just past end of array (11)
add     $1 $1 $0        // 63: $1 = $0
addi    $1 $1 0         // 64: $1 = pointer to free memory (11)
addi    $2 $2 2         // 65: $2 = j=2 -- next item to insert.
addi    $4 $4 2         // 66: $4 = 2 (initial value of j)
addi    $7 $7 1         // 67: $7 = 1 (final value of i)

j       70              // 68: jump into outer loop.
rj      103             // 69: If initial cond. fails, cont. outer loop
rbeq    $2 $4 -1        // 70: j=2: Initial condition for outer loop.

exch    $5 $2           // 71: $5 = m[j]
add     $3 $3 $2        // 72: $3 = i=j - current location in sorted list.

j       75              // 73: jump into inner loop.
rj      87              // 74: If initial condition fails, keep going back.
rbeq    $3 $2 -1        // 75: i=j: Initial condition for inner loop.

addi    $3 $3 -1        // 76: i--
exch    $6 $3           // 77: t2 <- mem[i] (next value in sorted list)
sub     $6 $6 $5        // 78: t2 <- value in sorted minus value to insert

bgez    $6 1            // 79: value in sorted list is larger, keep going
j       91              // 80: hey, we're in the right place! Quit loop.
rj      79              // 81:
```

```
add     $6 $6 $5        // 82: t2 <- value in sorted l.
addi    $3 $3 1         // 83: i++
exch    $6 $3           // 84: mem[i] <- t2
addi    $3 $3 -1        // 85: i--

beq     $3 $7 1         // 86: i=1: Final condition for inner loop.
j       74              // 87: If final cond fails, continue loop.
rj      86              // 88: Go back into inner loop.

// We get here, it means value to be inserted
//  should go at beginning of array.

exch    $5 $3           // 89: mem[1] = t1
j       97              // 90: skip to end of outer loop.

// Here is where we go when we terminate the loop from
// inside the middle of it.

rj      80              // 91: Back into middle of loop.
add     $6 $6 $5        // 92: t2 = value in sorted list.
exch    $6 $3           // 93: Put it back where it belongs.
addi    $3 $3 1         // 94: i++
exch    $5 $3           // 95: mem[i]=t1, put value to insert where it belongs

j       98              // 96:
rj      90              // 97: if i=1 then value was inserted at beginning.
rbne    $3 $7 -1        // 98: if i!=1 then value was inserted in middle.

exch    $3 $1           // 99: Put final i value in free memory.
addi    $1 $1 1         // 100: Increment free memory pointer.
addi    $2 $2 1         // 101: j++

beq     $2 $0 1         // 102: Final condition for outer loop: j=n+1.
j       69              // 103: If fails, continue outer loop
rj      102             // 104: backwards enter outer loop

finish                  // 105:
```

6 Testing

Since the complete architecture of Tick was almost all implemented on-chip, the chip can be tested by running programs on it and checking the results. The only thing missing from the chip is the external memory interface and the memory itself. This will be implemented off-chip and connected to the CPU in order to test it. This will make it a little easier to load the instruction memory with programs, as a mechanism to do this does not need to be on the chip, and can be implemented with a simple PAL to load an external RAM, or by burning an EPROM.

The biggest issue with having the memory external to the CPU is that Tick has an exchange instruction rather than load or store, which does not necessarily mesh well with conventional memories. The memory will have to be read and then written. This may require a little additional circuitry, depending on the cycle times of the memory.

7 Conclusion and Comments

In this project we have nearly completed the design of a CPU that is fully reversible on the instruction-set level. To our knowledge, Tick will be, when fabricated, the world's first reversible computer implemented directly in practical hardware (as opposed to being simulated on a computer that has a non-reversible instruction set), and serves as a concrete demonstration that such computers are straightforward to design. Moreover, when a charge-recovering version of Tick is designed later, this initial CMOS version will play the role of a "control" for comparison in power dissipation experiments.

Overall, this project turned out to be a little more ambitious than we had expected. In order to manage the size and complexity of the layout, if we had this to do over we would do a few things differently.

First of all, we learned that it pays to lay things out regularly with a consistent style. While some low level cells may turn out a little larger than they could have been, this ultimately allows these cells to be included in larger cells without much hassle. These larger cells can then be quite compact, as interconnections are much more regular if the basic cells follow a regular and consistent style.

It also would have made things significantly easier if we had thought about which basic cells we were going to implement right from the beginning. Then only those cells could have been used in the schematics, so that the layout of each schematic would have been much more straightforward.

In future designs, we would definitely spend more time thinking about how the pieces of the top-level layout would fit together. It was surprising to us how much space all of the buses actually took up. If major block placement in the layout were driven more by bussing concerns than by signal path and preconceived schematic notions, we probably could have saved a lot of space by making many of the buses shorter.

It was unfortunate that the design rule checker that we were using did not incorporate all of the correct rules for the process we were using. While we could have avoided this problem by using the same tools as the rest of the class, we were glad to have the extra area that we were able to get by using a better process. In the future, though, we would probably put more effort into verifying that the design rules were correct before we began any significant layout work.

Ultimately, we are very happy with the results of our project, and look forward to being able to run reversible programs on our Tick processors when we get them back!