Mudd][Microprocessor

E158 Chip Report

Spring 2008

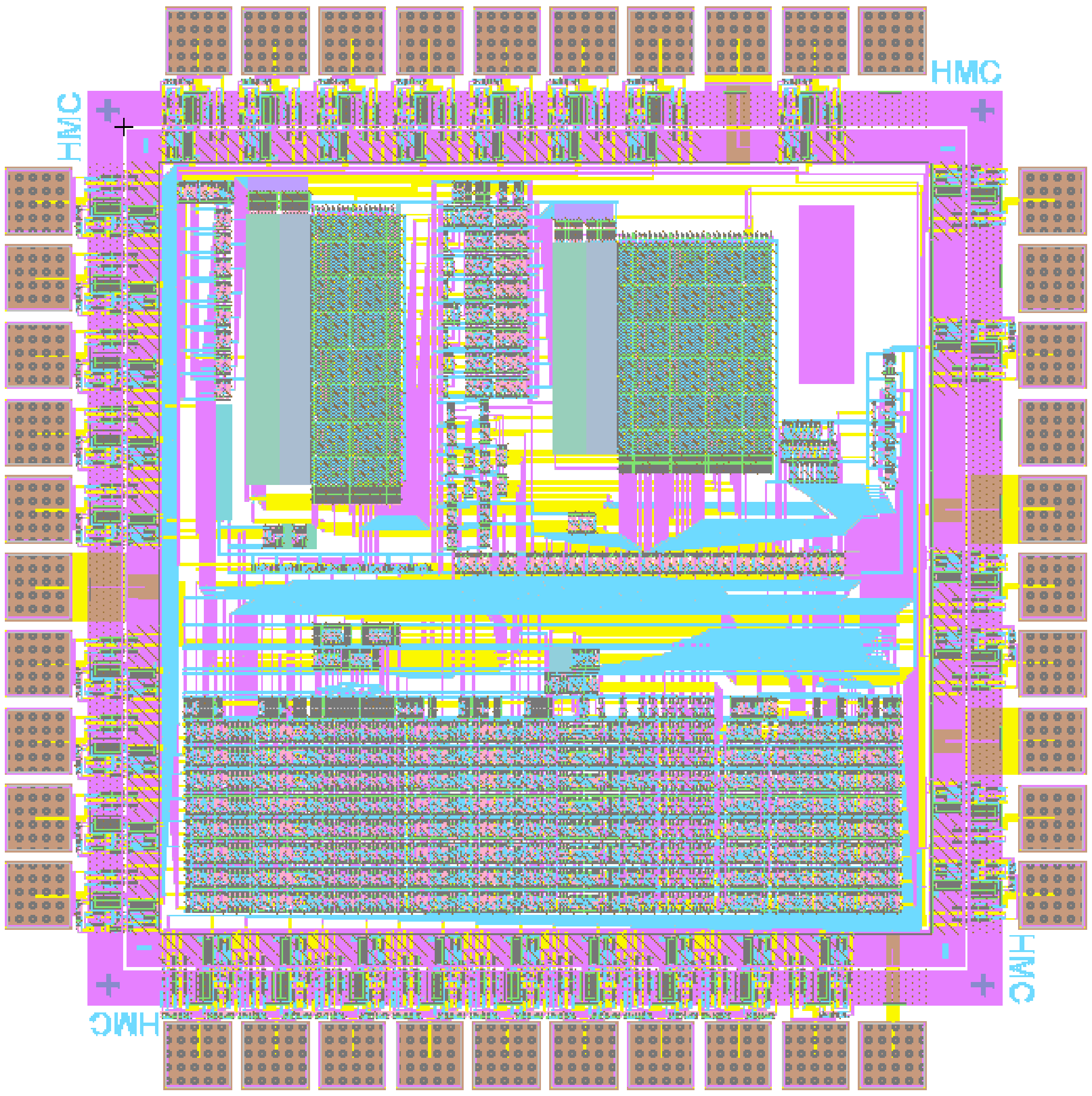


Table of Contents

Introduction

The Harvey Mudd College Spring 2008 class of E158, Introduction to CMOS VLSI Design, has designed and built the Mudd II microprocessor that is pin-compatible with the classic 6502 which inspired the first generation of personal computers. The chip supports nearly the full instruction set and is expected to be a drop-in replacement for the processor on an Apple II motherboard. Interrupts and binary-coded decimal (BCD) arithmetic are not needed or implemented.

The test chip is fabricated on a MOSIS 1.5x1.5 mm TinyChip in the AMI 0.5 micron CMOS process (λ = 0.3) in a 40-pin DIP package. It was developed using the Electric CAD suite, along with ModelSim for logic verification, HSPICE for electrical and timing verification, and IRSIM for silicon test vector generation.

The class wanted to build a relatively complex 8-bit CPU, and the success and history of the Apple II series computer made the MOS Technology 6502 the obvious choice.

One of the objectives of the project is to achieve minimum power consumption at the required 1 MHz operating frequency. The chip uses 5 V I/Os, but a variable core voltage, and includes 14,400 transistors. Core power consumption is predicted from SPICE to be 12 mW at 5 V and 46 μW at 1.5 V.

Another objective is to explore using Razor transparent latches to detect incipient timing errors before they occur. The chip produces an ERROR signal warning that the critical path is about to miss its setup time. This also facilitates lowering the core voltage until the chip is at the edge of failure.

The chip was designed by:

Nathaniel Pinckney – Chief of Circuit Design
Thomas Barr – Chief of Microarchitecture
Heather Justice – Microarchitecture
Kyle Marsh – Microarchitecture
Jason Squiers – Head of Schematics

Eric Burkhart – Schematics
Trevin Murakami – Schematics
Sam Gordon – Clocking and Razor Latches
Tony Evans – Clocking and Razor Latches
Michael Braly – Head of Layouts
Nisha George – Layouts
Corey Hebert – Layouts
Matt Jeffryes – ROM Generation
Steve Huntzicker – I/O
Professor David Harris – Instructor

History

The MOS Technology 6502 was one of the first low-cost, widely available microprocessor. At the same time that competing microprocessors cost over $125, the 6502 cost $25. An 8-bit design with 16-bit address space, the 6502 was a great success in the new personal computer world. The CPU quickly became the part of choice for both large manufacturers of PCs and individual hobbyists for its ease of programming, performance and cost.

The processor derives from the Motorola 6800. After the engineering team for the 6800 left Motorola, they formed MOS Technology to develop their own CPU. There, they developed the 6501, an enhanced version of the 6800. It supported new addressing modes while maintaining pin-compatibility with the original CPU. After a lawsuit, the 6501 was replaced with the 6502, a version that broke pin-compatibility by reordering the pins on the CPU. **Figure 1** shows a die photograph of the chip, which contains a datapath at the top, ROM or PLA at the bottom, and some random logic in the middle.

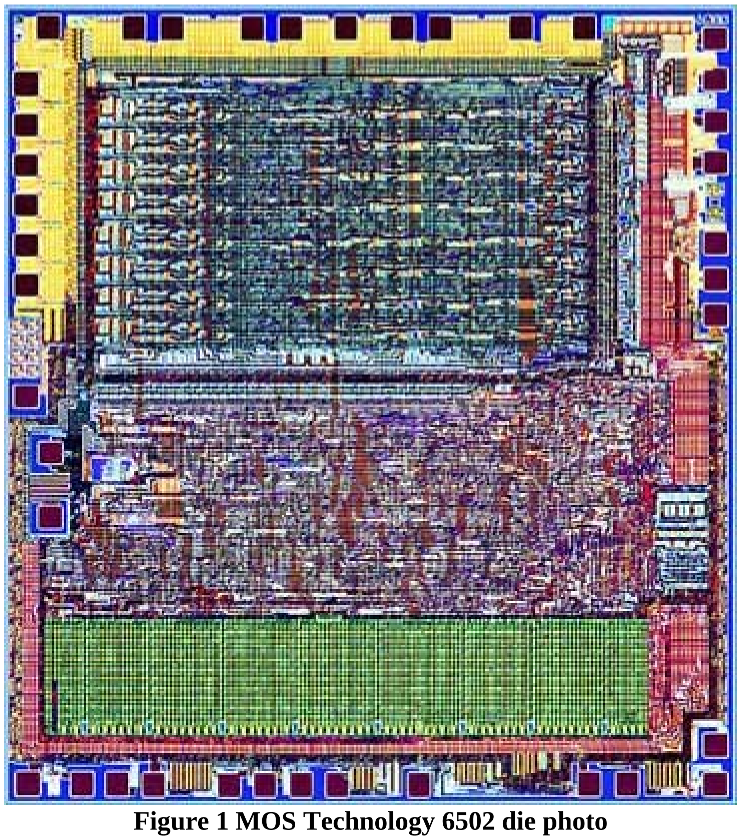


Figure 1 MOS Technology 6502 die photo
[http://micro.magnet.fsu.edu/chipshots/mos/6502small.html]

One of the first uses of the 6502 was the Apple I. The Apple I, a single board computer, was sold at computer club meetings around the Bay Area. While it never sold in great numbers, the Apple I led into the development of the Apple II. The Apple II had many useful features for a personal computer, such as a rudimentary sound system, color graphics and a built in BASIC interpreter, but one of the most revolutionary features of the Apple II was that it was one of the first personal computers to come built into a case. The Apple II was designed to be a computing appliance, for an end user, not a development platform for an engineer, and is largely responsible for the "microcomputer revolution".

Microarchitecture

The Mudd II is implemented with a multicycle datapath. The controller uses two ROMs; the opcode ROM decodes each instruction, while the state ROM sequences through the appropriate states to execute the instructions. **Figure 2** shows a block diagram of the processor's controller.

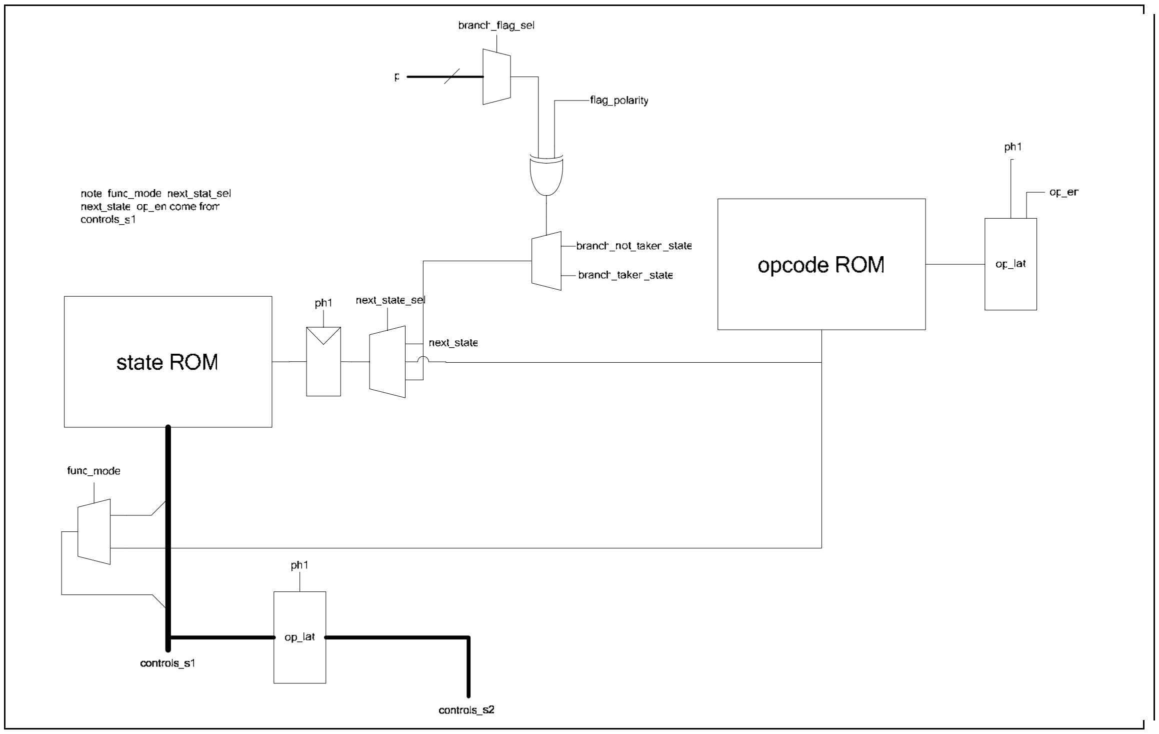


Figure 2 Mudd II Controller Block Diagram

Much of the control is accomplished with the state ROM. The state ROM contains a base state which allows an opcode to be read from the program into the opcode ROM, producing signals that are specific to that particular instruction. Additionally, the opcode ROM specifies which state to jump to after the base state. The controller then steps through a sequence of states and, upon completion of the instruction, returns to the base state in order to retrieve the next opcode. A state can either specify the control signals for the cycle or select the function-specific control signals decoded by the opcode ROM. Branches have a special effect on the sequence of states to be followed; the controller must first determine whether the branch will be taken and then either execute additional states in order to take the branch or jump directly back to the base state in order to not branch. The opcode ROM can decode all 151 opcodes of the 6502 complex instruction set and the state ROM contains 118 states which implement every 6502 instruction except for binary coded decimal arithmetic.

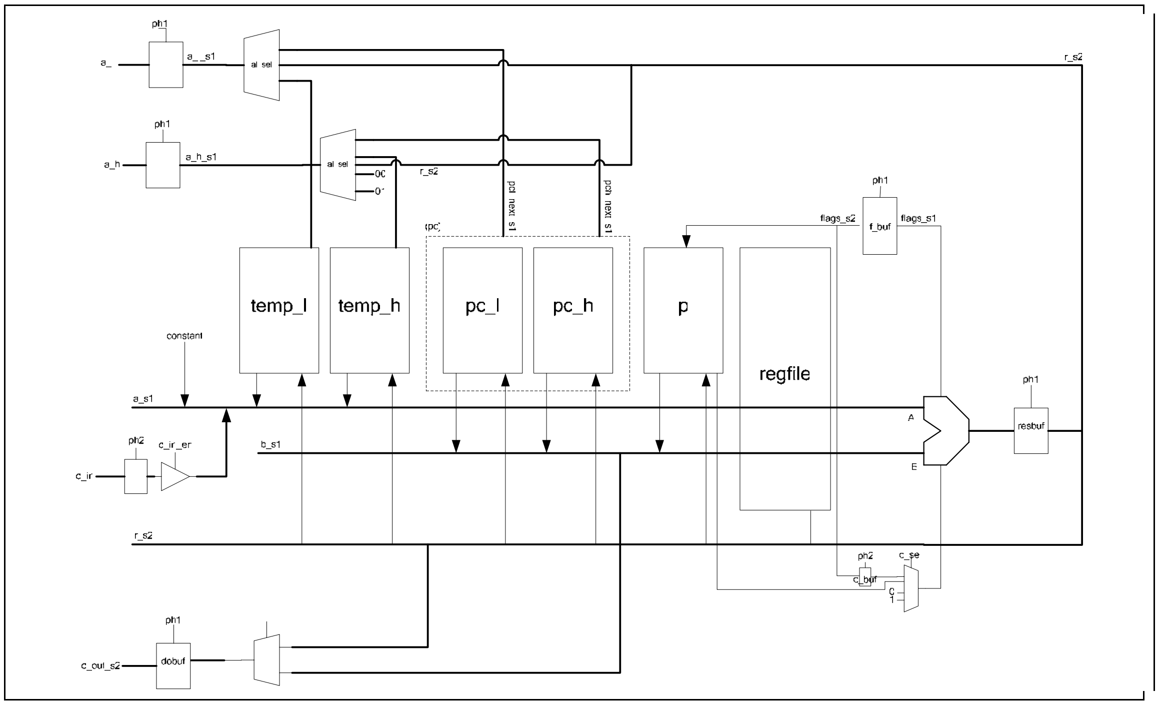


Figure 3 Mudd II Datapath Block Diagram

Figure 3 shows a block diagram of the processor's datapath. Registers in the datapath contain the program counter (both a low byte, pc_l, and a high byte, pc_h), the processor state flags (p), and the register file values of A, X, Y, and stack pointer (all stored in regfile). Temporary registers (temp_l and temp_h) may be used for storing values within a particular sequence of states. Each cycle sends certain data through the ALU on databuses A and B as determined by the control signals. Databus A may get a constant value (specified by the controller), the data read in from memory, the value of either temporary register, or any value stored in the register file. Databus B may get either byte of the program counter, the processor flags, or any value stored in the register file. The carry-in to the ALU may be selected from the processor carry flag, a temporary carry from a previous cycle, the constant one, and the constant zero. The ALU operation may change certain flags in the processor state register, as determined by the control signals. The result produced by the ALU may be stored in any one register, be sent as output data to memory, or used in the next address to be read from memory. The low byte of the next memory address (a_l_s1) may come from the PC low byte, the temporary low byte register, or immediately from the ALU result and the high byte of the next memory address (a_h_s1) may come from the PC high byte, the temporary high byte register, the constant zero, or the constant one.

As an example of the microcode functionality, consider the opcode 0x7D, which performs the instruction ADC using the absolute X addressing mode. ADC is the add with carry instruction, which takes a value from memory and adds it to the value in the accumulator, storing the result in the accumulator. The accumulator is a register, A, in the processor's register file generally used by the programmer to store intermediate results without writing to main memory. Absolute X addressing takes a two byte absolute address and increments that address by the value stored in register X. For example, if the register X has value 0x02, then the assembly instruction "ADC $10FF,X" will sum the value of the accumulator with the value stored at memory address $1101. The following is a line from our instruction table (Appendix: Microcode Assembler:instrtable.txt) which indicates what the control signals specific to this opcode should be:

Opcode	Instr	AddrMd	SrcA	SrcB	Dest	ALUOp	c_sel	B_polar	flags
7D	adc	abs_x	dp	A	A	2	p	0	C3

`Opcode` indicates which instruction byte value gets these signals, `Instr` indicates which instruction we're using (intended only for readability), and `AddrMd` indicates which state in the state ROM will be executed for this instruction. `SrcA` and `SrcB` indicate where the data into the ALU is coming from (where `dp` is the memory datapath and A is the accumulator) and `Dest` indicates where the result will be stored. `ALUOp` indicates which operation the ALU will perform (ALU operation 2 is addition, as indicated in the Verilog), and `c_sel` indicates where the carry-in comes from (where `p` is the processor state register, or flag register). `B_polar` indicates branch polarity (only relevant to branch instructions). The `flags` column indicates which bits of the processor state register can be changed by this instruction. Each bit in the processor state register represents a flag as follows:

Bit	Abbreviation	Flag
7	N	Negative result / Sign bit
6	V	Overflow
5	_	Expansion bit (unused)
4	B	Break command
3	D	Decimal mode (not implemented in the Mudd II)
2	I	Interrupt disable
1	Z	Zero result
0	C	Carry flag

Thus in this example, the opcode 0x7D enables the bits 11000011 (0xC3) so that it may only change the Negative flag, the Overflow flag, the Zero flag, and the Carry flag. Each line of the instruction table gets compiled into a set of bits to be output by the opcode ROM given the input opcode (see Appendix: Microcode Assembler).

The following is a sample of the states described in the microcode table (Appendix: Microcode Assembler: 6502.ucode) for the state ROM:

```
base:
a_sel b_sel alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel  tl_lat  flag  sta_src last_cy
db    -     pass    none    1       pc_n    pc_n    pc_n     0       0     opcode  0

imm:
a_sel b_sel alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel  tl_lat  flag
db    func  func    func    1       pc_n    pc_n    pc_n     0       1

abs_xy_final:
a_sel b_sel alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel  tl_lat  flag  next_s
db    -     pass+t  none    0       -       r       temp     0       0     imm

abs_x:
a_sel b_sel alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel  tl_lat  flag  next_s
db    x     add     none    1       pc_n    pc_n    pc_n     1       t     abs_xy_final
```

This sample shows four labeled sets of states (`base`, `imm`, `abs_xy_final`, and `abs_x`) which are all used by the ADC instruction with absolute X addressing. Below each label, each column name indicates which signals are important to that set of states (any other signals will be given default values). Each line below the column names indicates the signals for a state. In these examples, it happens that only a single state is given for each label.

The `base` state is always executed first for a new instruction. The general purpose of the `base` state is to read in the next opcode from the program and to decode it. Specifically, it reads in the opcode on databus A from memory (indicated by `a_sel` assigned `db`) and passes it unchanged through the ALU (technically incrementing the value by zero), and thus the value on databus B (`b_sel`) is irrelevant. No register data is written as indicated by `wrt_en` being assigned none. The `pc_w_en` signal assigned 1 indicates that the program counter may now be written to, and `pc_sel` assigned `pc_n` indicates that the program counter will be incremented. The address of the next value to read from memory is indicated by `a_h_sel` (high address byte) and `a_l_sel` (low address byte), and in this case it will read the value at the next program counter. The `signal tl_lat` assigned 0 indicates that nothing needs to be saved to the temporary low byte latch and flag assigned 0 indicates that no processor state flags will be written. The `sta_src` and `last_cy` columns are specific to the base state, indicating that the controller should read in the opcode-specific values, including the next state, from the opcode ROM.

In the example of opcode 0x7D, the next state to jump to is labeled `abs_x`, as indicated by the opcode ROM. The general purpose of this state is to add the value in register X to the low byte of the address indicated by the program. Specifically, it takes a value in from memory on databus A which, following the base state, will be the next byte in the program immediately after the opcode. In the case of the absolute X addressing mode, this value will be the low byte of an address. The value stored in register X will go on databus B. The ALU will then add these two values. The signal `tl_lat` assigned 1 indicates that the result from the ALU should be stored in the temporary low byte latch, and flag assigned t indicates that the flags produced by the ALU should be stored in the temporary flag register. The values for `wrt_en`, `pc_w_en`, `pc_sel, a_h_sel`, and `a_l_sel` act the same as in the base state; no register values are overwritten and the next byte of the instruction will be read from memory. The next state is labeled `abs_xy_final`, as indicated by `next_s`.

In the `abs_xy_final` state, the high byte of the address is now read from memory onto databus A. When this byte is passed through the ALU, it will be incremented by the carry flag from the temporary flag register, thus completing the two byte add of single byte register X and two byte address. The address for the next byte to be read from memory will use the result from the ALU as its high byte (as indicated by `a_h_sel` assigned `r`) and the value stored in the temporary low byte latch as its low byte (as indicated by `a_l_sel` assigned `temp`). In this state, the program counter is not changed, as indicated by `pc_w_en` assigned 0. The next state is labeled `imm`, as indicated by `next_s`.

Once the imm state begins, the memory has already been given the address of the value to be added to the accumulator, so this value is transmitted on databus A. The `func` label indicates that the values for a signal depend on the opcode. In this example, `b_sel` will read the accumulator value onto databus B as indicated by the opcode ROM. The ALU then performs an add on these two databus values and the result is stored back in the accumulator, again indicated by the opcode ROM. The `flag` signal assigned 1 indicates that flags in the processor state register may be assigned by the ALU, depending on which flags the opcode ROM allows to be changed. Since there is no next state specified, this state will jump back to the base state. Finally, the program counter is incremented so that the next opcode in the program will be read.

To minimize the number of datapath resets required in the hmc-6502 design, the controller is responsible for ensuring a consistent startup state. On processor reset, the controller FSM state is reset to zero via a resettable latch. This begins the reset sequence. Using the constant feature of the controller, zeros are loaded into the various flag and data registers of the CPUNext, the controller jumps to the startup location set in the "startup vector". The desired location of the program counter for startup is stored in memory at location {0xfffd, 0xfffc}. These addresses are asserted onto the memory bus, and the results stored in the program counter. Once the program counter points to the proper startup address, the first instruction is loaded and execution continues as normal.

External Interface

The Mudd II has the following I/O signals:

Inputs	**Outputs**	**Bidirectional**	**Power**
ph0	a[15:0]	d[7:0]	padvdd
reset_b	read_en		vdd
osc_en*	razor_error*		gnd
	ph1		
	ph2		
	osc_out*		

The osc_en, osc_out, and razor_error pins are not part of the original 6502 package. The razor_error output reads high when the razor latches in the clock generator and core detect an error in the signal through the latch. The original 6502 did not use razor latches- it used standard D latches. Signals osc_en and osc_out are the input and output respectively of the ring oscillator used in testing the padframe and the transistor speed at different voltages. These three signals use the three pins on the 6502 chip that are either not connected in the original package, or that are left out of the design because they are not used in the Apple II.

The read/write enable (read_en) is low during a write and changes on the rising edge of ph1. Thus the bidirectional data bus is an output for an entire cycle of ph0. The original 6502 only drives the data bus for a memory write during ph2, otherwise the data port is an input. To avoid contention issues with the Apple II motherboard during ph1, a separate enable signal (read_en_int) is the OR of read_en with ph2 inverse, and hence during a memory write only goes low during ph2. This internal enable signal controls the direction of the bidirectional pads for data, but is not externally accessible.

There are two types of I/Os that are not used in the final design that are implemented in the actual 650; those that can be considered as no connects, and those that must still be tied to either power or ground to send an appropriate signal to the Apple II motherboard. The SYNC signal (pin 7) is the only pad that is free to be used for another purpose. This is because pin 7 is not connected in the Apple II. Pins 2, 4 and 6 should be left floating with blank pads. Pin 2 originally was connected to the RDY signal. In the Apple II, this pin is tied to power, and also connects to the peripheral connecter. This means that if pin 2 were ever to be pulled low, all peripherals would be disconnected including the ROMs. Pins 4 and 6 are the IRQ and NMI pins in the Apple II. These signals are involved with the interrupts in the 6502 processor. Like pin 2, they are also tied to ground and the peripheral connector. It is possible that if the pin is used for another purpose, and an interrupt signal gets sent to the peripheral connector all of the information being written to the disk will be lost.

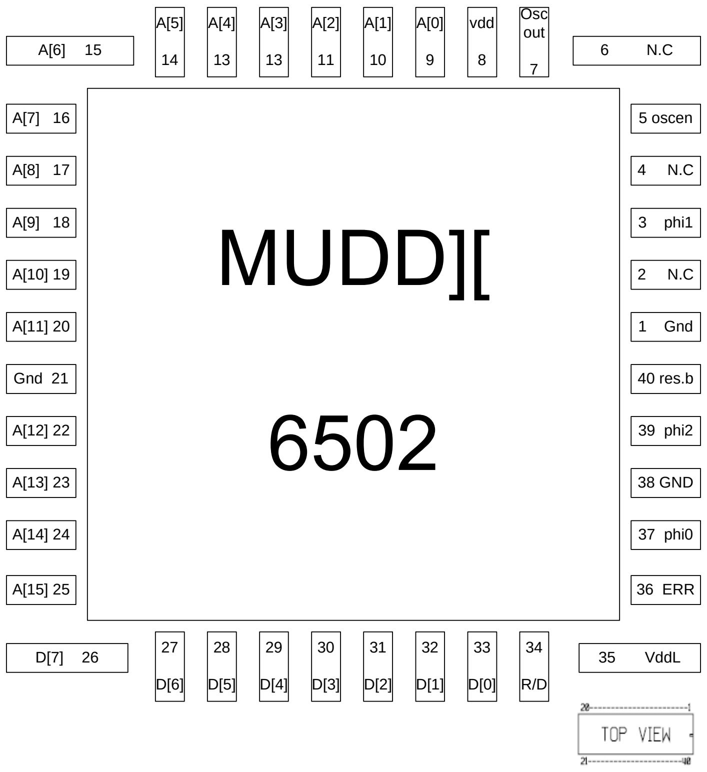


Figure 4 shows the bonding diagram, with the pins numbered and labeled

In this implementation of the 6502 there are two voltage sources used. The original 6502 chip ran from a five volt source. However, in order to save energy, this implementation obtains its core power from an external source that can be set lower than the original five volts. The rest of the Apple II must still run at five volts, so it is still necessary to provide the chip with a separate five volt I/O source so that it can output signals referencing this voltage. The five volt source is connected to pin 8 in the 40 pin DIP implementation from the Apple II power supply. The lower voltage source needs to be supplied externally. This is achieved by placing the 6502 chip in a special 40 pin socket with pins 35 and 21(power and ground) bent so that they can be soldered to wires and jumpered to an external source. For the ground connection, it should be noted, that the ground wire needs to be connected to both the power supply, and the body of the Apple II for the actual ground connection. First, connect the wire to the power supply. The wire should be long enough so that it can be screwed in at the base of the ground receptor. After this is done, a banana plug with a clip at one end can be attached to a suitable part of the computer. Make sure that the connection is with a metal plate of the outer shell, and not a plastic one. The socket can then be inserted into the computer.   In the modified socket, pin 36 (razor error) is also bent so that the razor latch errors can be monitored on an oscilloscope.

It is necessary that all of the signals leaving the chip use five volt signaling so that there are no indeterminate voltage levels being sent to other parts of the computer. To ensure of this, all of the output signals leaving the core are fed through voltage level converters before they enter the pads. These converters are located in the chip between the pad and the power line in the padframe.

The level converters used are standard CVSL style. In a level converter with input a and output y, there are two NMOS and two PMOS transistors. NMOS transistors with gate voltages of a and a_b (with respect to the lower core voltage) connect to PMOS transistors through the gate/drain. The model used here has been modified from the model in the text by choosing the opposite node to be the output, thus making the level converter inverting. An inverter was then added to the output of the level converter to help drive the pads. In the layout, the level converter had to be planned so that it would fit between the power and ground lines in the pad. To do this, the entire design has to be strung out so that it was limited to being at the most 3 metal 1 tracks tall. Also, the design used only polysilicon and metal 1 so that metals 2 and 3 could be used to cross over the level converters bringing the level converters' inputs and outputs.

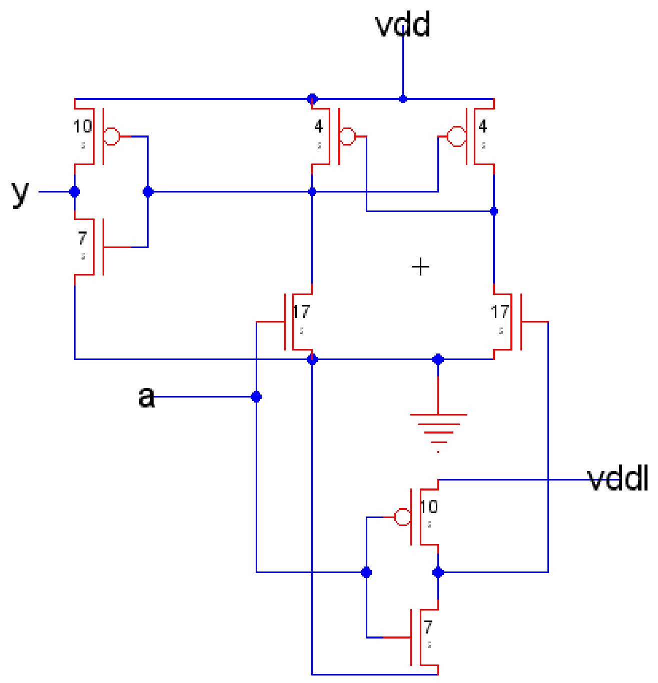


Figure 5 shows a schematic of the level converter used

The one potential pitfall in this design is that during a transition, it is possible for both the NMOS and PMOS in series between Vdd and ground in to be on (in Fig. 4, the transisters sized at 4 and 17). If there is a large voltage difference between the low and high levels, the PMOS transistor may take longer than one clock cycle to overpower the NMOS. In order to combat this problem, it is necessary to size the NMOS gates much larger than the PMOS. In this implementation, the NMOS are 17 λ wide and the PMOS are 4 λ wide. The below plot shows the relationship between propagation delay and the core VDD.

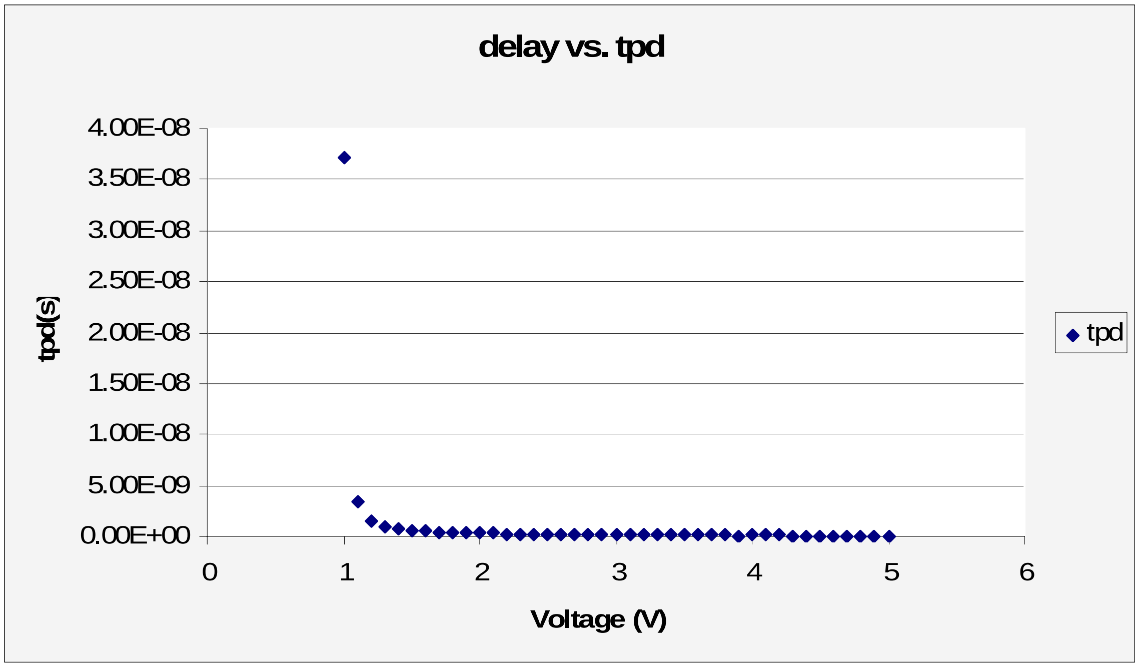


Figure 6 is a plot of propagation delay vs. core VDD for the level converter

The ring oscillator mentioned above was created to take advantage of the two pads that are not used at all by the Apple II or the razor latches. In the case that the 6502 chip does not work out of the box, one step in debugging would be to make sure that the padframe works, and that the correct voltage is being supplied. The ring oscillator takes an enable signal, which, if high, starts the 30 inverter ring oscillator. At a gate level, the ring oscillator is a nand gate taking in the enable signal and the output of the last inverter. The output of the nand gate goes into the first inverter.

The ring oscillator is meant to be tested while the chip is in the One Hot Logic chip tester (see test plan for details). If everything works correctly, osc_out (pin 7) can be hooked up to an oscilloscope, and the frequency of oscillation should correspond to the below plot comparing oscillator frequency to core voltage. This plot was created using an HSPICE model exported from the Electric schematic.

Floorplan

An initial floorplan was made by listing cells in the RTL datapath, adding widths of cells already in wordlib8, and estimating widths of new cells. The initial floorplan is shown in Figure 7. The chip is 1.5 mm x 1.5 mm or 5000 λ x 5000 λ. Padframe consumes 750 λ on each of the four sides, leaving 3500 λ x 3500 λ for core and routing. The controller was estimated by the ROM sizes. Routing of controll signals was not accuretly estimated.

The completed core consumed all space within the padframe. The datapath is 3257 λ x 1320 λ, which is nearly 800 λ wider than the initial floorplan. This discrepancy is because the width of ALU was overlooked in the preliminary floorplan, and is considerably more complex than previous MIPS chips, which new cells were estimated from. Controller is 3329 λ x 1807 λ due to more controller states than additionally planned, and more space needed for routing.

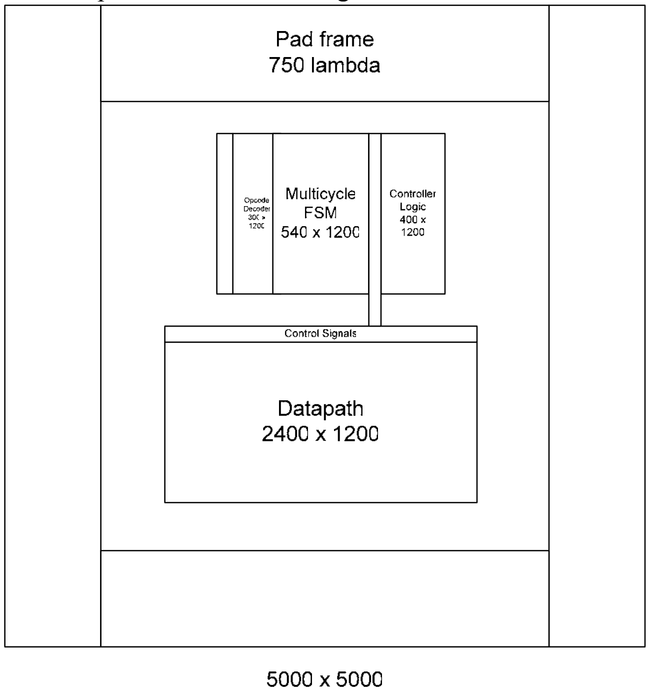

Pad frame
750 lambda
Opcode Decoder 300 x 1200
Multicycle FSM 540 x 1200
Controller Logic 400 x 1200
Control Signals
Datapath
2400 x 1200
5000 x 5000

Figure 7 Mudd II Floorplan

Figure 7 shows the slice plan of the Mudd II datapath. Vertical columns are word slices and horizontal lines are bit lines. In the figure, if a bit line terminates at a word slice, or contains a black dot in the word slice, it is physically connected to that word slice. Above each word slice name is the actual word slice width in vertical Metal-2 tracks, which are on an 8 λ pitch. This helped in calculating the datapath's width for later revisions of the floorplan.

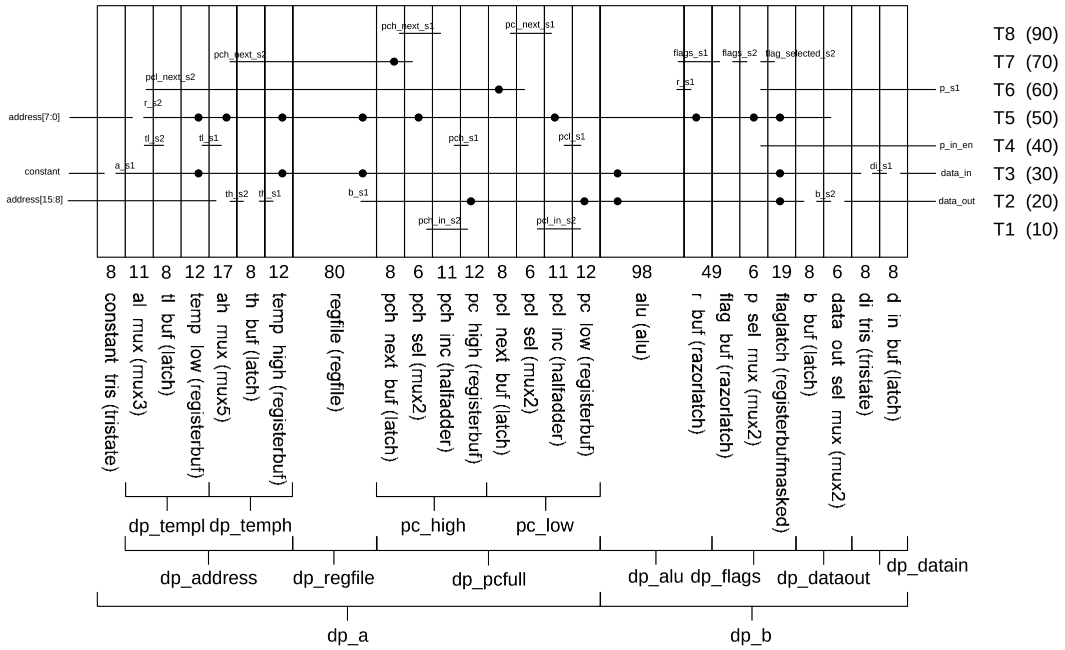


Figure 8. Mudd II Slice Plan

Bit lines are on a vertical 10 λ pitch starting from the cell origin. Since cells in the HMC standard cell library are 90 λ tall, including power and ground lines, there are seven bit line tracks within each cell and one track above. Bit line tracks are indicated by T1 to T8 in the figure, with λ with from the origin denoted in the parentheses. Power and ground tracks (80 λ and 10 λ from origin, respectively) and are not included in the slice plan and reserved for power/ground only. Additionally, there is a bit line track between each cell in a word slice (at 90 λ). This was not included in the sliceplan, and is reserved for use within individual word slices. Metal-2 vertical tracks were not included in the slice plan.

Timing

The Mudd II uses two-phase clocks and transparent latches because that was the design of the original 6502 and because the memory system operates on two phases. It implements nonoverlapping clocks in order to avoid hold time errors or creating a race situation where data gets let through one latch and speeds through the second latch before it closes. Figure 9 shows the critical paths through the controller, datapath, and external memory.

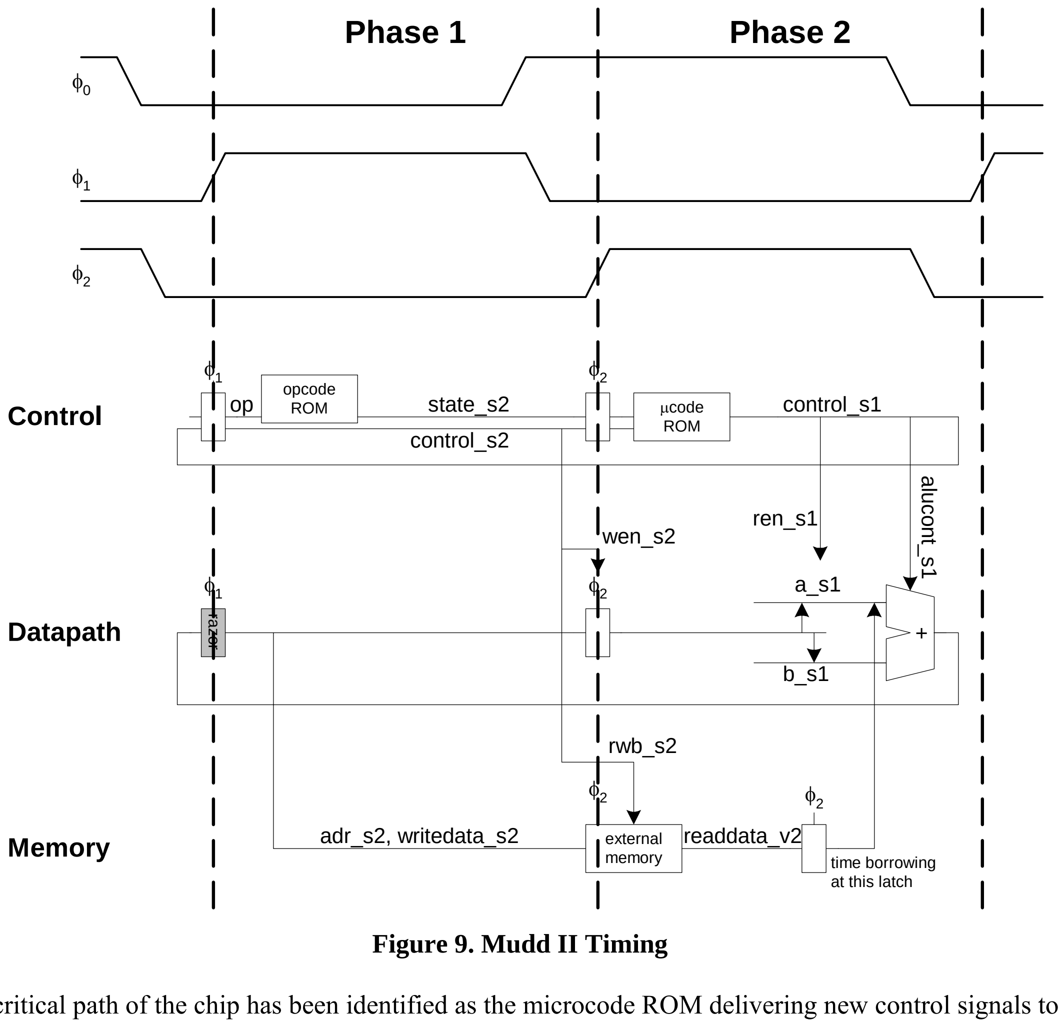


Figure 9. Mudd II Timing

The critical path of the chip has been identified as the microcode ROM delivering new control signals to the ALU, which then must compute the desired function and output it to the results signal in time for the next clock cycle. The reason that this is the critical path of the chip is because the ALU is expected to be the slowest component on the chip, and it can receive new control signals at any time during phase 2, and needs to compute the outputs in time for phase 1.

A single clock, ϕ_0, is provided to the Mudd II. It internally creates the nonoverlapping clocks ϕ_1 and ϕ_2, where ϕ_1 is 180 degrees out of phase with ϕ_0 and ϕ_2 is in phase with ϕ_0. Since the clock generator operates at the core voltage, it has an inherent delay and nonoverlap that tracks with the core voltage. The final version of the clock generator can be seen in Figure 10. The two-phase clocks created using the generator are expected to have a nonoverlap equal to approximately 2.5 FO4 delays at each voltage. This strategy works well for creating the clocks to run the core of the chip, but it creates issues when the generated clocks are also being used to synchronize with other parts of the computer. The specifications that were most concerning were the ones on when ϕ_1 and ϕ_2 could rise/fall in relation to ϕ_0 rising/falling. For simplicity of testing, we took the smallest maximum delay, 65 ns, and used it as the standard between all of them. We ensured that the delays from the input pads to the output pads between rising and falling edges of the clocks were no more than 65 ns all the way down to 1V. Waveforms of the clock being generated at 1V core voltage and the two-phase clocks overlaid on top of each other at both core and external voltage can be seen in Figure 11 and Figure 12. In this case, the maximum delay between a change in the input clock and the corresponding change in the output clock is 65 ns, just barely meeting the specification.

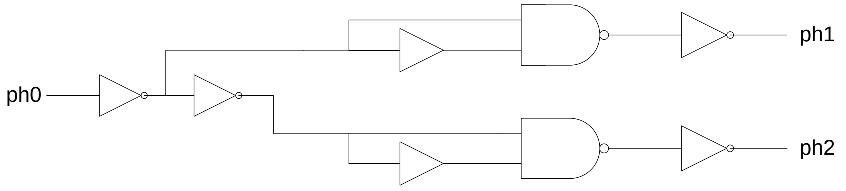


Figure 10. Schematic of the two-phase nonoverlapping clock generator

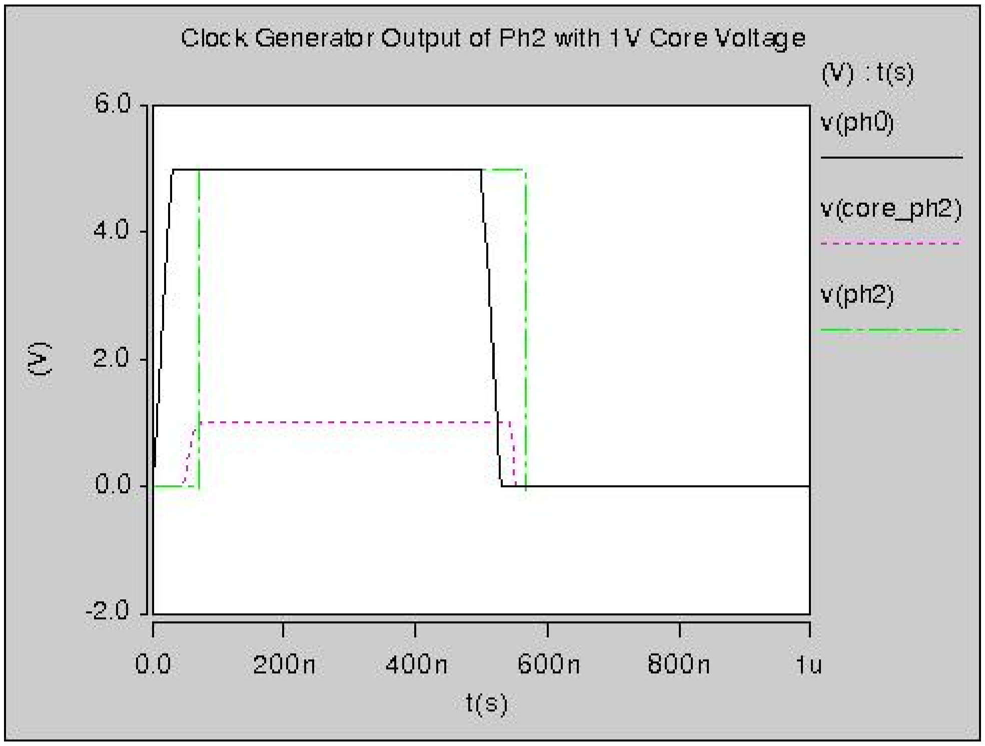


Figure 11. Waveform of ph2 at core and output voltages

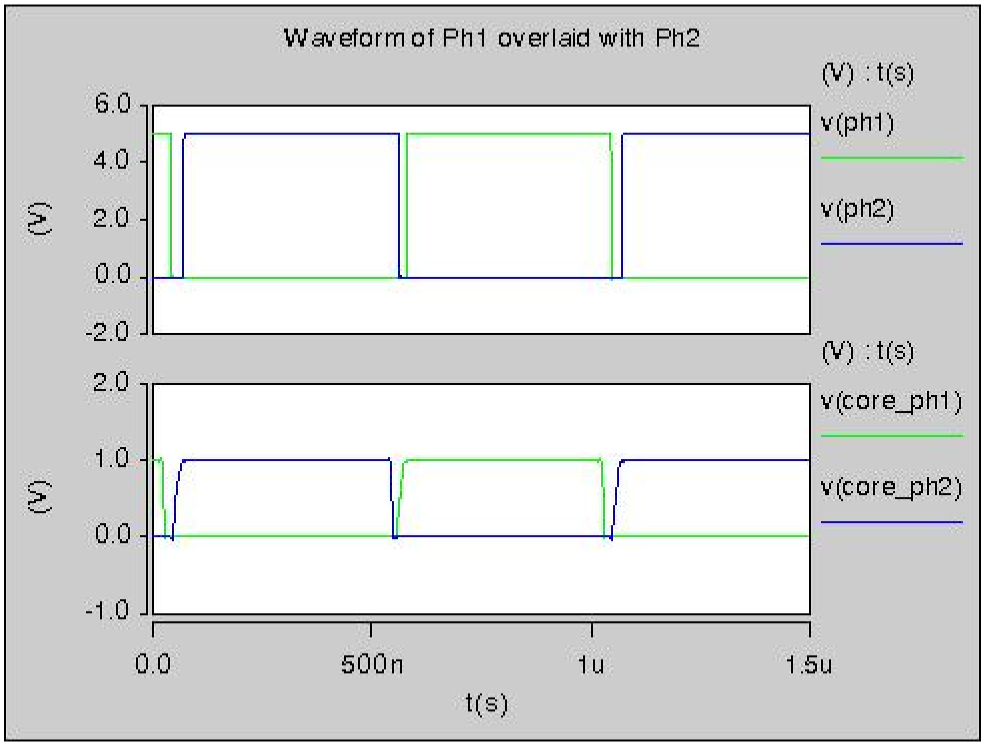


Figure 12. Waveform of ph1 and ph2 overlaid displaying nonoverlap

A Razor latch is a type of latch that has the ability either to detect when signal failure has occurred or to detect when signal failure is about to occur. The method used in this chip consists of two latches, as can be seen in Figure 13. One which is used only for checking the output and the other, which is used to drive the output signal to the next stage. The data latch is run on a slightly delayed clock while the check latch is run on an on-time clock so that the Razor Latch will determine when the input signal is just barely arriving before the main latch goes opaque. The data stored in these latches is compared using an XOR and the result is a signal that indicates if there is an error. This allows the latch to detect when the data changes right before the falling edge of the late clock because the data stored in the two latches will be different. The error signals of all the Razor latches are then combined using an OR tree to create a single error signal for the whole system. This signal is latched on the other clock so that it does not glitch due to transient differences but only asserts when the latches hold different values while opaque. The error detection window is characterized by the time between when the error signal goes high and the output signal stops making it through the latch. Simulations have determined this window to be about 3 FO4 delays, and an example of the razor latch generating an error signal before signal failure can be seen in Figure 14. This use of the late clock reduces the amount of time that the next stage has to propagate but, as we can see in the timing diagram, the next stage is fairly simple and won’t need too much time.

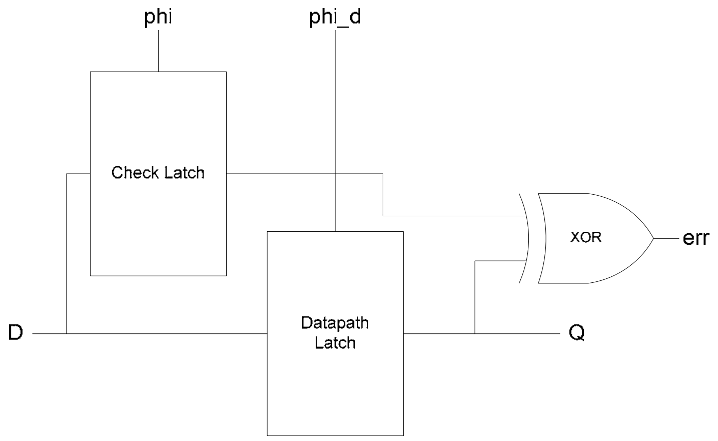


Figure 13. High level schematic of a Razor Latch

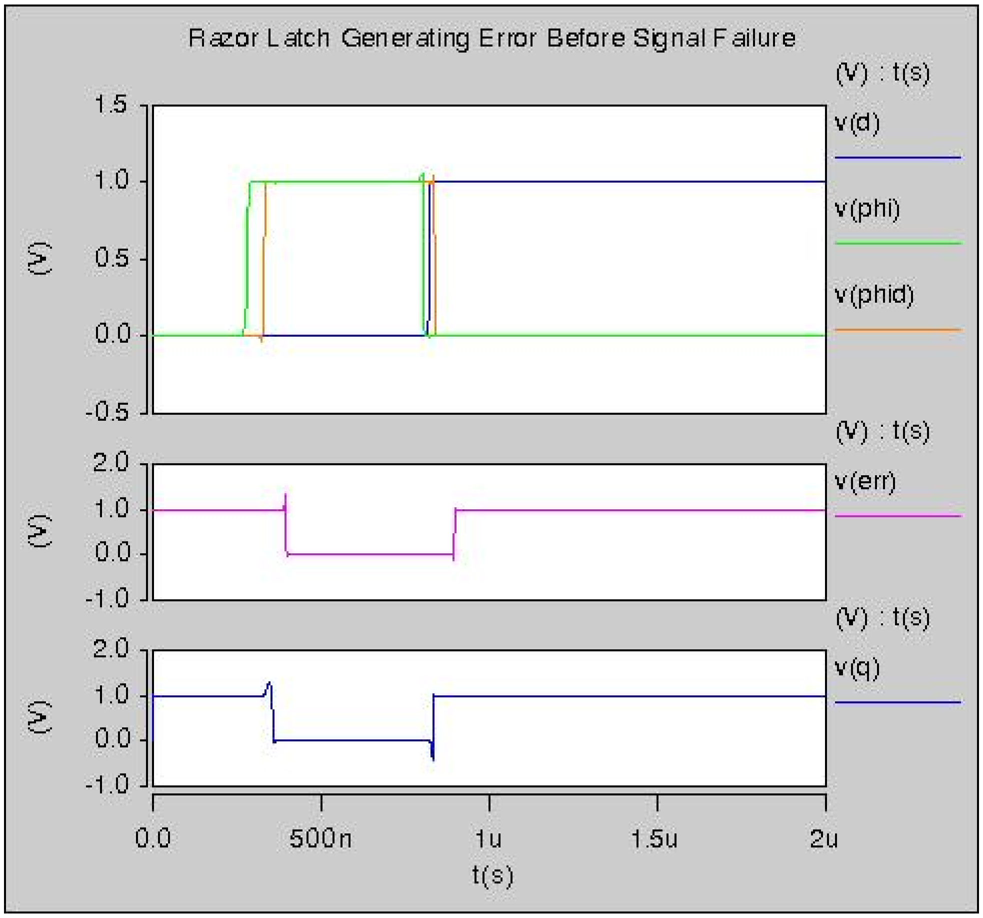


Figure 14. Demonstration of the Razor Latch generating an error before failure at 1V in simulation.

There are two sets of 8-bit Razor latches in the chip. They have been placed on the results and flags outputs of the ALU because the ALU is the slowest part of the datapath. By putting the Razor latches on these signals from the ALU, the team hopes to catch errors before they occur. This will allow the team to implement significant time borrowing into the next stage without leaving large safety margins due to uncertainties in manufacturing and temperature. In addition, the voltage level can be dynamically adjusted to minimize power while barely avoiding signal failure.

Design Decisions

The layout took its direction from following the specifications on the schematics, as laid out through the RTL. After the schematics were passed down to the layout team, it was usually straightforward to organize the layout so the individual gates and transistors conform to the slice plan. Main considerations included designing the layout to conserve the number of vertical metal tracks, minimize spacing, share power and ground connections and permit simple, elegant wiring. The tradeoff was between optimizing power consumption, speed, and area on the chip and man-hours required to do so efficiently. If it was achieved, it allowed for extra room for emergency wiring when connecting the cells in higher levels became more complicated. In a few cases, portions of the layout were moved into the zipper area to keep the datapath width as small as possible.

Most of the lower level layout designs did not change, but all components were subject to change if the RTL was updated or the schematics were reorganized. Changes did get pushed back from the layout team to the schematics team when more hierarchy was needed to allow for a simpler validation process or when components needed to move from the datapath to the controller or zipper to fit in the floorplan. The layout team requested several changes in the schematics with ALU and the controller during the design process. The most challenging changes occurred when the RTL was changed after schematics and layouts had already been completed.

The following sections included particular challenges in the layout design process:

ALU architecture

As written in the RTL, the ALU was designed to perform a subtract-and-carry operation as well as the decrement function which led to adding extra hardware. The RTL defined the decrement function as *a - c_in*, but subtract as *b – a*. It would have been desirable to save hardware by changing subtract to *a-b*, but this was not possible due to the requirements of the Apple II architecture.

The ALU was one of the hardest components to fit into its allotted space in the floorplan. Originally it incorporated a ten-input multiplexer, with each input being nine bits in length. Because this was a complicated design, the mux was simplified and split into its component tristate buffers to better fit into the layout for the datapath.

Ripple carry adder

In the original plan, the adder for the ALU was to be a carry lookahead adder. However, a carry lookahead adder would require a significant amount of hardware and would not significantly increase the ALU speed. The design was revised to incorporate a ripple carry adder because it called for considerably less hardware.

Synchronous reset

The latches were designed using a synchronous reset. This meant that they would only reset when both clock and reset were high. Firstly, less hardware was required to build a synchronous reset instead of an asynchronous reset; the latter would involve allowing for it to be reset at any time. Second, a synchronous reset prevents an unstable reset signal from accidentally resetting the latches when they are not allowing inputs, i.e. while clock is low. While this was a very unlikely scenario, we chose to use the safer option. Finally, the reset functionality of the latches would only be invoked during the initial startup of the chip. Thus, since reset would be held high for several cycles, it was not necessary to add the extra functionality of an asynchronous reset.

Clock enables for power savings

Several latches incorporated clock enables. Not only was this a safeguard so that they would only be written onto when desired, but it also saves energy. Since the latch is fed a gated clock, the AND conjunction of clock and enable, the latch will only be affected by a high clock when enable is also high. Since the latches will not respond to changes in the input when they are disabled, the only energy loss will come from static energy dissipation due to leakage in the transistors. If the latches were active when they could have been disabled, a change in the inputs would result in a change in the outputs which creates dynamic energy loss.

ROM layout style with pseudo-nMOS

Because of the large number of states in the controller FSM, a pseudo-nMOS NOR ROM layout was chosen to conserve space. Layouts and schematics for the ROM were generated using a tool developed by the design team early in the project. While using pseudo-nMOS for the ROM is a significant power cost, removing pMOS from the logic of the array saves considerable space and delay over a CMOS layout, without introducing the timing challenges of dynamic logic into the chip. Because the state ROM is also in the critical path (along with the ALU), the team performed simulation in HSpice to characterize the timing and determine if predecoding would be necessary. Simulation found the state ROM had 150ns of delay at 1.5V using a simple 8-input NOR decoder, which is significantly less than the half-cycle(500ns) performance requirement, so the team did not pursue predecoding.

Carry select signal

The carry select signal is generated inside the datapath from the op-code and the carry_in. It is then fed into the ALU. The carry select signal is already conditioned based on the operation being performed eg. subtraction, and therefore the ALU does not need to incorporate additional hardware to deal with the carry for the subtract command. The generation of the carry_select signal is placed in the zipper of the datapath, so it doesn't add any additional horizontal length to the overall size of the datapath as well, very important considering the size constraints of the project.

Register design (regbufs)

The register file layout design conserved space by separating the bitlines in the datapath portion of the cell from the control logic and moving the control portion into the zipper. This kept the register layouts narrow and small enough to fit into the floorplan of the datapath.

Validation

The Mudd II was validated at the RTL, schematic, and layout levels.

The RTL logic validation included regression suites P, A, and R. Suite P is a short test using 12 instructions to prove preliminary functionality. It is also used with HSPICE to estimate power consumption. Suite A test all of the instructions except BCD arithmetic. Suite R involves booting the Apple II ROM in simulation to get to the command prompt.

The chip passed suites P and A at the schematic level. Suite R was not able to be completed before tapeout.

The schematic verification involved running all of the RTL regression suites on transistor-level netlists of `core` and `chip`. The schematic requires two-phase nonoverlaping clocks for correct operation of the register file. However, the netlist does not model delays, resulting in incorrect simulation. To solve this, the netlist for the clock generator had to be modified manually. The buffers used in the clock generator were replaced by the Verilog statement:

```
assign #1 y = a;
```

so that they would actually produce a delay, which would ensure that the phases of the generated clock do not overlap.

The schematic was also simulated in HSPICE. The `vcd2spice` script was used to generate SPICE digital test vectors from a Verilog Change Dump record of regression Suite P. A limitation of SPICE's digital test vector format is its bidirectional signal capabilities. Bidirectional enable signals can only be inputs for SPICE to properly drive inputs and check outputs. If the enable signal is an output, contention will occur. Because `read_en` is an output from `chip`, `data` output cannot be validated against Suite P vectors. When simulating chip, `data` was set to high impedenace (Z) during memory write to avoid contention and accurately estimate power. Since `core` does not have a bidirectional data bus, it was simulated and validated in SPICE independent of `chip` to check data_out. The `chip` schematic and layout port names did not match the RTL (e.g. a instead of address, d instead of data). Thus, the port names in the digital test vectors file had to be changed to match `chip` schematic. Simulating `chip` schematic in SPICE, with vectors from Suite P, requires 2 hours of run-time and found no errors.

Schematics were simulated in IRSIM against Suite A. IRSIM .cmd were generated from Suite A's Verilog Change Dump file using `vcd2cmd`. As with the HSPICE simulations, port names had to be changed to match schematics. A small Unix SED script was used to convert the names, and is included in the appendix. Core voltage (vdd) is assumed to be HIGH in IRSIM. However, the padframe voltage (padvdd) had to be added manually to the top of the .cmd file, using the command

```
h padvdd
```

The IRSIM .cmd file was then read into Electric's built-in IRSIM simulator and the stimulus was read from disk. Simulation passed without errors.

The layout verification included running DRC, ERC, and NCC hierarchically on `chip`. The library was first checked for validity. Then the DRC dates were cleared. DRC was checked hierarchically, which took 15 minutes to complete. No DRC errors were found, but there were 84 DRC warnings. These warnings were all unrouted arcs in padframe layout. Unrouted arcs are due to auto-route of padframe dvdd and dgnd, with unrouted arc selected as arc type. Since autoroute would not complete without dvdd and dgnd exports touching, there should be no breaks in dvdd and dgnd rails. A visual inspection confirmed this, so we ignored warnings instead of manually replacing the unrouted arcs with metal-2.

NCC was checked using Hierarchical Comparison and Check Transistor Sizes selected and "Don't recheck cells that have passed in this Electric run" unselected. NCC was also checked with Flat Comparison and passed.

Wells were checked with ERC. The farthest distance from a well contact was reported to be 873 λ for P-well and 1164 λ for N-well. These distances are high, and could cause latchup in the chip and cause traces to melt. If latchup happens, core voltage should be decreased, since there is a quadratic relationship of voltage on current. These distances are in the controller and opcode ROMs. The ROM generator should be revised in future projects to add more well contacts and prevent risk of latchup.

A Verilog deck was generated from the layout and `chip` was validated against the RTL regressions tests. Running Suite A on the chip layout in ModelSim took approximately 40 seconds. IRSIM simulations of Suite A were repeated on layout with the same .cmd file, and required about 10 minutes. Finally, `chip` layout was simulated in HSPICE with the digital test vectors generated from Suite P. Simulating chip layout in SPICE did not have DC conferegence at 70 F. When simulation temperature was increased to 180 F, DC convergence was found and simulation passed with no errors.

Power Consumption

The HSPICE simulations of the `chip` schematic found a core power consumption of 12 mW at 5 V and 46 μW at 1.5 V. No data was recorded from layout simulations, but the wire capacitance is expected to increase power consumption. There is no explanation of why power consumption drops by more than a quadratic factor at low voltage. The chip schematic did not simulate correctly in HSPICE at 1.0 V, so the minimum operating voltage is believed to be between 1.0 and 1.5 V. Added capacitance from wires in layout could increase the minimum operation voltage above 1.5 V. Razor operation at the edge of failure has not been demonstrated in simulation.

In comparison, the original NMOS 6502 was rated for 700 mW. The savings comes from using CMOS in place of NMOS, process scaling from 6 microns to 0.5 microns, and reducing the core voltage.

Tapeout

After verification, the chip was exported to CIF as chip.cif. The MOSIS CRC given by Electric 8.07e was 268016765 5677896. This did not match the checksum by MOSIS (or mosiscrc.exe), which is actually 2206588185 5430278. We think Electric is generating a bad checksum and we used the checksum from mosiscrc.exe.

The chip was sent to MOSIS for fabrication on May 13, 2008, and is scheduled for fabrication on May 19, 2008 AMI 0.5 μm run through MOSIS. The design name was mudd2and the design password was the same. The design ID is 79499.

The following files are needed to reconstruct the design. They are available in the \\charlie.hmc.edu\Courses\Engineering\E158\proj2_2008 directory. Contained in this directory are:

• muddlib07.jelib:	Muddlib
• wordlib8.jelib:	Wordlib
• Alu_support.jelib:	Support cells for the ALU
• logo.jelib:	Logo on chip
• razorlib.jelib:	Razor latches
• muddpads14_ami05.jelib:	I/O pads
• 6502opcode.jelib:	Opcode ROM
• 6502controller.jelib:	Controller ROM
• 6502Layouts.jelib:	Schematics and layouts of 6502
• Mosisrc.exe:	DOS program to generate MOSIS checksum from .cif file
• Chip.cif:	CIF file of chip
• outSuiteP.cmd:	IRSIM .cmd file containing vectors from Suite P
• outSuiteA.cmd:	IRSIM .cmd file containing vectors from Suite A
• electric-8.07e.jar:	Electric version used for IRSIM simulations
• 6502.arr:	Padframe generation file
• 1.5v/:	HSPICE stimulus and output of chip schematic with 1.5 V core voltage
• 5.0v/:	HSPICE stimulus output of chip schematic with 5.0 V core voltage
• RTL/:	RTL used to behaviorally validate chip schematic and layout

Also included in the \\charlie.hmc.edu\Courses\Engineering\E158\ directory are:

• Final Chip/:	The completed chip, as described above.
• Presentation/:	Spring 2008 end-of-semseter presentation
• Razorsims/:	Simulations of Razor latches in HSPICE
• ROM Generator/:	Rom generator scripts
• Vcd2/:	vcd2sp/vcd2cmd perl scripts
• Working Files/:	Intermediate files
• 6502chipreport.doc:	This file
• 6502.cif.pdf:	PDF of chip layout
• sy6502.pdf:	Synertek 6502 Datasheet

The RTL is also stored on the Google Codes hmc-6502 website at http://code.google.com/p/hmc-6502/. The code may be retrieved using Subversion (on Linux, and Mac OS X) or TortoiseSVN on Windows. http://code.google.com/p/hmc-6502/source/checkout gives the location of the source respository and how to check it out.

Test Plan

When the chip returns from fabrication in the fall, the following members of the team have signed up for indepdendent study to test the chip:

Michael Braly
Tony Evans
Sam Gordon
Steve Huntzicker
Kyle Marsh
Trevin Murikami

The objectives in testing the Mudd II chip are:

1. Validate or locate problem(s) with the chip
2. Determine the lowest possible voltage at which the chip will consistently run
3. Obtain Frequency vs. voltage plot for ring oscillator
    a. Compare with expected results
4. Obtain Power vs. VDD plot
5. Travel to Oregon without dying

The three functionality tests covered in the document are:

I. Running the chip in an Apple IIe, and verifying functionality
II. Testing the chip with a representative set of IRSIM test vectors
III. Testing inputs/outputs of the chip

The first test that should be run on the chip is Test I. If the Mudd II passes this test, Test II and III can be administered for assurance sake and for obtaining plots and Vdd (core voltage) values; however, you can breathe easily- the chip works. If Test I fails, and the computer does not boot, the IRSIM test vectors should be used. Use these results to attempt to track down the specific problem in the chip. If these tests are inconclusive, or if there is suspicion in the padframe or power/ground signals, the last test should be run.

After the chip is working, Test I can be run at increasingly lower voltages, until the razor latch pin (36) or any other part of the chip fails. Spice predicts that the level converters will fail at 1.0 V while the razor latches will still be functional. The razor error output should be the first component on the chip to fail. If this is not the case, it is worth looking into the reason for this. One possible explanation would be that the level converts are actually not as fast as the spice simulation predicts. One way to test the level converters if this problem is encountered would be to run the ring oscillator part of Test III, and check that the oscillation matches what the provided table predicts. It is very likely that the numbers will not correspond exactly, but if there is a large difference, it is worth taking note of. The main reasons that the ring oscillator would give a different oscillation are that the transistor speed is faster or slower than the spice simulation accounted for, or that the pads are not receiving the expected voltage. The latter reason can be checked by the second part of Test III. If the two voltage levels are shorted, the correct numbers will not be observed. Once the chip passes Test II at the lower voltage, Test I should be repeated one last time for verification.

After all the tests are working for the lowest achievable voltage, the power of the chip should be measured. This can be done by connecting an ammeter in parallel with the core power supply. The easiest way to do this is to use banana/banana cables and plug one side of each into the same power and ground ports in the power supply that the chip is using. Then, plug the other ends into the ammeter. This will measure the current through the chip. Take a current reading while the chip is on in the Apple II. Once the current is determined, the power can be found by the equation P=IV, where I is the current through the chip and V is the voltage.

Supplies for all tests

1. TestosterICs brain box
2. TestosterICs 40 pin DIP DUT board
3. Pin electronics adapter (5 volt)
4. Null modem serial cable (D-Sub 9 pin female to D-Sub 9 pin female)
5. TestosterICs Chip Tester Interconnect Cable (HD D-Sub 15 pin male to HD DSub15 pin male)
6. AC adapter (5V DC output)
7. Jumpers for TestisterICs pins
8. Mudd II chip
9. Analog/digital oscilloscope
10. BNC oscilloscope cable
11. Apple II computer
12. 40 pin DIP socket
13. Sheathed wire
14. Variable power supply
15. Ammeter
16. Banana/Alligator cable for power supply

Test I "Million dollar test"
In this test the Mudd II chip is put into the Apple II to check functionality

Setup
1. Bend pins 35, 21, and 36 out of socket so that they are perpendicular to the other pins
2. Solder a length of wire to the pins with both ends stripped

Procedure
1. Place 6502 chip in socket
2. Place socket in AppleII
3. Connect wires from pin 21 and 35 of socket to power supply (ground, power respectively)
    a. Make sure that the power supply is off when connecting
    b. Use banana/alligator lead
    c. Set power supply to 5V.
4. Insert Oregon Trail floppy in drive 1
5. Cross Fingers
6. Turn on computer and power supply
7. Make sure that the game boots and functions properly
    a. Hint: don't ford the river

Oscilloscope instructions for detecting razor errors.
1. Connect a digital oscilloscope lead to the wire soldered to pin 36.
2. Set the oscilloscope to trigger once on a rising edge of the connected lead

If the oscilloscope is ever triggered, that means that there was a razor error, and the input voltage is too low. If the voltage is still set to the maximum five volts, there is another more sinister problem.

Test II "Diagnostic test"
This test uses IRSIM test vectors to test the functionality of the Mudd II

Setup
1. Connect the brain box to serial port of Windows computer

2. Connect 5 volt pin electronics adapter to DUT board
3. Connect DUT board to the brain box using a TestosterICs interconnect cable
4. Connect AC adapter to brain box and 120V power outlet
5. Set 2 external power supplies to 5 V and turn off
    a. Only need one if testing core at 5 volts
6. Connect core power supply to pin 8 in 40-pin DIP DUT board
7. Connect pad power supply to pin 35 in 40 pin DIP DUT board
8. Connect ground to pins 1,21 and 38.
9. Turn on the brain box and note that amber and green leds on the brain box turn on. Amber indicates that the brain box is on and connected and green indicates that it is safe to remove the DUT.
10. Put on chip
    a. Note that pin one is in the standard top left corner of the chip
11. Turn on external power

Procedure

1. Test chip with IRSIM vectors
    a. Navigate to TesterGen directory on lab computer and type “TesterGen”
    b. Type “selftest”
        i. This checks that there is no device in the socket
    c. Type “@ examples/OutSuiteP.cmd” to run Suite A+ test

Test III: The padframe/power test

This test checks that the chip is receiving and outputting the proper power and ground signals and that the padframe works.

Setup

1. Same setup as Test II.

Procedure

1. Test power and ground
    a. Ensure that there is a large resistance between power and ground, and low resistance between ground pins
        i. Vddh: pad 8
        ii. Vdd: pad 35
        iii. Gnd: pad 1,21,38
2. Test ring oscillator
    a. Restart TesterGen
    b. Type “selftest”
    c. Type “@ examples/ring_osc.cmd” to run ring oscillator test
    d. Oscilloscope should produce oscillations that match the graph and table below on pin 7.

Ring Oscillator Spice Test Results:

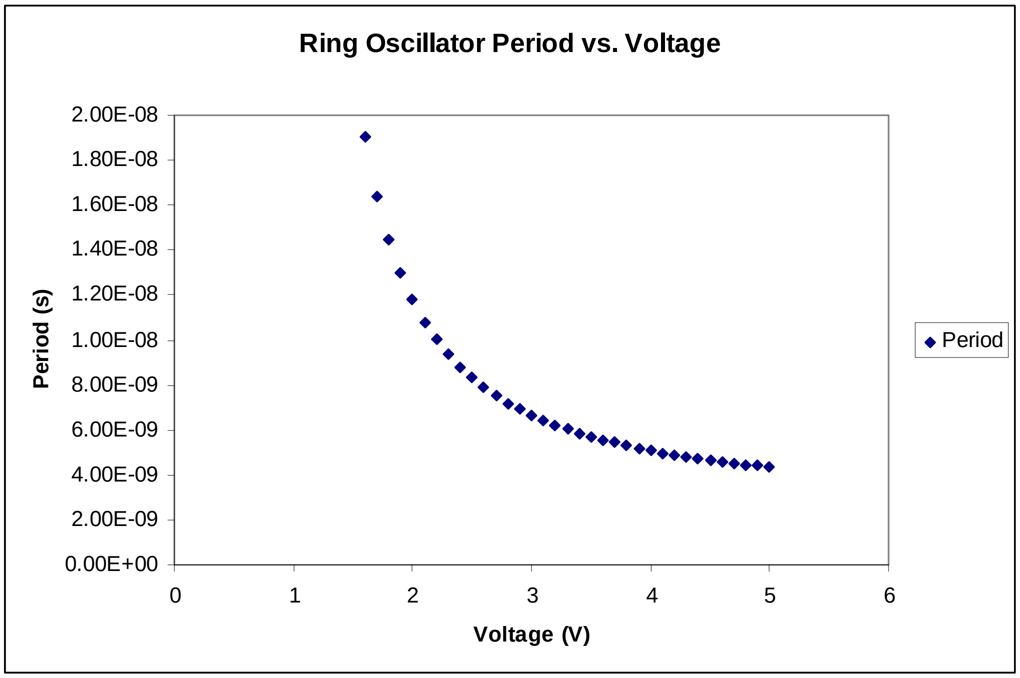


Figure 15 Graph of ring oscillator Period vs. Voltage

V	Period	V	Period
1.6	1.90E-08	3.9	5.19E-09
1.7	1.64E-08	4	5.08E-09
1.8	1.45E-08	4.1	4.98E-09
1.9	1.30E-08	4.2	4.89E-09
2	1.18E-08	4.3	4.80E-09
2.1	1.08E-08	4.4	4.73E-09
2.2	1.00E-08	4.5	4.65E-09
2.3	9.35E-09	4.6	4.58E-09
2.4	8.79E-09	4.7	4.52E-09
2.5	8.31E-09	4.8	4.45E-09
2.6	7.88E-09	4.9	4.39E-09
2.7	7.52E-09	5	4.34E-09
2.8	7.19E-09		
2.9	6.91E-09		
3	6.66E-09		
3.1	6.43E-09		
3.2	6.22E-09		
3.3	6.03E-09		
3.4	5.85E-09		
3.5	5.70E-09		
3.6	5.55E-09		
3.7	5.43E-09		
3.8	5.30E-09		

Summary

This report describes the design of a 6502 microprocessor, including RTL, schematics, and layout. The design includes Razor latches, used to detect bit errors and help reduce power consumption. The project included the development of a ROM generator to automatically generated controller ROM layouts.

We have behaviorally validated the RTL, schematics, and layout against two regression suites. Schematics and layout have been simulated in IRISIM against test vectors. Schematics and layouts have also been simulated in HSPICE using the BSIM model parameters for the AMI 0.5μ process. All simulations passed, except HSPICE simulation of chip layout did not find DC convergence at room temperature. DC convergence was found at 180 F. The HSPICE simulations of chip schematic found a core power consumption of 12 mW at 5 V and 46 μW at 1.5 V. Since this was schematics, and did not account for wire capacitance, actual power consumption should be higher. Since we were unable to simulate layout in HSPICE at room temperature, there is a possibility that the design will not work at room temperature (e.g. due to hold time violations).

Electric's ERC found the farthest distance from a well contact to be 873 λ for P-well and 1164 λ for N-well. This may cause latchup and melt the chip. Lowering core voltage will quadratically reduce the current, and reduce heat generated. These distances were in the automatically generated controller ROMs, and the ROM generation code should be updated to add more well contacts.

Suite R was never complete, thus the design was never validated against the Apple II Boot ROM. It is possible that the chip will not boot the Apple II, and thus will need to be fixed and fabricated.

Lessons

This section concludes with some of the major lessons that each of the team members take away from the project, along with helpful hints they would have liked to know before they started.

Workplan

The microarchitects were optimistic with regards to their original workplan. The milestones for the first week consisted of writing Suite A and Suite P tests and debugging the microcode until the RTL passed. This actually took several weeks to complete. One reason for this is that, at the time, very little microcode had been written and the opcode ROM was not written nor could it be automatically generated. The RTL first passed Suite P in the third week of the project and Suite A around the fifth week. Additionally, the RTL was planned to pass Suite R in the third week, however, Suite R never passed in simulation.

Microarchitecture

The chief microarchitect had done a great deal of work before the class joined the project. The biggest challenge faced by the rest of microarchitecture team was catching up on what had already been accomplished and understanding what still needed to be implemented. Poor communication within the team resulted in assumptions that more of the microarchitecture was complete than was the case.

One important lesson learned was that following good practices, such as keeping up-to-date comments and avoiding magic numbers in code, will save a lot of time and effort. Additionally, it is important to not be afraid to take responsibility for work that urgently needs to get done, even if you don't feel most qualified.

Schematics and Simulation

An important aspect of building schematics is to design with layout in mind. Several schematics had to be redone or reorganized because they would have been very difficult to lay out in their initial forms. This includes developing a hierarchy for complicated parts of the chip such as the datapath or register file.

Furthermore, the hierarchy needs to match the slice plan to ensure that everything will fit in the required space.

Dependencies were the largest problem on the project. The schematics were heavily dependent on the microarchitecture, both for functionality and testing. Even late in the project, new tests and functions were added and things needed to be changed as a result. The schematics also have responsibility to the layout team, which is dependent on them. Layout done on out-of-date schematics is useless. Luckily, the final layout had only minor inconsistencies with the final schematics. Overall the lesson is frequent and effective communication between teams.

Even within a team, version control is essential. We were able to get by using a simple change log, but this was inconvenient and caused minor confusion and overwrites. Version control solutions are readily available, so there's no reason not to use a robust form of version control on a large-scale project of this nature.

Debugging of simulation results went much more smoothly when the microcode team members were involved, as they knew the most about what the tests were doing and could help pinpoint error sources more quickly. If physical cooperation is not feasible, the test authors should provide the people running the test with some quick documentation for guidance in debugging.

Physical Design

Throughout the project, all layouts had to be revised anytime there was a change made in either the RTL or the schematics. The project would have run more efficiently if elements of the design process such as the microcode and schematics were finished and frozen earlier in the semester. It is difficult to make a final layout without the reassurance that the schematics pass all the necessary tests. Many man-hours were spent making and revising layouts when the schematics changed several times over a week.

It is extremely important to create a slice plan for the controller as well as datapath at the beginning of the project. Then, once the building block cells are built, the control, input, and output signals can be placed to follow the flow of the signals in the slice plan. In the same way, it is helpful for the layout designers of the controller and datapath to map exactly how to connect signals between the two components. In general it is much easier to wire cells together when the inputs and outputs are pitch matched on the layout grid.

When working with any new software, there is a steep learning curve towards understanding its full potential and capability. It would be extremely beneficial if basic techniques covered in class labs like arraying and staying on the correct lambda pitch are reviewed before the project. In addition, the layout team should remember what Mosis rules imply before beginning any work. Distributing a list of the frequently used functions and preference and properties selections would save time over searching blindly for the right option.

A better way to manage the current state of the project needs to be devised. Many problems could be avoided if all team members knew exactly which schematics or layouts were the most recent, when they changed, why they were changed and who the last author was. It would be most convenient if this could be managed through Electric, but if it is not possible another method should be discussed. This would allow all the teams using electric libraries know what is current and more easily integrate their work into a common library or set of libraries.

The various design rules for the project, in addition to lessons learned from previous teams, should be more clearly available and/or presented to the class as a whole or at least to the various teams in particular. Passing information from year to year will help avoid common issues and would ideally generate a common repository for knowledge and advice on fixing problems to make things easier for future VLSI students or users of Electric, HSPICE, and Modelsim.

I/O

One problem that the I/O team had was keeping a big picture understanding of what was going on in the rest of the chip, and how the level converters and padframe fit into the system as a whole. There were a few instances, such as what voltage level to use in different parts of the padframe, that the way that the I/O team originally created the circuit looked perfectally fine, but when integrated with the rest of the chip, it was clear that it needed a different source.

The I/O team did not have too many software issues. The few things that did come up were firstly, that for situations where transistors need to be week, so that an output can be pulled low, the transistors in the layout and the schematic should be marked "weak" is electric. Secondly, Steve had issues with using Electric on his own personal computer. If you plan to use Electric on a non-lab PC, make sure that the directory that is being written to is mapped as a drive on the computer. It is also a good idea to regularly sinc your work with the common class library if you are working from a local copy. Even if you are using the class library, you should keep a local copy in case the class copy become corrupt.

Clocking and Latching

The Razor team was involved in the most research oriented portion of the chip. With that said, the design of the razor latches often changed from week to week. While this was sometimes frustrating for the team because they were building the latches on the transistor level, each change represented a significant improvement in the design.

The team also took on the compilation of this Final Chip Report, by far their most difficult task. They learned that the best possible way to understand the edits that people are making is by using Microsoft Word's "Track Changes" option. In the future, Chip Report editors should make change tracking mandatory rather than just suggesting it, and refuse to accept any material that has not been edited using that feature.

ROM Generator

ROM Generation was a front loaded task since the ROM schematics were necessary for the schematics team to simulate the chip. It would have been useful to start this component earlier to ensure that the design would be complete for the schematics team; in general, project milestones for one team which depended on the milestones for another team in the same week were rarely achievable.

Another lesson from the ROM/schematics handoff is that it is extremely valuable to meet in person when passing on results to address any problems quickly. Several of the ROM bugs found by the schematics team took much longer to solve because of the delay of email.

Overall Implementation

For a successful project, planning is critical. Preliminary back-of-the-envelope calculations, floorplan, and sliceplan should be completed as soon as possible, to help plan troublesome areas of the chip. Without these completed early, it is difficult to estimate layout time and the feasibility of the design. For the 6502 project, this was done later in the project and, as a result, difficult modules to design and layout were overlooked (such as the ALU).

During every step of project, freezing functional RTL, schematic, and layout at intervals is helpful for testing. When RTL is constantly changing, debugging schematics and layouts can be challenging because versions often are out of sync. Thus, many hours can be wasted debugging vectors from a previous version.

Since the 6502 used latches instead of flip-flops, timing was different from previous E158 chips, such as the MIPS processor. This made it difficult to test vcd2sp until schematics were stabilized, which was very late in the project. This also complicated validation, since the vcd2sp tool and the schematics needed to be debugged in parallel. This also increased critical path to RTL -> schematics -> layout -> verification tools.

In the future, it is wise to thoroughly test new verification tools, and create sample test modules, when there is any free time to reduce critical path of the project.

Lastly, weekly work sessions can dramatically increase productivity, because team members can easily communicate. Ideally, teams can schedule meetings themselves, but it can be necessary for the chief circuit designer to schedule meetings. Written library and version management policies, especially between teams, are helpful so that teams do not work with out-of-date cells. This was not emphasized enough in this project and, as a result, man-power is wasted. It may be necessary for the chief designer to manage a single library and authorize changes.

References

1. Apple II Reference Manual
2. Synertek SY6502 Datasheet

Appendix: RTL Code

RTL Code: top.sv

```
// top.sv
// top level test module for hmc-6502
// 2dec07
// tbarr at cs hmc edu
// The highest level of the project.  Contains the entire chip and connects
// it to memory for simulation.

`timescale 1 ns / 1 ps

module top(input logic ph1, ph2, resetb);

  // holds memory system and CPU. nothing exported, use heirarchical names
  // to examine operation

  logic [15:0] address;
  wire [7:0] data;
  logic ph0;
  logic osc_en, osc_out;
  wire razor_error;

  logic ph1_gen, ph2_gen;

  assign ph0 = !ph1;

  chip chip(address, data, ph0, resetb, read_en, razor_error, osc_en,
            osc_out, ph1_gen, ph2_gen);
  mem mem(ph1_gen, ph2_gen, !(resetb), address, data, read_en);

endmodule
```

RTL Code: chip.sv

```
// chip.sv
// chip module for hmc-6502 CPU
// 5mar08
// tbarr at cs hmc edu
// chip is the top of the heirarchy, interfacing the core with the outside
// world.

`timescale 1 ns / 1 ps
`default_nettype none

module chip(output logic [15:0] address,
            inout wire [7:0] data,
            input logic ph0, resetb,
            output logic read_en, razor_error,
            input logic osc_en,
            output logic osc_out, ph1, ph2);

            logic [7:0] data_in, data_out;
            // data is a two-way signal, so we have to be sure it drives
            // data_in if it's input and gets driven by data_out if output.
            assign data = (read_en) ? 8'bz : data_out;
            assign data_in = (read_en) ? data : 8'bz;

            // Instantiate the core module described in core.sv.
            core core(.address(address), .data_in(data_in),
                      .data_out(data_out), .ph0(ph0),
                      .resetb(resetb), .read_en(read_en),
                      .razor_error(razor_error),
                      .osc_en(osc_en), .osc_out(osc_out),
                      .ph1(ph1), .ph2(ph2));
endmodule
```

RTL Code: mem.sv

```
// mem.sv
// basic memory system for development
// 0xf000-0xffff - ROM, from file. program counter redirection = 0xf000
// 0x0000-0x1000 - RAM, inits to X
// 31 October 2007, Thomas W. Barr
// tbarr at cs dot hmc dot edu

`timescale 1 ns / 1 ps
`default_nettype none

module mem(input logic ph1, ph2, reset,
                input logic [15:0] address,
           inout wire [7:0] data,
           input logic read_write_sel );

  // 0x1000 = 4096
  logic [7:0] RAM[4095:0];
  logic [7:0] ROM[4095:0];
  reg [7:0] data_out;

  assign #3 data = (read_write_sel) ? data_out : 8'bz;

  always_ff @ ( posedge ph2 ) begin
    if ( read_write_sel ) begin
      if ( address[15:12] == 4'b0000 ) data_out = RAM[address[11:0]];
      else if ( address[15:12] == 4'b1111 ) data_out = ROM[address[11:0]];
           else data_out = 8'b0; // zero on undefined read
      end
    //memwrite
    else if ( address[15:12] == 4'b0000 ) RAM[address[11:0]] <= data;
  end

endmodule
```

RTL Code: core.sv

```
// core.sv
// core module for hmc-6502 CPU
// 31oct07
// tbarr at cs hmc edu

`timescale 1 ns / 1 ps

// core wires together the datapath and controller and throws in the clock
// generator.
module core(output logic [15:0] address,
            input [7:0] data_in,
            output [7:0] data_out,
            input logic ph0, resetb,
            output logic read_en, razor_error,
            input logic osc_en,
            output logic osc_out, ph1, ph2);

  // giant wad of controls
  logic th_in_en;
  logic th_out_en;
  logic tl_in_en;
  logic tl_out_en;
  logic [7:0] p_in_en, op_flags, p;
  logic p_out_en;
  logic p_sel;
  logic reg_write_en;
  logic [1:0] reg_read_addr_a;
  logic [1:0] reg_read_addr_b;
  logic [1:0] reg_write_addr;
  logic reg_a_en;
  logic reg_b_en;
  logic pch_in_en;
  logic pch_out_en;
  logic pcl_in_en;
  logic pcl_out_en;
  logic pc_inc_en;
  logic pc_sel;
  logic d_in_en;
  logic d_out_sel;
  logic [2:0] ah_sel;
  logic [1:0] al_sel;
  logic [3:0] alu_op;
  logic c_temp_en;
  logic [1:0] carry_sel;
  logic [7:0] constant;
  logic constant_en;
  logic flag_en;

  logic [9:0] alu_tristate_controls, alu_tristate_controls_b;

  test_structure test_structure(osc_en, osc_out);

  clockgen clockgen(ph0, ph1, ph2);

  datapath dp(data_in, data_out, address, p, ph1, ph2, resetb, razor_error,
            th_in_en, th_out_en, tl_in_en, tl_out_en, p_in_en, p_out_en, p_sel,
            reg_write_en, reg_read_addr_a, reg_read_addr_b, reg_write_addr, reg_a_en,
            reg_b_en, pch_in_en, pch_out_en, pcl_in_en, pcl_out_en, pc_inc_en, pc_sel,
            d_in_en, d_out_sel, ah_sel, al_sel, alu_op, c_temp_en, carry_sel, constant,
            constant_en, alu_tristate_controls, alu_tristate_controls_b);

  control con(data_in, p, ph1, ph2, resetb, p_in_en, alu_tristate_controls,
alu_tristate_controls_b, {
                  th_in_en,
                  th_out_en,
                  tl_in_en,
                  tl_out_en,
                  p_sel,
                  p_out_en,
                  pch_in_en,
                  pch_out_en,
                  pcl_in_en,
                  pcl_out_en,
                  pc_inc_en,
                  pc_sel,
                  d_out_sel,
                  ah_sel,
                  al_sel,
                  c_temp_en,
                  carry_sel,
                  flag_en,
                  read_en,
                  constant_en,
                  constant,
                  alu_op,
                  d_in_en,
                  reg_write_en,
                  reg_read_addr_a,
                  reg_read_addr_b,
                  reg_write_addr,
                  reg_a_en,
                  reg_b_en});
endmodule
```

RTL Code: clock.sv

```
// clock.sv
// clock system for hmc-6502 CPU
// 20mar08


`timescale 1 ns / 1 ps

module clockgen(input logic ph0,
                output logic ph1, ph2);

  assign ph2 =  ph0;
  assign ph1 = !ph0;

endmodule
```

RTL Code: test_structure.sv

```
// test_structure.sv

`timescale 1 ns / 1 ps

module test_structure( input logic osc_en,
                       output logic osc_out );
  logic a;

  assign a = !(osc_out & osc_en);
  assign #7 osc_out = a;

endmodule
```

RTL Code: control.sv

```
// control.sv
// control FSM and opcode ROM for hmc-6502 CPU
// 31oct07
// tbarr at cs hmc edu

`timescale 1 ns / 1 ps
`default_nettype none

// always kept in this order!
parameter C_STATE_WIDTH = 32;
parameter C_OP_WIDTH = 14;
parameter C_INT_WIDTH = 12; // total width of internal state signals

parameter BRANCH_TAKEN_STATE = 8'd12;
parameter BRANCH_NOT_TAKEN_STATE = 8'd11;

parameter C_TOTAL = (C_STATE_WIDTH + C_OP_WIDTH + C_INT_WIDTH);

// controller module that, together with the datapath module, makes up the
// core.
module control(input logic [7:0] data_in, p,
               input logic ph1, ph2, resetb,
               output logic [7:0] p_in_en,
               output logic [9:0] alu_tristate_controls, alu_tristate_controls_b,
               output logic [(C_STATE_WIDTH + C_OP_WIDTH - 1):0] controls_out);

  // all controls become valid on ph1, and hold through end of ph2.
  logic [7:0] latched_opcode;
  logic first_cycle, last_cycle, c_op_sel, last_cycle_s2;

  logic [(C_OP_WIDTH - 1):0] c_op_state, c_op_opcode;
  logic [(C_OP_WIDTH - 1):0] c_op_selected;
  logic [(C_STATE_WIDTH - 1):0] c_state;

  logic branch_polarity, branch_taken;

  logic [7:0] state, next_state, next_state_states, next_state_opcode;
  logic [7:0] next_state_s2;
  logic [7:0] next_state_branch;
  logic [1:0] next_state_sel;

  logic [7:0] op_flags;

  // lines for carry_sel patch
  logic [1:0] carry_sel_op, carry_sel_selected;

  // opcode logic
  latch #1 opcode_lat_p1(last_cycle, last_cycle_s2, ph1);
  latch #1 opcode_lat_p2(last_cycle_s2, first_cycle, ph2);
  latchren #8 opcode_buf(data_in, latched_opcode, ph1, first_cycle, resetb);
  opcode_pla opcode_pla(latched_opcode, {carry_sel_op,
                                       c_op_opcode, branch_polarity,
                                       op_flags, next_state_opcode});

  // branch logic
  // - p is stable 1, but won't be written on the branch op.
  // - the paranoid would add a latch to make it stable 2.
  branchlogic branchlogic(p, op_flags, branch_polarity, branch_taken);
  mux2 #8 branch_state_sel(BRANCH_NOT_TAKEN_STATE,
                           BRANCH_TAKEN_STATE, branch_taken, next_state_branch);

  // next state logic
  mux3 #8 next_state_mux(next_state_states, next_state_opcode, next_state_branch,
                         next_state_sel, next_state);

  // state PLA
  latchr #8 state_lat_p1(next_state, next_state_s2, ph1, resetb);
  latchr #8 state_lat_p2(next_state_s2, state, ph2, resetb);

  state_pla state_pla(state, {c_state, c_op_state, {last_cycle,
                                                    c_op_sel,
                                                    next_state_sel,
                                                    next_state_states}});

  and8 flag_masker(op_flags, controls_out[24], p_in_en);

  // opcode specific controls; when the microcode in the state ROM specifies
  // that signals should be specific to a particular instruction, they are
  // taken from the opcode ROM and chosen here.
  mux2 #(C_OP_WIDTH) func_mux(c_op_state, c_op_opcode, c_op_sel, c_op_selected);

  // create decoded signal for ALU
  alu_mux_controls alu_mux_generator(controls_out[13:10], alu_tristate_controls,
alu_tristate_controls_b);


  latch #1 controls_op_02(c_op_selected[02], controls_out[02], ph1);
  latch #1 controls_op_03(c_op_selected[03], controls_out[03], ph1);

  latch #1 controls_op_08(c_op_selected[08], controls_out[08], ph1);

  assign controls_out[00] = c_op_selected[00];
  assign controls_out[01] = c_op_selected[01];

  assign controls_out[04] = c_op_selected[04];
  assign controls_out[05] = c_op_selected[05];
  assign controls_out[06] = c_op_selected[06];
  assign controls_out[07] = c_op_selected[07];

  assign controls_out[09] = c_op_selected[09];
  assign controls_out[10] = c_op_selected[10];
  assign controls_out[11] = c_op_selected[11];
  assign controls_out[12] = c_op_selected[12];
  assign controls_out[13] = c_op_selected[13];

  latch #1 controls_state_09(c_state[09], controls_out[23], ph1);
  latch #1 controls_state_10(c_state[10], controls_out[24], ph1);
  latch #1 controls_state_13(c_state[13], controls_out[27], ph1);
  latch #1 controls_state_14(c_state[14], controls_out[28], ph1);
  latch #1 controls_state_15(c_state[15], controls_out[29], ph1);
  latch #1 controls_state_16(c_state[16], controls_out[30], ph1);
  latch #1 controls_state_17(c_state[17], controls_out[31], ph1);
  latch #1 controls_state_18(c_state[18], controls_out[32], ph1);
  latch #1 controls_state_19(c_state[19], controls_out[33], ph1);
  latch #1 controls_state_20(c_state[20], controls_out[34], ph1);
  latch #1 controls_state_21(c_state[21], controls_out[35], ph1);

  latch #1 controls_state_23(c_state[23], controls_out[37], ph1);

  latch #1 controls_state_25(c_state[25], controls_out[39], ph1);
  latch #1 controls_state_26(c_state[26], controls_out[40], ph1);
  latch #1 controls_state_27(c_state[27], controls_out[41], ph1);

  latch #1 controls_state_29(c_state[29], controls_out[43], ph1);
  latch #1 controls_state_31(c_state[31], controls_out[45], ph1);

  assign controls_out[14] = c_state[00];
  assign controls_out[15] = c_state[01];
  assign controls_out[16] = c_state[02];
  assign controls_out[17] = c_state[03];
  assign controls_out[18] = c_state[04];
  assign controls_out[19] = c_state[05];
  assign controls_out[20] = c_state[06];
  assign controls_out[21] = c_state[07];
  assign controls_out[22] = c_state[08];


  // carry_sel chooses where the carry in to the ALU comes from and can come
  // either from the state ROM or the opcode ROM.  This was a late addition
  // to the RTL which is why it is not integrated with func_mux above.
  mux2 #2 carry_sel_mux(c_state[12:11], carry_sel_op, c_op_sel,
                      carry_sel_selected);
  assign controls_out[26:25] = carry_sel_selected;

  assign controls_out[36] = c_state[22];

  assign controls_out[38] = c_state[24];

  assign controls_out[42] = c_state[28];
  assign controls_out[44] = c_state[30];


endmodule

// the state ROM (no longer a PLA) is where most of the microcode lives.  It
// is generated by running /src/ucode/ucasm.py on /src/ucode/6502.ucode.
module state_pla(input logic [7:0] state,
                 output logic [(C_TOTAL - 1):0] out_controls);
  always_comb
  case(state)
      `include "src/ucode/6502.ucode.compiled"
      default: out_controls <= 'x;
    endcase
endmodule

// the opcode ROM (also no longer a PLA) is where the pieces of the
// microcode specific to individual instructions live.  It is generated by
// running /src/ucode/opcode_translator/opcode_label2bin.py on the output
// from running /src/ucode/opcode_translator/instrtable2opcodes.py on
// /src/ucode/instrtable.txt.
module opcode_pla(input logic [7:0] opcode,
                 output logic [(C_OP_WIDTH + 17 - 1 + 2):0] out_data);
  always_comb
  case(opcode)
      `include "src/ucode/opcode_translator/translated_opcodes.txt"

    default: out_data <= '0; // processor resets on undefined opcode
  endcase
```

```
endmodule

// the decoder for the ALU.  Takes in ALUOP and hands off a one-hot encoding
// of the operation to the ALU module.
module alu_mux_controls(input logic [3:0] op,
                 output logic [9:0] alu_tristate_controls, alu_tristate_controls_b);

 assign alu_tristate_controls_b = ~alu_tristate_controls;

 always_comb begin;
    case (op)
      4'h0: alu_tristate_controls = 10'b1000000000; // Increment
      4'h1: alu_tristate_controls = 10'b1000000000; // Decrement
      4'h2: alu_tristate_controls = 10'b1000000000; // Add
      4'h3: alu_tristate_controls = 10'b1000000000; // Subtract
      4'h4: alu_tristate_controls = 10'b0100000000; // OR
      4'h5: alu_tristate_controls = 10'b0010000000; // Arithmetic Left Shift
      4'h6: alu_tristate_controls = 10'b0001000000; // Rotate Left
      4'h7: alu_tristate_controls = 10'b0000100000; // Rotate Right
      4'h8: alu_tristate_controls = 10'b0000010000; // AND
      4'h9: alu_tristate_controls = 10'b0000001000; // test bits (used for "bit"
instruction)
      4'ha: alu_tristate_controls = 10'b0000000100; // XOR
      4'hb: alu_tristate_controls = 10'b0000000010; // Ones
      default: alu_tristate_controls = 10'b0000000001;
    endcase
  end
endmodule
```

RTL Code: datapath.sv

```
// datapath.sv
// datapath for hmc-6502 CPU
// 31oct07
// tbarr at cs hmc edu

`timescale 1 ns / 1 ps
`default_nettype none

module datapath(input logic [7:0] data_in,
                output logic [7:0] data_out,
                output logic [15:0] address,
                output logic [7:0] p_s1,
                input logic ph1, ph2, resetb,
                output logic razor_error,

                // controls list from ucodeasm:
                input logic th_in_en,
                input logic th_out_en,
                input logic tl_in_en,
                input logic tl_out_en,
                input logic [7:0] p_in_en,
                input logic p_out_en,
                input logic p_sel,
                input logic reg_write_en,
                input logic [1:0] reg_read_addr_a,
                input logic [1:0] reg_read_addr_b,
                input logic [1:0] reg_write_addr,
                input logic reg_a_en,
                input logic reg_b_en,
                input logic pch_in_en,
                input logic pch_out_en,
                input logic pcl_in_en,
                input logic pcl_out_en,
                input logic pc_inc_en,
                input logic pc_sel,
                input logic d_in_en,
                input logic d_out_sel,
                input logic [2:0] ah_sel,
                input logic [1:0] al_sel,
                input logic [3:0] alu_op,
                input logic c_temp_en,
                input logic [1:0] carry_sel,
                input logic [7:0] constant,
                input logic constant_en,

                input logic [9:0] alu_tristate_controls, alu_tristate_controls_b
                );

  wire  [7:0] a_s1, flag_selected_s2;
  logic [7:0] th_s1, tl_s1, r_s2;

  logic [7:0] reg_a_s1, reg_b_s1;

  logic [7:0] pch_in_s2, pcl_in_s2, pch_next_s1, pch_next_s2, pcl_next_s1,
              pcl_next_s2, pch_s1, pcl_s1;
  logic pch_carry, pcl_carry;

  logic [7:0] di_s1;

  wire  [7:0] b_s1, b_s2;

  logic [7:0] th_s2, tl_s2;

  logic [7:0] r_s1;
  logic [7:0] flags_s1, flags_s2;
  logic c_in_s1, c_temp_s1;

  logic dp_error, flags_error;

  // registers
  registerbuf temp_high(th_in_en, th_out_en, th_s1, r_s2, a_s1, ph2);
  registerbuf  temp_low(tl_in_en, tl_out_en, tl_s1, r_s2, a_s1, ph2);
  registerbufmasked flaglatch(p_in_en, p_out_en, p_s1, flag_selected_s2,
                              b_s1, ph2);
  mux2 #8 p_sel_mux(flags_s2, r_s2, p_sel, flag_selected_s2);

  // constant
  tristate #8 constant_tris(constant, constant_en, a_s1);

  // register file
  regfile regfile(ph2, reg_write_en, reg_read_addr_a, reg_read_addr_b,
          reg_write_addr, r_s2, reg_a_s1, reg_b_s1);
  tristate #8 rfile_tris_a(reg_a_s1, reg_a_en, a_s1);
  tristate #8 rfile_tris_b(reg_b_s1, reg_b_en, b_s1);

  // program counter
  registerbuf pc_high(pch_in_en, pch_out_en, pch_s1, pch_in_s2, b_s1, ph2);
  halfadder #8 pch_inc(pch_s1, pcl_carry, pch_next_s1, pch_carry);
  mux2 #8 pch_sel(pch_next_s2, r_s2, pc_sel, pch_in_s2);
  latch #8 pch_next_buf(pch_next_s1, pch_next_s2, ph1);

  registerbuf  pc_low(pcl_in_en, pcl_out_en, pcl_s1, pcl_in_s2, b_s1, ph2);
  inc #8 pcl_inc(pcl_s1, pc_inc_en, pcl_next_s1, pcl_carry);
  mux2 #8 pcl_sel(pcl_next_s2, r_s2, pc_sel, pcl_in_s2);
  latch #8 pcl_next_buf(pcl_next_s1, pcl_next_s2, ph1);

  // memory I/O
  // -input
  latch #8 d_in_buf(data_in, di_s1, ph2);
  tristate #8 di_tris(di_s1, d_in_en, a_s1);

  // -output
  // we must buffer the b_s1 line into a b_s2 line to put on the bus
  latch #8 b_buf(b_s1, b_s2, ph1);
  mux2 #8 data_out_sel_mux(r_s2, b_s2, d_out_sel, data_out);

  // -address bus
  latch th_buf(th_s1, th_s2, ph1);
  latch tl_buf(tl_s1, tl_s2, ph1);

  mux5 #8 ah_mux(pch_next_s2, r_s2, th_s2, 8'h00, 8'h01, ah_sel, address[15:8]);
  mux3 #8 al_mux(pcl_next_s2, r_s2, tl_s2, al_sel, address[7:0]);
```

```
  // ALU and carry logic
  alu alu(a_s1, b_s1, r_s1, alu_op, alu_tristate_controls, alu_tristate_controls_b,
c_in_s1, p_s1[4],
          flags_s1[1], flags_s1[7], flags_s1[6], flags_s1[0]);

  // -buffer to prevent loops
  razorlatch #8 r_buf(r_s1, r_s2, ph1, dp_error);
  razorlatch #8 flag_buf(flags_s1, flags_s2, ph1, flags_error);

  assign razor_error = dp_error | flags_error;

  // -select carry source
  latchen #1 c_temp(flags_s2[0], c_temp_s1, ph2, c_temp_en);
  mux4 #1 carry_sel_mux(p_s1[0], c_temp_s1, 1'b0, 1'b1, carry_sel, c_in_s1);

endmodule
```

RTL Code: alu.sv

```
// alu.sv
// alu for hmc-6502 CPU
// 31oct07
// tbarr at cs hmc edu

`timescale 1 ns / 1 ps

module alu(input logic [7:0] a, b,
                output logic [7:0] y,
           input logic [3:0] op,
           input logic [9:0] alu_tristate_controls, alu_tristate_controls_b,
           input logic c_in, bcd,
           output logic zero, negative, overflow, c_out);

  logic testbits;
  assign testbits = (op === 4'h9);

  // Set the flags
  assign zero = (y === 8'b0); // Z flag
  assign negative = y[7] | (testbits & a[7]); // S flag

  // If we're doing a subtract, invert a to put through the adders.
  logic [7:0] a_conditionally_inverted;
  assign a_conditionally_inverted = (op === 4'h3) ? ~a : a;

  logic [7:0] full_sum;
  logic [6:0] low7_sum;
  logic low7_cout, high_sum, full_cout;

  adderc #7 lower7_add(a_conditionally_inverted[6:0], b[6:0], c_in, low7_sum, low7_cout);
  adderc #1 high_add(a_conditionally_inverted[7], b[7], low7_cout, high_sum, full_cout);
  assign full_sum = {high_sum, low7_sum};

  // The whole purpose of the last 8 lines was to get the carry out from bits
  // 6 and 7 to produce the overflow flag:
  assign overflow = (low7_cout ^ full_cout) | (testbits & a[6]);

  always_comb begin
    case (alu_tristate_controls)
      10'b1000000000: begin
                      case (op)
                        4'h0: {c_out, y} = a + c_in; // inc
                        4'h1: {c_out, y} = a - c_in; // dec
                        4'h2: {c_out, y} = {full_cout, full_sum}; // add
                        4'h3: {c_out, y} = {full_cout, full_sum}; // sub
                      endcase
                      end
      10'b0100000000: {y, c_out} = {c_in, a}; // ror
      10'b0010000000: {c_out, y} = {a, 1'b0}; // asl
      10'b0001000000: {c_out, y} = {a, c_in}; // rol
      10'b0000100000: {c_out, y} = {1'b0, a | b}; // OR
      10'b0000010000: {c_out, y} = {1'b0, a & b}; // AND
      10'b0000001000: {y, c_out} = {1'b0, a & b}; // test bits
      10'b0000000100: {c_out, y} = {1'b0, a ^ b}; // EOR
      10'b0000000010: {y, c_out} = 9'b111111111; // ones (for setting flags)
      default: {y, c_out} = 9'b0;
  endcase

  end
endmodule
```

RTL Code: parts.sv

```
// parts.sv
// parts bin for hmc-6502 CPU
// 31oct07
// tbarr at cs hmc edu
// All the bits and pieces used in larger modules.

`timescale 1 ns / 1 ps

// Adder with carry in and out
module adderc #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0]  a, b,
              input logic              cin,
              output logic [WIDTH-1:0] y,
              output logic             cout);

  assign {cout, y} = a + b + cin;
endmodule

// Adder with no carry in (actually described in code as A+C_in rather than
// A+B but it's functionally equivalent)
module halfadder #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0]  a,
              input logic              cin,
              output logic [WIDTH-1:0] y,
              output logic             cout);

  assign {cout, y} = a + cin;
endmodule

// Decides whether or not to take a branch by comparing unmasked flags to the
// processor state register and branhc polarity.
module branchlogic(input logic [7:0] p, op_flags,
                   input logic branch_polarity,
                   output logic branch_taken);

  logic flag_high;
  assign flag_high = | (op_flags & p);
  assign branch_taken = branch_polarity ^ flag_high;
endmodule

// The almighty tristate.
module tristate #(parameter WIDTH = 8)
                (input logic [WIDTH-1:0] in,
                 input logic enable,
                 output [WIDTH-1:0] bus);

  wire [WIDTH-1:0] highz;
  assign highz = {WIDTH{1'bz}};

  assign bus = (enable) ? in : highz;
endmodule

module latch #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q,
              input logic clk);

  always_latch
      if (clk) q <= d;
endmodule
```

```
// Razor Latch module.  In verilog this is functionally equivallent to a
// normal latch, but the error output is important in actual hardware.  See
// the description of razor latches in the chip report for more information.
module razorlatch #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q,
              input logic clk,
              output logic error);

  always_latch
      if (clk) q <= d;
endmodule

module latchr #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q,
              input logic clk, resetb);

  always_latch
    if (!resetb) q <= 0;
      else if (clk) q <= d;
endmodule

module latchen #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q,
              input logic clk, en);

  always_latch
    if (clk & en) q <= d;
endmodule

module latchren #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q,
              input logic clk, en, resetb);

  always_latch
    if (!resetb) q <= 0;
      else if (clk & en) q <= d;
endmodule

module and8 (input logic [7:0] a,
             input logic s,
             output logic [7:0] y);
  assign y[0] = a[0] & s;
  assign y[1] = a[1] & s;
  assign y[2] = a[2] & s;
  assign y[3] = a[3] & s;
  assign y[4] = a[4] & s;
  assign y[5] = a[5] & s;
  assign y[6] = a[6] & s;
  assign y[7] = a[7] & s;
endmodule

module flopr #(parameter WIDTH = 8)
             (input logic [WIDTH-1:0] d,
              output logic [WIDTH-1:0] q,
              input logic clk, resetb);

  always_ff @ (posedge clk)
    if (!resetb) q <= 0;
    else q <= d;
endmodule

module registerbuf (input logic in_enable, out_enable,
                 output logic [7:0] value,
                 input logic [7:0] in_bus,
                 inout [7:0] out_bus,
                 input logic clk);
  logic gated_clk;
  assign gated_clk = in_enable & clk;

  latch #8 latch(in_bus, value, gated_clk);
  tristate #8 tris(value, out_enable, out_bus);
endmodule

// Incrementer
module inc #(parameter WIDTH = 16)
                  (input logic [WIDTH-1:0] a,
                   input logic c_in,
                   output logic [WIDTH-1:0] y,
                   output logic c_out);
  assign {c_out, y} = a + c_in;
endmodule

module registerbufmasked (input logic [7:0] in_enable,
                  input logic out_enable,
                  output logic [7:0] value,
                  input logic [7:0] in_bus,
                  inout [7:0] out_bus,
                  input logic clk);

  // fanned out registerbuf
  latch #1 latch0(in_bus[0], value[0], clk & in_enable[0]);
  latch #1 latch1(in_bus[1], value[1], clk & in_enable[1]);
  latch #1 latch2(in_bus[2], value[2], clk & in_enable[2]);
  latch #1 latch3(in_bus[3], value[3], clk & in_enable[3]);
  latch #1 latch4(in_bus[4], value[4], clk & in_enable[4]);
  latch #1 latch5(in_bus[5], value[5], clk & in_enable[5]);
  latch #1 latch6(in_bus[6], value[6], clk & in_enable[6]);
  latch #1 latch7(in_bus[7], value[7], clk & in_enable[7]);

  tristate #8 tris(value, out_enable, out_bus);
endmodule

// muxes - from MIPS project
module mux2 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1,
              input              s,
              output [WIDTH-1:0] y);

  assign y = s ? d1 : d0;
endmodule

module mux3 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1, d2,
              input  [1:0]       s,
              output [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);
endmodule

module mux4 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1, d2, d3,
              input  [1:0]       s,
              output [WIDTH-1:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2)
                    : (s[0] ? d1 : d0);
endmodule

module mux5 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1, d2, d3, d4,
              input   [2:0]  s,
              output  [WIDTH-1:0] y);

  // 101 = d4; 100 = d3; 010 = d2; 001 = d1; 000 = d0

  assign y = s[2] ? (s[0] ? d4 : d3)
                    : (s[1] ? d2 : (s[0] ? d1 : d0));
```

```
endmodule

// modified from MIPS project
module regfile(input         clk,
               input         write_enable,
               input  [1:0]  read_addr_a, read_addr_b, write_addr,
               input  [7:0]  write_data,
               output [7:0]  read_data_a, read_data_b);

  reg [7:0] reg_file [3:0];

  logic gated_clk;
  assign gated_clk = clk & write_enable;

  // three ported register file
  // read two ports combinationally
  // write third port as latch

  always_latch
    if (gated_clk) reg_file[write_addr] <= write_data;

  assign read_data_a = reg_file[read_addr_a];
  assign read_data_b = reg_file[read_addr_b];
endmodule
```

RTL Code: applemem.sv

```
// applemem.sv
// basic memory system for development
// 0xb000-0xffff - ROM, from file.
// 0x0000-0x1000 - RAM, inits to X
// 31 October 2007, Thomas W. Barr
// tbarr at cs dot hmc dot edu

`timescale 1 ns / 1 ps
`default_nettype none

module mem(input logic ph1, ph2, reset,
                input logic [15:0] address,
           inout wire [7:0] data,
           input logic read_write_sel );

  // 0x1000 = 4096
  logic [7:0] RAM[4095:0];
  logic [7:0] ROM[16383:0];
  reg [7:0] data_out;

  assign #3 data = (read_write_sel) ? data_out : 8'bz;

  always_ff @ ( posedge ph2 ) begin
    if ( read_write_sel ) begin
      if ( address[15:12] == 4'b0000 ) data_out = RAM[address[11:0]];
      else if ( address > 16'hbfff )
          begin
            data_out = ROM[(address - 16'hc000)];
          end
           else data_out = 8'b00; // zero on undefined read
      end
    //memwrite
    else
      begin
        if ( address[15:12] == 4'b0000 ) RAM[address[11:0]] <= data;
        else $display("wrote (0x%h) out of bounds to %h", data, address);
      end
  end

endmodule
```

RTL Code: control-prototype.sv

```
// control-prototype.sv
// control FSM and opcode ROM for hmc-6502 CPU
// 31oct07
// tbarr at cs hmc edu
// this was a proof of concept for the controller early in the design
// process.  It is no longer used in the RTL.  Please refer to control.sv.

`timescale 1 ns / 1 ps

module control(input logic [7:0] data_in,
               input logic ph1, ph2, reset,

               output logic [3:0] controls
               );

  // opcode decoding logic
  logic [7:0] latched_opcode;
  logic op_en, op_en_buf, opcode_gated_clk;
  logic [7:0] op_controls;

  flopr #1 op_en_reg(op_en, op_en_buf, ph2, reset);
  assign opcode_gated_clk = ph1 & op_en_buf;
  latch #8 opcode_buf(data_in, latched_opcode, opcode_gated_clk, reset);
  opcode_pla opcode_pla(latched_opcode, op_controls);

  // FSM logic
  logic [3:0] state, next_state;
  logic [9:0] c_s1, c_s2;
  logic next_state_sel;

  assign op_en = c_s2[4];
  assign next_state_sel = c_s2[5];

  mux2 #4 next_state_mux(c_s2[3:0], op_controls[3:0], next_state_sel,
                         next_state);
  flopr #4 state_flop(next_state, state, ph1, reset);
  state_pla state_pla(state, c_s2);

  latch #10 control_buf(c_s2, c_s1, ph2, reset);

  // output controls on correct phase
  assign controls[1:0] = c_s2[7:6];
  assign controls[3:2] = c_s1[9:8];

endmodule

module state_pla(input logic [3:0] state,
                 output logic [9:0] out_controls);
  always_comb
  case(state)
    4'h0 : out_controls <= 10'b0000_1_0_0000;
    4'h1 : out_controls <= 10'b0001_0_0_0010;
    4'h2 : out_controls <= 10'b0010_0_0_0011;
    4'h3 : out_controls <= 10'b0111_0_1_0000;
    4'h4 : out_controls <= 10'b0001_0_0_0101;
    4'h5 : out_controls <= 10'b0010_0_0_0110;
    4'h6 : out_controls <= 10'b1011_0_1_0000;
    default: out_controls <= 8'b0;
  endcase
endmodule

module opcode_pla(input logic [7:0] opcode,
                 output logic [7:0] out_data);
  always_comb
  case(opcode)
    8'h01: out_data <= 8'h11;
    8'h02: out_data <= 8'h14;
    default: out_data <= 8'h00;
  endcase
endmodule
```

Appendix: Test Vectors and Verilog Test Benches

Suite P: Power Test

Suite P is provided as a short test of the major components of the processor datapath for estimating power consumption in HPICE. It executes 11 instructions (if successful) and reports whether it succeeded or failed based on whether or not the memory address $42 stores the value 0xCF when the test bench ends.

The following is the bytes of the program (test/roms/PowerTest.rom) which get loaded into the chip's ROM as instructions:

```
// Power Regression Test
// Heather Justice 3/12/08
// EXPECTED RESULT: $42 = 0xCF
//
// LDA #$E7 (A=0xE7)
a9
e7
// STA $20 ($20=0xE7)
85
20
// LDA #$18 (A=0x18)
a9
18
// STA $10 ($10=0x18)
85
10
// EOR #$FF (A=0xE7)
49
ff
// CMP $20
c5
20
// BNE final (not taken)
d0
08
// SBC $10 (A=0xCF)
e5
10
// CMP $20
c5
20
// BNE final (taken)
d0
02
// EOR #$AA (shouldn't happen, would result in A=0x65)
49
aa
// final:
// STA $42 ($42=0xCF)
85
42
00
00
```

This program can be run in ModelSim by executing the appropriate script (test/scripts/PowerTest.fdo). The following is the Verilog test bench (test/unittests/PowerTest.sv) for this test:

```
// PowerTest.sv
// Heather Justice 3/25/08

`timescale 1 ns / 1 ps

module optest;
  reg ph1, ph2, resetb;

  wire [7:0] data;

  top top(ph1, ph2, resetb);

  always begin
    ph1 <= 1; #10; ph1 <= 0; #10;
  end
  always begin
```

```
    ph2 <= 0; #10; ph2 <= 1; #10;// ph2 <= 0; #2;
  end

  initial begin
    // for VCD file
    $dumpfile("test/VCD/outSuiteP.vcd");
    $dumpvars(1, top.chip.address, top.chip.data_in, top.chip.address,
              top.chip.data_out, top.chip.ph0, top.chip.resetb, top.chip.read_en,
              top.chip.razor_error);

    // init ROM
    top.mem.ROM[4093] = 8'hf0;
    top.mem.ROM[4092] = 8'h00;

    // path relative to this file.
    $readmemh("test/roms/PowerTest.rom", top.mem.ROM);

    // start test
    resetb = 0;
    #100;
    resetb = 1;
    #1000;
    assert (top.mem.RAM[66] == 8'hCF) $display ("PASSED Power Test");
      else $error("FAILED Power Test");
    $dumpflush;
    $stop;
  end
endmodule
```

Suite A+: Test of All Instructions

Suite A consists of 15 tests which together test every instruction for the 6502 (151 total opcodes). The first program in this suite only tests loads and stores. Once the first test passes, later tests can assume that loads and stores, along with any other instructions tested in earlier tests, are known to work. This means that the tests should be run in numerical order in order to simplify the debugging process. The programs (stored individually in test/roms/SuiteA/) in Suite A test subsets of the instructions as follows:

Test #	Instructions Tested
00	loads and stores (for all registers and all addressing modes)
01	AND, EOR, ORA (all addressing modes)
02	INC, DEC (all addressing modes)
03	bit shift instructions (all addressing modes)
04	jumps and RTS
05	transfer register instructions and increment/decrement for registers X & Y
06	ADC, SBC (all addressing modes)
07	CMP (all addressing modes), BEQ, BNE
08	CPX, CPY, BIT (all addressing modes)
09	other branches (BPL, BMI, BVC, BVS, BCC, BCS)
10	CLC, SEC, CLV, NOP
11	stack instructions (PHA, PLA, PHP, PLP)
12	RTI
13	other flag instructions (SEI, CLI, SED, CLD)
14	BRK

The assembly code which these programs implement is also available as .asm files in the directory emu/testvectors/TestAllInstructions/. The scripts for running the tests individually are in the directory test/scripts/Suite/ and the Verilog test benches are in the directory test/SuiteATests/.

Suite A+ incorporates all fifteen Suite A tests into a single test program with a single test bench. It can be run in ModelSim with the appropriate script (test/scripts/AllSuiteA.fdo), and it will report whether it succeeds or fails. If it fails, it will report which test number it believes it failed on, but if branches or jumps do not function correctly then it will likely report the wrong number for the failed test. This program essentially contains copies of each of the fifteen Suite A tests along with small portions of code for checking whether or not each test passed. The program should end in an infinite loop on the final jump instruction, with either 0xFF or the number of the failed test stored at memory address $0210. The following is the Verilog test bench (test/SuiteATests/AllSuiteA.sv) for Suite A+:

```
// AllSuiteA.sv
// Heather Justice 4/17/08

`timescale 1 ns / 1 ps

module optest;
  reg ph1, ph2, resetb;

  reg [7:0] aresult;

  wire [7:0] data;

  top top(ph1, ph2, resetb);

  always begin
    ph1 <= 1; #8; ph1 <= 0; #12;
  end
  always begin
    ph2 <= 0; #10; ph2 <= 1; #8; ph2 <= 0; #2;
  end

  initial begin
    // for VCD file
    $dumpfile("test/VCD/outAllSuiteA.vcd");
    $dumpvars;

    // init ROM
    top.mem.ROM[4093] = 8'hf0;
    top.mem.ROM[4092] = 8'h00;

    // BRK Interrupt Vector
    top.mem.ROM[4095] = 8'hf5;
    top.mem.ROM[4094] = 8'ha4;

    // path relative to this file.
    $readmemh("test/roms/AllSuiteA.rom", top.mem.ROM);

    // start test
    resetb = 0;
    #100;
    resetb = 1;
    #45000;
    assign aresult = top.mem.RAM[528];
    assert (aresult == 8'hFF) $display ("SUCCESS! PASSED SUITE A.");
        else $error("FAILURE: Suite A failed at test%d.", aresult);
    //assert (top.mem.RAM[528] == 8'hFF) $display ("PASSED");
    //  else $error("FAILED");
    $dumpflush;
    $stop;
  end
endmodule
```

The following is the bytes of the program (test/roms/AllSuiteA.rom) which get loaded into the chip's ROM as instructions for executing Suite A+:

```
//////////////////////////////////////////
//
// CUMULATIVE TEST OF SUITE A
// Heather Justice
// 4/16/08
//
// EXPECTED RESULTS: $0210 = 0xFF
// Result Details:
//   Check $0210 for result. Result 0xFF
// means that all tests in Suite A
// [hopefully] passed. Result 0xFE means
// that something went really really
// wrong (for example, if JMP went to
// someplace really wrong). Results 0x00
// through 0x14 indicate the first test
// that was recognized to fail.
//
//////////////////////////////////////////
// INITIALIZE EXPECTED VALUES
a9
00
8d
10
02
a9
55
8d
00
02
a9
aa
8d
01
02
a9
ff
8d
02
02
a9
6e
8d
03
02
a9
42
8d
04
02
a9
33
8d
05
02
a9
9d
8d
06
02
a9
7f
8d
07
02
a9
a5
8d
08
02
a9
1f
8d
09
02
a9
ce
8d
0a
02
a9
29
8d
0b
02
a9
42
8d
0c
02
a9
// note that this depends on other flags...
0c
8d
0d
02
a9
42
8d
0e
02
//////////////////////////////////////////
//
// TEST 00
//
//////////////////////////////////////////
// 6502 Test #00
// Heather Justice 3/11/08
// Tests instructions LDA/LDX/LDY &
// STA/STX/STY with all addressing modes.
//
// EXPECTED RESULTS:
//  $022A = 0x55 (decimal 85)
//  A = 0x55, X = 0x2A, Y = 0x73
//
// LDA imm #85 (A=85)
a9
55
// LDX imm #42 (X=42)
a2
2a
// LDY imm #115 (Y=115)
a0
73
// STA zpg $81 ($81=85)
85
81
// LDA imm #$01 (A=01)
a9
01
// STA zpg $61 ($61=01)
85
61
// A=0x7E
a9
7e
// LDA zpg $81 (A=85)
a5
81
// STA abs $0910 ($0910=85)
8d
10
09
// A=0x7E
a9
7e
// LDA abs $0910 (A=85)
ad
10
09
// STA zpx $56,X ($80=85)
95
56
// A=0x7E
a9
7e
// LDA zpx $56,X (A=85)
b5
56
// STY zpg $60 ($60=115)
84
60
// STA idy ($60),Y ($01E6=85)
```

```
91
60
// A=0x7E
a9
7e
// LDA idy ($60),Y (A=85)
B1
60
// STA abx $07ff,X ($0829=85)
9d
ff
07
// A=0x7E
a9
7e
// LDA abx $07ff,X (A=85)
bd
ff
07
// STA aby $07ff,Y ($0872=85)
99
ff
07
// A=0x7E
a9
7e
// LDA aby $07ff,Y (A=85)
b9
ff
07
// STA idx ($36,X) ($0173=85)
81
36
// A=0x73
a9
7e
// LDA idx ($36,X) (A=85)
a1
36
// STX zpg $50 ($50=42)
86
50
// LDX zpg $60 (X=115)
a6
60
// LDY zpg $50 (Y=42)
a4
50
// STX abs $0913 ($0913=115)
8e
13
09
// X=0x22
a2
22
// LDX abs $0913 (X=115)
ae
13
09
// STY abs $0914 ($0914=42)
8c
14
09
// Y=0x99
a0
99
// LDY abs $0914 (Y=42)
ac
14
09
// STY zpx $2D,X ($A0=42)
94
2D
// STX zpy $77,Y ($A1=115)
96
77
// Y=0x99
a0
99
// LDY zpx $2D,X (Y=42)
b4
2d
// X=0x22
a2
22
// LDX zpy $77,Y (X=115)
b6
77
// Y=0x99
a0
99
// LDY abx $08A0,X (Y=115)
bc
a0
08
// X=0x22
a2
22
// LDX aby $08A1,Y (X=42)
be
a1
08
// STA abx $0200,X ($022A=85)
9d
00
02
//////////////////////////////////////////
//
// CHECK TEST 00
//
ad
2a
02
cd
00
02
f0
03
4c
ca
f5
a9
fe
8d
10
02
//////////////////////////////////////////
//
// TEST 01
//
//////////////////////////////////////////
// 6502 Test #01
// Heather Justice 3/12/08
// Tests instructions AND & EOR & ORA with
// all addressing modes.
// Assumes that LDA/LDX/LDY & STA/STX/STY
// work with all addressing modes.
//
// EXPECTED RESULTS: $A9 = 0xAA
//
// A = 0x55
a9
55
// A = 0x55&0x53 = 0x51
29
53
// A = 0x51|0x38 = 0x79
09
38
// A = 0x79^0x11 = 0x68
49
11
// Stores...
85
99
a9
b9
85
10
a9
e7
85
```

```
11
a9
39
85
12
a5
99
// A = 0x68&0xB9 = 0x28
25
10
// A = 0x48|0xE7 = 0xEF
05
11
// A = 0xEF^0x39 = 0xD6
45
12
// X = 0x10
A2
10
// Stores...
85
99
a9
bc
85
20
a9
31
85
21
a9
17
85
22
a5
99
// A = 0xD6&0xBC = 0x94
35
10
// A = 0x94|0x31 = 0xB5
15
11
// A = 0xB5^0x17 = 0xA2
55
12
// Stores...
85
99
a9
6f
8d
10
01
a9
3c
8d
11
01
a9
27
8d
12
01
a5
99
// A = 0xA2&0x6F = 0x22
2d
10
01
// A = 0x22|0x3C = 0x3E
0d
11
01
// A = 0x3E^0x27 = 0x19
4d
12
01
// Stores...
85
99
a9
```

```
8a
8d
20
01
a9
47
8d
21
01
a9
8f
8d
22
01
a5
99
// A = 0x19&0x8A = 0x08
3d
10
01
// A = 0x08|0x47 = 0x4F
1d
11
01
// A = 0x47^0x8F = 0xC0
5d
12
01
// Y = 0x20
a0
20
// Stores...
85
99
a9
73
8d
30
01
a9
2a
8d
31
01
a9
f1
8d
32
01
a5
99
// A = 0xC0&0x73 = 0x40
39
10
01
// A = 0x40|0x2A = 0x6A
19
11
01
// A = 0x6A^0xF1 = 0x9B
59
12
01
// Stores...
85
99
a9
70
85
30
a9
01
85
31
a9
71
85
32
a9
01
85
```

```
33
a9
72
85
34
a9
01
85
35
a9
c5
8d
70
01
a9
7c
8d
71
01
a9
a1
8d
72
01
a5
99
// A = 0x9B&0xC5 = 0x81
21
20
// A = 0x81|0x7C = 0xFD
01
22
// A = 0xFD^0xA1 = 0x5C
41
24
// Stores...
85
99
a9
60
85
40
a9
01
85
41
a9
61
85
42
a9
01
85
43
a9
62
85
44
a9
01
85
45
a9
37
8d
50
02
a9
23
8d
51
02
a9
9d
8d
52
02
a5
99
// Y = 0xF0
a0
f0
// A = 0x5C&0x37 = 0x14
31
40
// A = 0x14|0x2B = 0x37
11
42
// A = 0x37^0x9D = 0xAA
51
44
// final store $A9 = 0xAA
85
a9
//////////////////////////////////////////////
// CHECK TEST 01
//
a5
a9
cd
01
02
f0
08
a9
01
8d
10
02
4c
ca
f5
////////////////////////////////////////////
//
// TEST 02
//
////////////////////////////////////////////
// 6502 Test #02
// Heather Justice 3/12/08
// Tests instructions INC & DEC with all
// addressing modes.
// Assumes that LDA/LDX/LDY & STA/STX/STY
// work with all addressing modes.
//
// EXPECTED RESULTS: $71=0xFF
//
// initial loads (A=0xFF & X=0x00)
a9
ff
a2
00
// will result in A=0x01 & X=0x01 &
// $90=0x01
85
90
e6
90
e6
90
a5
90
a6
90
// will result in A=0x02 & X=0x02 &
// $91=0x02
95
90
f6
90
b5
90
a6
91
// will result in A=0x03 & X=0x03 &
// $0192=0x03
9d
90
01
ee
92
01
bd
```

```
90
01
ae
92
01
// will result in A=0x04 & X=0x04 &
// $0193=0x04
9d
90
01
fe
90
01
bd
90
01
ae
93
01
// will result in A=0x03 & X=0x03 &
// $0174=0x03
9d
70
01
de
70
01
bd
70
01
ae
74
01
// will result in A=0x02 & X=0x02 &
// $0173=0x02
9d
70
01
ce
73
01
bd
70
01
ae
73
01
// will result in A=0x01 & X=0x01 &
// $72=0x01
95
70
d6
70
b5
70
a6
72
// final result $71=0xFF
95
70
c6
71
c6
71
///////////////////////////////////////////
//
// CHECK TEST 02
//
a5
71
cd
02
02
f0
08
a9
02
8d
10
02
4c
ca
f5
///////////////////////////////////////////
//
// TEST 03
//
////////////////////////////////////////////
6502 Test #03
// Heather Justice 3/12/08
// Tests instructions ASL & LSR & ROL & ROR
// with all addressing modes.
// Assumes that loads & stores & ORA work
// with all addressing modes.
//
// EXPECTED RESULTS: $01DD = 0x6E
//
// will result in A=0x4A
a9
4b
4a
0a
// will result in A=0x14 & $50=0x14
85
50
06
50
06
50
46
50
a5
50
// will result in A=0x2E & X=0x14 & $60=2E
a6
50
09
c9
85
60
16
4c
56
4c
56
4c
b5
4c
// will result in A=0x36 & X=2E & $012E=36
a6
60
09
41
8d
2e
01
5e
00
01
5e
00
01
1e
00
01
bd
00
01
// will result in A=0x5A & X=0x36 &
// $0136=0x5A
ae
2e
01
09
81
9d
00
01
4e
36
01
4e
```

```
36
01
0e
36
01
bd
00
01
// now testing rol & ror...
// will result in A=0xB4 & $70=0xB4
2a
2a
6a
85
70
// will result in A=0x5B & X=0xB4 &
// $C0=0x5B
a6
70
09
03
95
0C
26
c0
66
c0
66
c0
b5
0c
// will result in A=0xB7 & X=0x5B &
// $D0=0xB7
a6
c0
85
d0
36
75
36
75
76
75
a5
d0
// will result in A=0xDD & X=0xB7 &
// $01B7=0xDD
a6
d0
9d
00
01
2e
b7
01
2e
b7
01
2e
b7
01
6e
b7
01
bd
00
01
// will result in X=0xDD & $01DD=0x6E
ae
b7
01
8d
dd
01
3e
00
01
7e
00
01
7e
00
01
//////////////////////////////////////////
//
// CHECK TEST 03
//
ad
dd
01
cd
03
02
f0
08
a9
03
8d
10
02
4c
ca
f5
//////////////////////////////////////////
//
// TEST 04
//
////////////////////////////////////////////
6502 Test #04
// Heather Justice 3/12/08
// Tests instructions JMP (both addressing
// modes) & JSR & RTS.
// Assumes that loads & stores & ORA work
// with all addressing modes.
// NOTE: Depends on addresses of
// instructions... Specifically, the
// "final" address is actually hard-coded
// at address $0020 (first 4 lines of
// code). Additionally, a JMP and JSR
// specify specific addresses.
//
// EXPECTED RESULTS: $40=0x42
//
// start:
// LDA #$E8
a9
e8
// STA $20
85
20
// LDA #$F2
a9
f2
// STA $21
85
21
// LDA #$00
a9
00
// ORA #$03
09
03
// JMP jump1
4c
d5
f2
// ORA #$FF ; not done
09
ff
// jump1:
// ORA #$30
09
30
// JSR subr
20
e1
f2
// ORA #$42
09
42
// JMP ($0020)
6c
```

```
20
00
// ORA #$FF ; not done
09
ff
// subr:
// STA $30 ($30=0x33)
85
30
// LDX $30
a6
30
// LDA #$00
a9
00
// RTS
60
// final: (as hardcoded by $0020)
// STA $0D,X
95
0d
///////////////////////////////////////////
//
// CHECK TEST 04
//
a5
40
cd
04
02
f0
08
a9
04
8d
10
02
4c
ca
f5
///////////////////////////////////////////
//
// TEST 05
//
/////////////////////////////////////////////
6502 Test #05
// Heather Justice 3/15/08
// Tests instructions for transfering
// values between registers (TAX, TXA, TYA,
// TAY, DEX, INX, DEY, INY, TXS, TSX).
// Assumes that loads & stores work with
// all addressing modes.
//
// EXPECTED RESULTS: $40 = 0x33
//
// LDA #$35 (A=0x35)
a9
35
// A -> X
aa
// X--
ca
// X--
ca
// X++
e8
// X -> A (A=0x34)
8a
// A -> Y
a8
// Y--
88
// Y--
88
// Y++
c8
// Y -> A (A=0x33)
98
// A -> X
aa
// LDA #$20 (A=0x20)
a9
20
// X -> S (S=0x33)
9a
// LDX #$10 (X=0x10)
a2
10
// S -> X (X=0x33)
ba
// X -> A (A=0x33)
8a
// STA $40 ($40=0x33)
85
40
///////////////////////////////////////////
//
// CHECK TEST 05
//
a5
40
cd
05
02
f0
08
a9
05
8d
10
02
4c
ca
f5
///////////////////////////////////////////
//
// TEST 06
//
// Flag setup...
2A
///////////////////////////////////////////
// 6502 Test #06
// Heather Justice 3/20/08
// Tests instructions ADC & SBC with all
// addressing modes.
// Assumes that loads & stores work with
// all addressing modes.
//
// EXPECTED RESULTS: $30=0x9D
//
// will result in $50=0x6A & $51=0x6B &
// $60=0xA1 & $61=0xA2
a9
6a
85
50
a9
6b
85
51
a9
a1
85
60
a9
a2
85
61
// will result in A=0x50
a9
ff
69
ff
69
ff
e9
ae
// will result in A=0x4F & X=0x50
85
40
a6
40
```

```
75
00
f5
01
// will result in A=0x4E
65
60
e5
61
// will result in A=0x24 & $0120=0x4E &
// $0121=0x4D
8d
20
01
a9
4d
8d
21
01
a9
23
6d
20
01
ed
21
01
// will result in A=0x28 & X=0x24 &
// $0124=0x64 & $0125=0x62
85
f0
a6
f0
a9
64
8d
24
01
a9
62
8d
25
01
a9
26
7d
00
01
fd
01
01
// will result in A=0x31 & Y=0x28 &
// $0128=0xE5 & $0129=0xE9
85
f1
a4
f1
a9
e5
8d
28
01
a9
e9
8d
29
01
a9
34
79
00
01
f9
01
01
// will result in A=0x16 & X=0x31 & $70-
// $73=20,01,24,01
85
f2
a6
f2
a9
20
85
70
a9
01
85
71
a9
24
85
72
a9
01
85
73
61
41
e1
3f
// will result in A=0x9D & Y=0x16 &
// $80=0xDA & $82=0xDC & $30=0x9D
85
f3
a4
f3
a9
da
85
80
a9
00
85
81
a9
dc
85
82
a9
00
85
83
a9
aa
71
80
f1
82
85
30
//////////////////////////////////////////
//
// CHECK TEST 06
//
a5
30
cd
06
02
f0
08
a9
06
8d
10
02
4c
ca
f5
//////////////////////////////////////////
//
// TEST 07
//
//////////////////////////////////////////
// 6502 Test #07
// Heather Justice 3/25/08
// Tests instructions CMP (all addressing
// modes) & BEQ & BNE.
// Assumes that loads & stores work with
// all addressing modes.
// Also assumes that AND & ORA & EOR work
// with all addressing modes.
//
// EXPECTED RESULTS: $15 = 0x7F
//
// prepare memory
a9
00
85
34
a9
ff
8d
30
01
a9
99
8d
9d
01
a9
db
8d
99
01
a9
2f
85
32
a9
32
85
4f
a9
30
85
33
a9
70
85
af
// A = 0x18
a9
18
85
30
// cmp imm...
// CMP #$18
c9
18
// BEQ beq1 ; taken
f0
02
// AND #$00 ; not done
29
00
// cmp zpg...
// ORA #$01 (A = 0x19)
09
01
// CMP $30
c5
30
// BNE bne1 ; taken
d0
02
// AND #$00 ; not done
29
00
// cmp abs...
// LDX #$00 (X = 0x00)
a2
00
// CMP $0130
cd
30
01
// BEQ beq2 ; not taken
f0
04
// STA $40
85
40
// LDX $40 (X = 0x19)
a6
40
// cmp zpx...
// CMP $27,X
d5
27
// BNE bne2 ; not taken
d0
06
// ORA #$84 (A = 0x9D)
09
84
// STA $41
85
41
// LDX $41 (X = 0x9D)
a6
41
// cmp abx...
// AND #$DB (A = 0x99)
29
db
// CMP $0100,X
dd
00
01
// BEQ beq3 ; taken
f0
02
// AND #$00 ; not done
29
00
// cmp aby...
// STA $42
85
42
// LDY $42 (Y = 0x99)
a4
42
// AND #$00 (A = 0x00)
29
00
// CMP $0100,Y
d9
00
01
// BNE bne3 ; taken
d0
02
// ORA #$0F ; not done
09
0f
// cmp idx...
// STA $43
85
43
// LDX $43 (X = 0x00)
a6
43
// ORA #$24 (A = 0x24)
09
24
// CMP ($40,X)
c1
40
// BEQ beq4 ; not taken
f0
02
// ORA #$7F (A = 0x7F)
09
7f
// cmp idy...
// STA $44
85
44
// LDY $44 (Y = 0x7F)
a4
44
// EOR #$0F (A = 0x70)
```

```
49
0f
// CMP ($33),Y
d1
33
// BNE bne4 ; not taken
d0
04
// LDA $44 (A = 0x7F)
a5
44
// STA $15 ($15 = 0x7F)
85
15
///////////////////////////////////////////
//
// CHECK TEST 07
//
a5
15
cd
07
02
f0
08
a9
07
8d
10
02
4c
ca
f5
/////////////////////////////////////////////
// TEST 08
//
///////////////////////////////////////////
// 6502 Test #08
// Heather Justice 3/25/08
// Tests instructions CPX & CPY & BIT for
// all addressing modes.
// Assumes that loads & stores (with all
// addressing modes) & BEQ & BNE work.
// Also assumes that AND & ORA & EOR work
// with all addressing modes.
//
// EXPECTED RESULTS: $42 = 0xA5
//
// prepare memory...
a9
a5
85
20
8d
20
01
a9
5a
85
21
// cpx imm...
// LDX #$A5 (X = 0xA5)
a2
a5
// CPX #$A5
e0
a5
// BEQ b1 ; taken
f0
02
// LDX #$01 ; not done
a2
01
// cpx zpg...
// CPX $20
e4
20
// BEQ b2 ; taken
f0
02
// LDX #$02 ; not done
a2
02
// cpx abs...
// CPX $0120
ec
20
01
// BEQ b3 ; taken
f0
02
// LDX #$03 ; not done
a2
03
// cpy imm...
// STX $30
86
30
// LDY $30 (Y = 0xA5)
a4
30
// CPY #$A5
c0
a5
// BEQ b4 ; taken
f0
02
// LDY #$04 ; not done
a0
04
// cpy zpg...
// CPY $20
c4
20
// BEQ b5 ; taken
f0
02
// LDY #$05 ; not done
a0
05
// cpy abs...
// CPY $0120
cc
20
01
// BEQ b6 ; taken
f0
02
// LDY #$06 ; not done
a0
06
// bit zpg...
// STY $31
84
31
// LDA $31 (A = 0xA5)
a5
31
// BIT $20
24
20
// BNE b7 ; taken
d0
02
// LDA #$07 ; not done
a9
07
// bit abs...
// BIT $0120
2c
20
01
// BNE b8 ; taken
d0
02
// LDA #$08 ; not done
a9
08
// BIT $21
24
21
// BNE b9 ; not taken
```

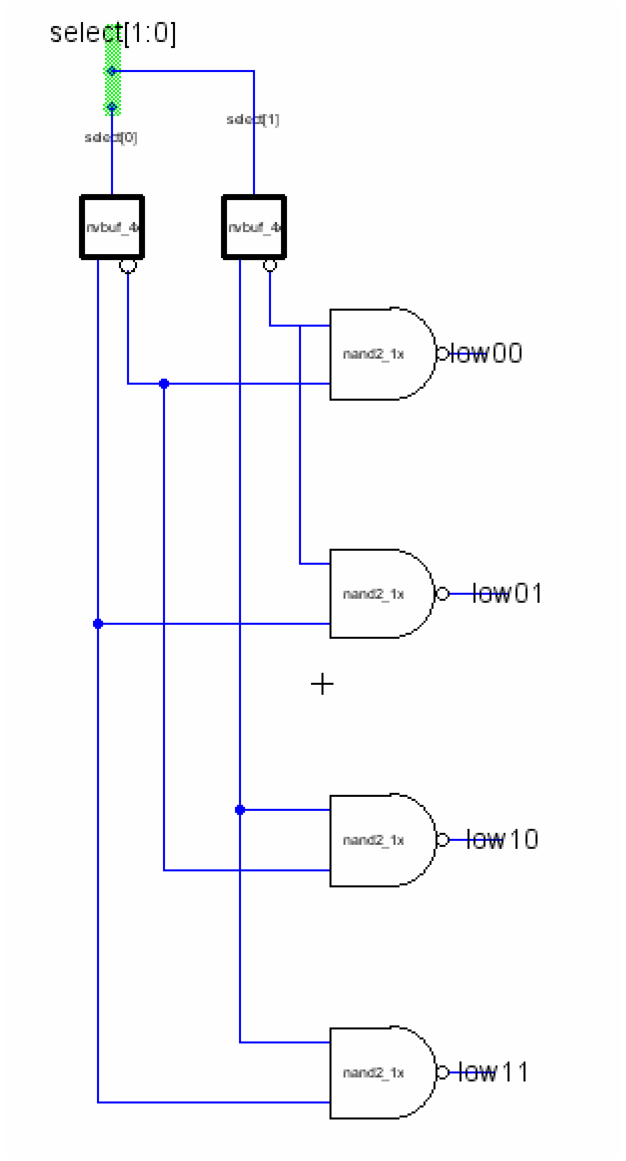
```
d0
02
// STA $42 ($42 = 0xA5)
85
42
/////////////////////////////////////////////
// CHECK TEST 08
//
a5
42
cd
08
02
f0
08
a9
08
8d
10
02
4c
ca
f5
///////////////////////////////////////////
//
// TEST 09
//
///////////////////////////////////////////
// 6502 Test #09
// Heather Justice 3/26/08
// Tests all other branch instructions (BPL
// & BMI & BVC & BVS & BCC & BCS).
// Assumes that ADC & SBC & EOR work with
// all addressing modes.
//
// EXPECTED RESULTS: $80 = 0x1F
//
// prepare memory...
// LDA #$54
a9
54
// STA $32 ($32 = 0x54)
85
32
// LDA #$B3
a9
b3
// STA $A1 ($A1 = 0xB3)
85
a1
// LDA #$87
a9
87
// STA $43 ($43 = 0x87 & A = 0x87)
85
43
// bpl...
// LDX #$A1 (X = 0xA1)
a2
a1
// BPL bpl1 ; not taken
10
02
// LDX #$32 (X = 0x32)
a2
32
// LDY $00,X (Y = 0x54)
b4
00
// BPL bpl2 ; taken
10
04
// LDA #$05 ; not done
a9
05
// LDX $A1 ; not done
a6
a1
// bmi...
// BMI bmi1 ; not taken
30
02
// SBC #$03 (A = 0x83)
e9
03
// BMI bmi2 ; taken
30
02
// LDA #$41 ; not done
a9
41
// bvc...
// EOR #$30 (A = 0xB3)
49
30
// STA $32 ($32 = 0xB3)
85
32
// ADC $00,X (A = 0x67)
75
00
// BVC bvc1 ; not taken
50
02
// LDA #$03 (A = 0x03)
a9
03
// STA $54 ($54 = 0x03)
85
54
// LDX $00,Y (X = 0x03)
b6
00
// ADC $51,X (A = 0x07)
75
51
// BVC bvc2 ; taken
50
02
// LDA #$E5 ; not done
a9
e5
// bvs...
// ADC $40,X (A = 0x8E)
75
40
// BVS bvs1 ; not taken
70
04
// STA $0001,Y ($55 = 0x8E)
99
01
00
// ADC $55 (A = 0x1C)
65
55
// BVS bvs2 ; taken
70
02
// LDA #$00
a9
00
// bcc...
// ADC #$F0 (A = 0x0D)
69
f0
// BCC bcc1 ; not taken
90
04
// STA $60 ($60 = 0x0D)
85
60
// ADC $43 (A = 95)
65
43
// BCC bcc2 ; taken
90
02
// LDA #$FF
a9
ff
// bcs...
// ADC $54 (A = 0x98)
65
54
// BCS bcs1 ; not taken
b0
04
// ADC #$87 (A = 0x1F)
69
87
// LDX $60 (X = 0x0D)
a6
60
// BCS bcs2 ; taken
b0
02
// LDA #$00 ; not done
a9
00
// STA $73,X ($80 = 0x1F)
95
73
/////////////////////////////////////////////
// CHECK TEST 09
//
a5
80
cd
09
02
f0
08
a9
09
8d
10
02
4c
ca
f5
///////////////////////////////////////////
//
// TEST 10
//
// Flag setup...
69
00
///////////////////////////////////////////
// 6502 Test #10
// Heather Justice 3/26/08
// Tests flag instructions (CLC & SEC & CLV
// & CLD & SED) & NOP.
// Assumes that loads & stores (all
// addressing modes) and all branches work.
// Also assumes ADC works with all
// addressing modes.
//
// EXPECTED RESULTS: $30 = 0xCE
//
// LDA #$99 (A = 0x99)
a9
99
// ADC #$87 (A = 0x20)
69
87
// CLC
18
// NOP
ea
// BCC bcc1 ; taken
90
04
// ADC #$60 ; not done
69
60
// ADC #$93 ; not done
69
93
// SEC
38
// NOP
ea
// BCC bcc2 ; not taken
90
01
// CLV
b8
// BVC bvc1 ; taken
50
02
// LDA #$00 ; not done
a9
00
// ADC #$AD (A = 0xCE)
69
ad
// NOP
ea
// STA $30 ($30 = 0xCE)
85
30
///////////////////////////////////////////
//
// CHECK TEST 10
//
a5
30
cd
0a
02
f0
08
a9
0a
8d
10
02
4c
ca
f5
///////////////////////////////////////////
//
// TEST 11
//
// Flag setup...
69
01
///////////////////////////////////////////
// 6502 Test #11
// Heather Justice 3/26/08
// Tests stack instructions (PHA & PLA &
// PHP & PLP).
// Assumes that loads & stores (all
// addressing modes).
// Also assumes ADC (all addressing modes)
// and all flag instructions work.
//
// EXPECTED RESULTS: $30 = 0x29
//
// LDA #$27 (A = 0x27)
a9
27
// ADC #$01 (A = 0x28)
69
01
// SEC
38
// PHP
08
// CLC
18
// PLP
28
// ADC #$00 (A = 0x29)
69
00
// PHA
48
// LDA #$00 (A = 0x00)
a9
00
// PLA (A = 0x29)
68
```

```
// STA $30 ($30 = 0x29)
85
30
//////////////////////////////////////////
//
// CHECK TEST 11
//
a5
30
cd
0b
02
f0
08
a9
0b
8d
10
02
4c
ca
f5
//////////////////////////////////////////
//
// TEST 12
//
//////////////////////////////////////////
// 6502 Test #12
// Heather Justice 4/10/08
// Tests RTI instruction.
// Assumes lots of other instructions work
// already...
//
// EXPECTED RESULTS: $33 = 0x42
//
// CLC
18
// LDA #$42
a9
42
// BCC runstuff
90
04
// STA $33
85
33
// BCS end
b0
0a
// runstuff:
// LDA #$F5
a9
f5
// PHA
48
// LDA #$61
a9
61
// PHA
48
// SEC
38
// PHP
08
// CLC
18
// RTI
40
// end:
//////////////////////////////////////////
//
// CHECK TEST 12
//
a5
33
cd
0c
02
f0
08
a9
0c
8d
10
02
4c
ca
f5
//////////////////////////////////////////
//
// TEST 13
//
// Flag setup...
69
01
//////////////////////////////////////////
// 6502 Test #13
// Heather Justice 4/10/08
// Tests SEI & CLI & SED & CLD.
// Assumes prior tests pass...
//
// EXPECTED RESULT: $21 = 0x0C
//
// SEI
78
// SED
f8
// PHP
08
// PLA
68
// STA $20
85
20
// CLI
58
// CLD
d8
// PHP
08
// PLA
68
// ADC $20
65
20
// STA $21
85
21
////////////////////////////////////////////////
//////////////////////////////
//
// CHECK TEST 13
//
a5
21
cd
0d
02
f0
08
a9
0d
8d
10
02
4c
ca
f5
//////////////////////////////////////////
//
// TEST 14
//
//////////////////////////////////////////
// 6502 Test #14
// Heather Justice 4/23/08
// Tests BRK.
// Assumes prior tests pass...
//
// EXPECTED RESULT: $60 = 0x42
//
// JMP pass_intrp
4c
```

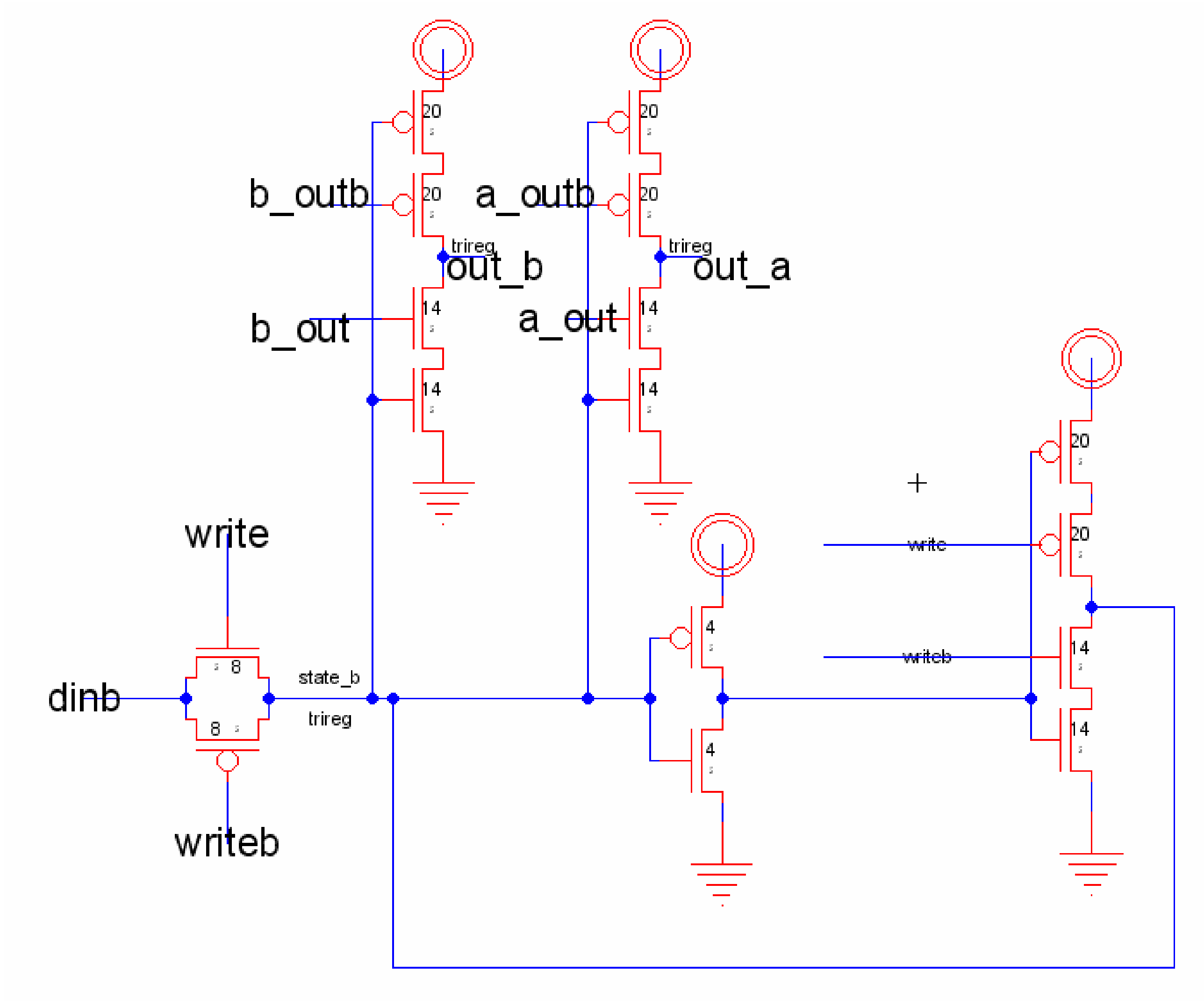
```
a9
f5
// LDA #$41
a9
41
// STA $60
85
60
// RTI
40
// pass_intrp:
// LDA #$FF
a9
ff
// STA $60
85
60
// BRK
00
00
// INC $60
e6
60
//////////////////////////////////////////
//
// CHECK TEST 14
//
a5
60
cd
0e
02
f0
08
a9
0e
8d
10
02
4c
ca
f5
////////////////////////////////////////////
// FINAL CONFIRMATION STAGE
//
a9
fe
cd
10
02
d0
03
ee
10
02
//////////////////////////////////////////
//
// theend : INFINITE LOOP!
//
4c
ca
f5
```

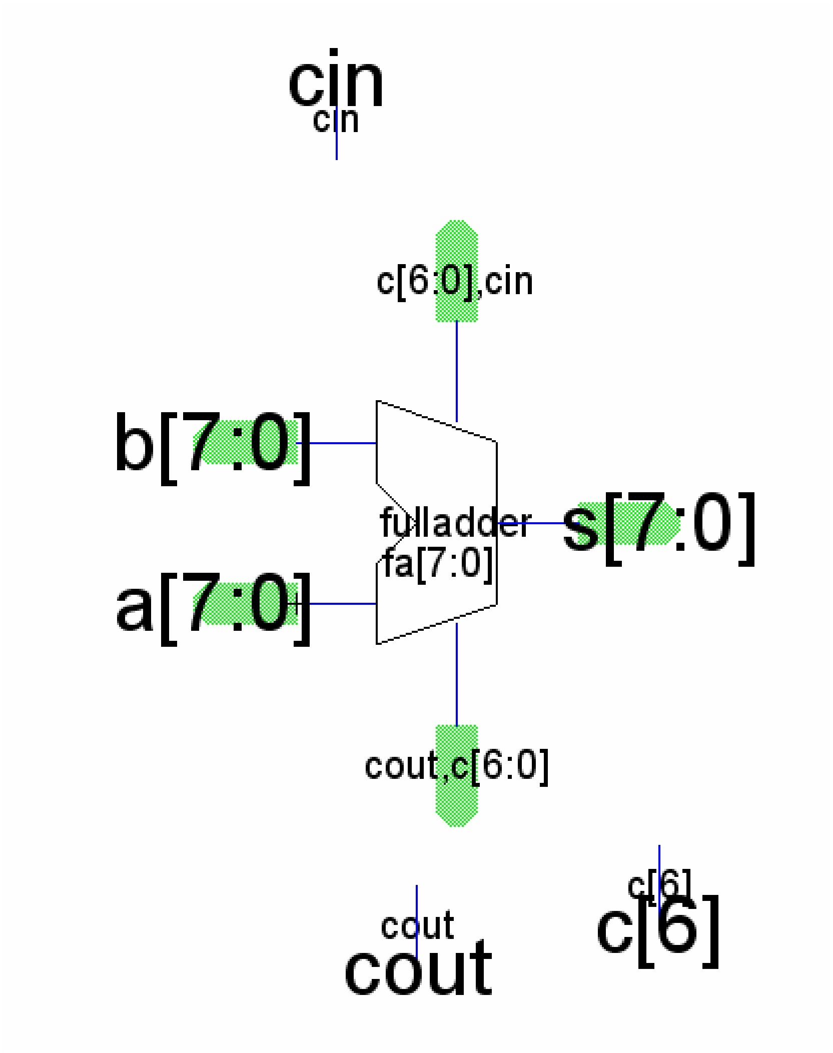
Appendix: Schematics

Following are screenshots of all schematic cells used in the Mudd II project. Captions describe design decisions, functions, and other interesting aspects of each cell.

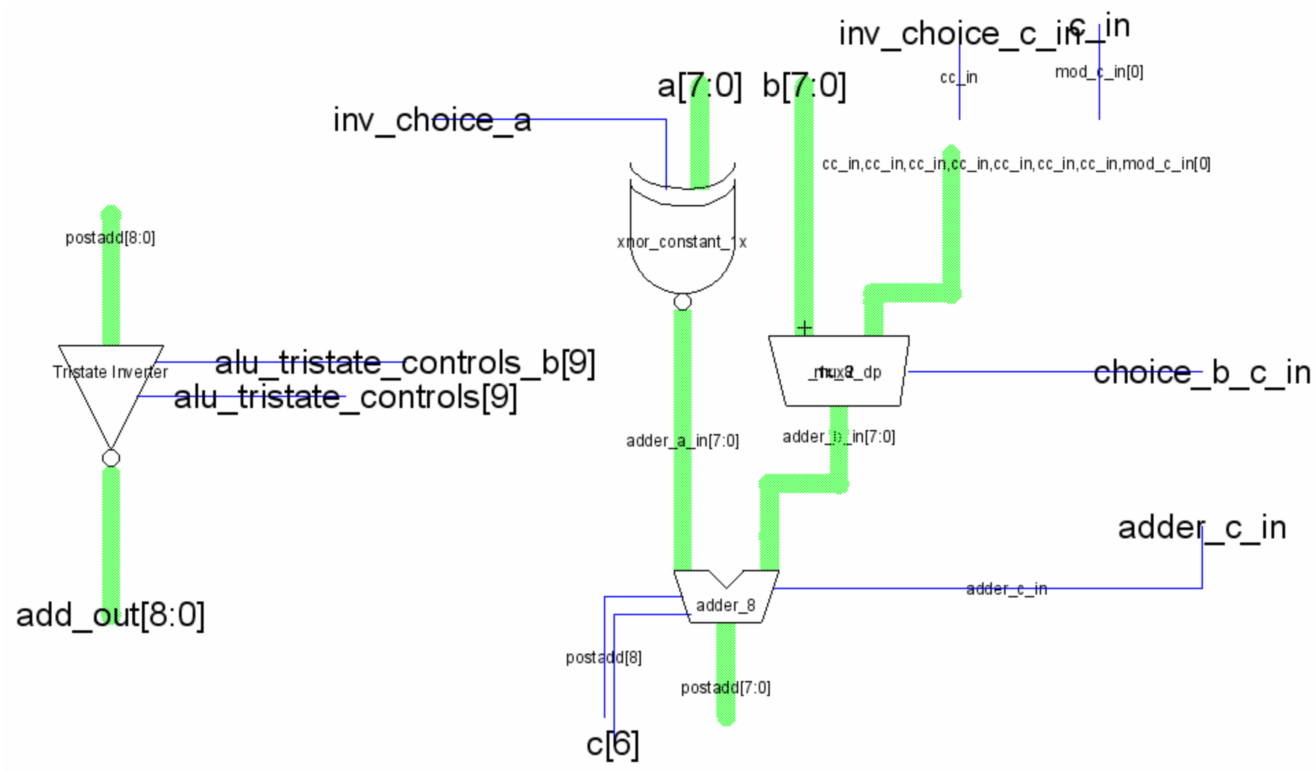


2bitdec_lowen – Register file control cell that decodes the 2-bit select signal.

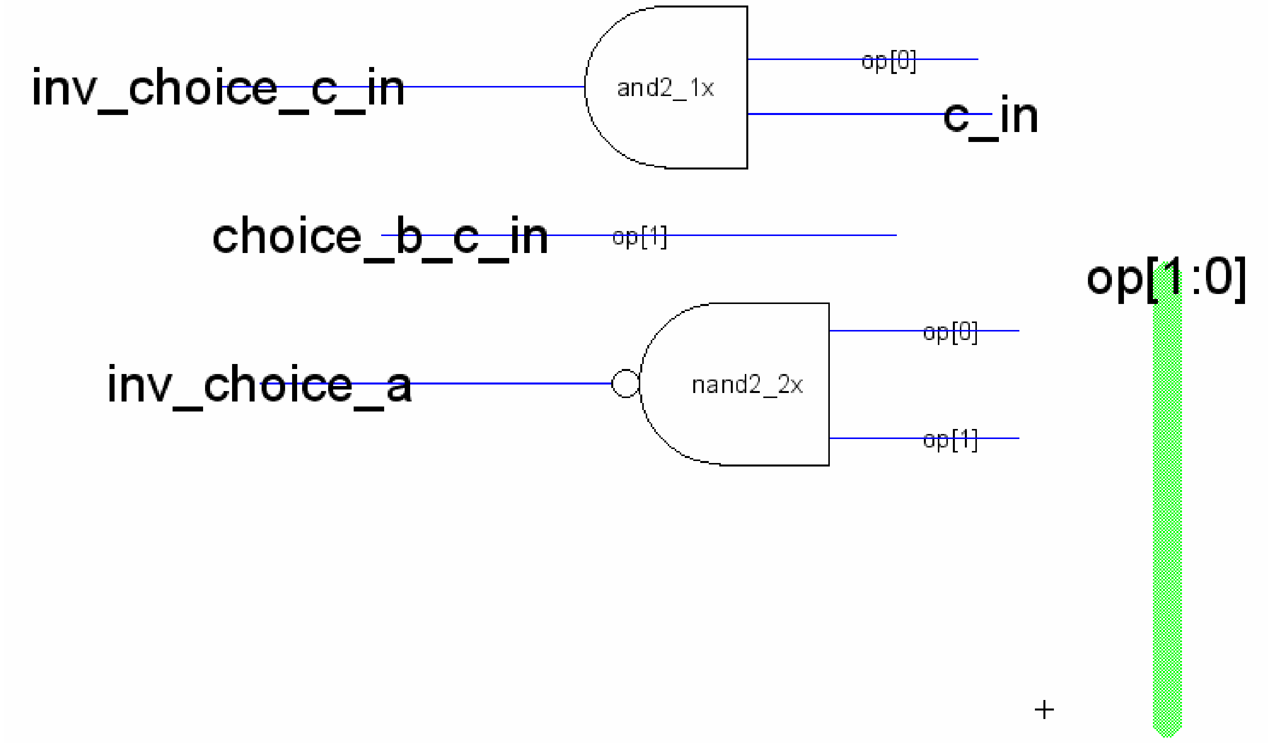

b_outb
a_outb
trireg
out_b
trireg
out_a
b_out
a_out
write
dinb
state_b
trireg
writeb
write
writeb
20
20
14
14
8
8
4
4



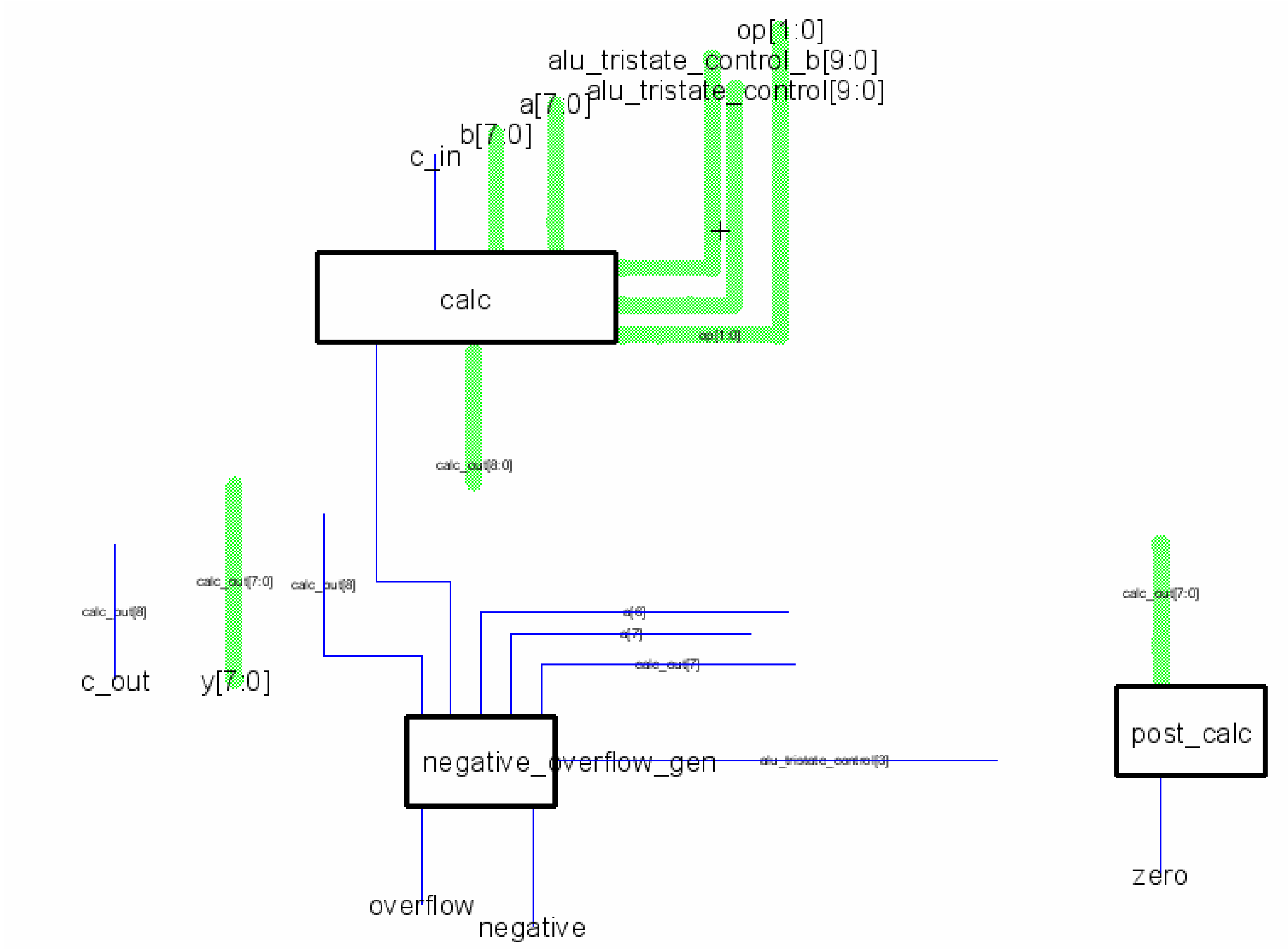
adder8 – 8-bit full adder used in the ALU



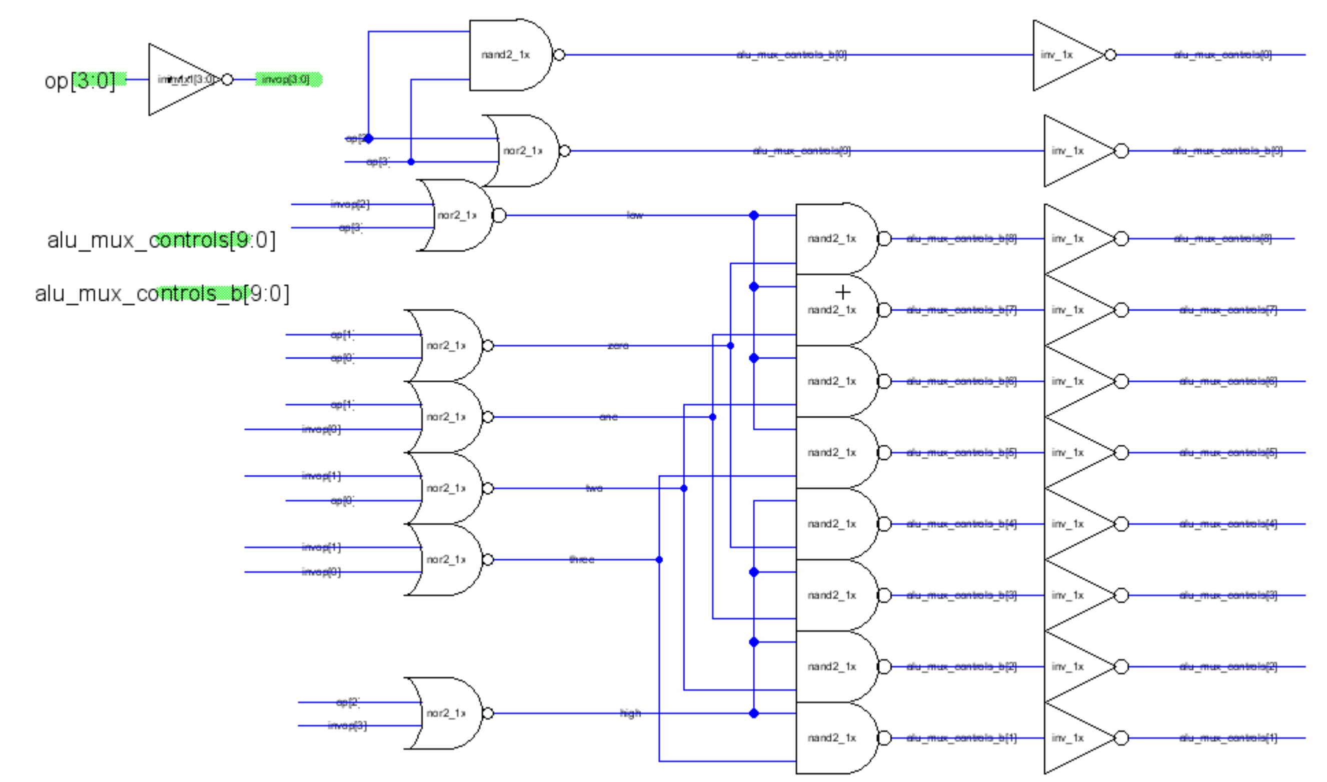
adder_group – ALU subcell consisting of full adder and supporting hardware.



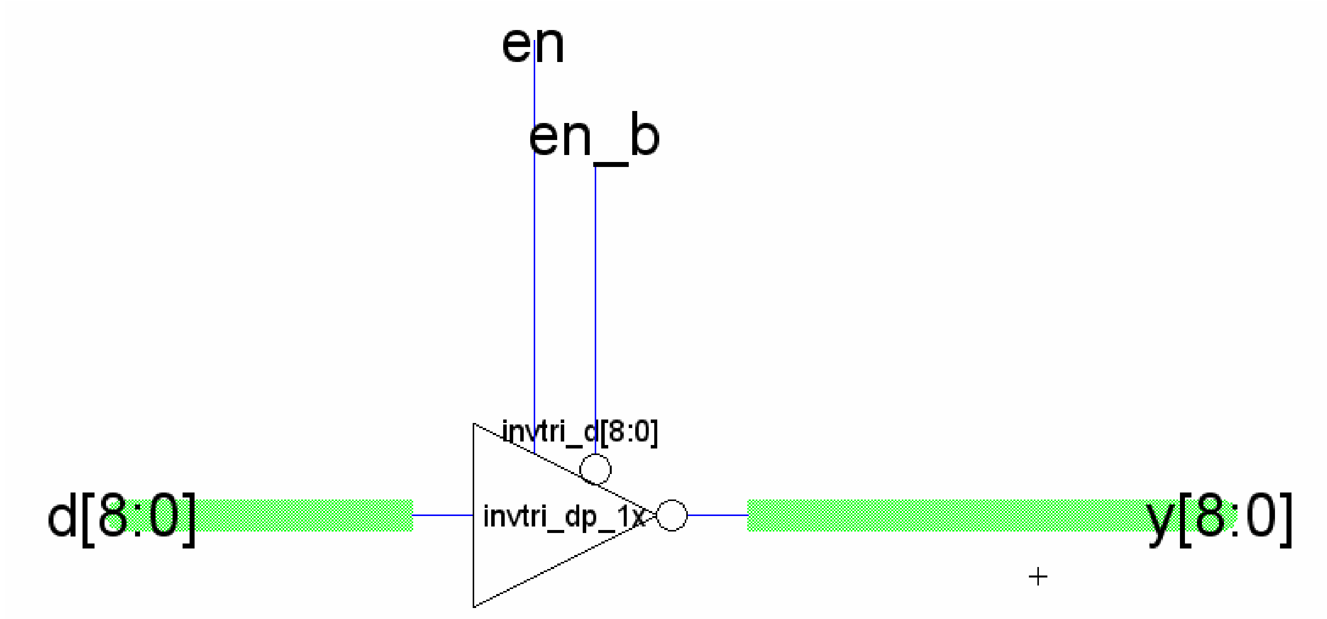
adder_group_controls – ALU subcell that provides control signals to adder_group block based on the received ALU opcode.



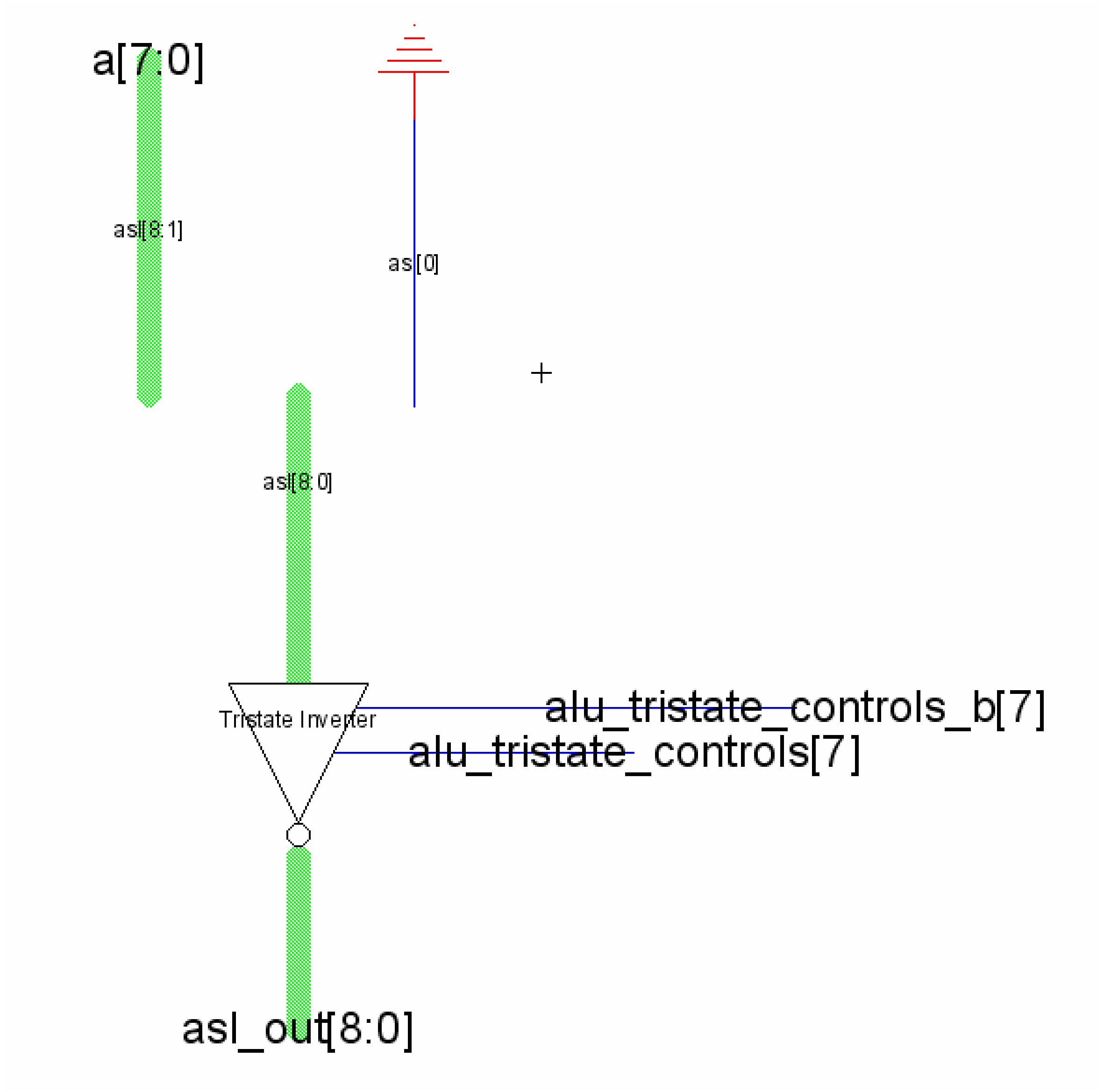
alu – Unit that performs various arithmetic and logic functions. Broken up into many hierarchical cells for layout purposes.



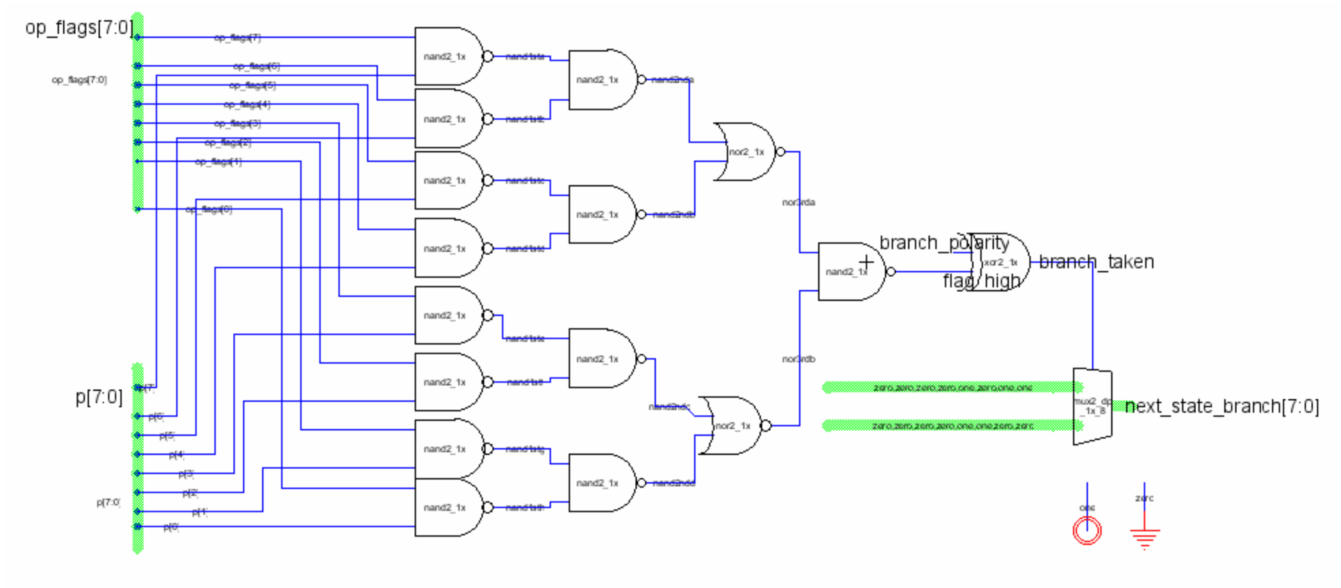
alu_tristate_controls – Translates the ALU opcode into 10 bits of control signal (with inverted values) that are used by the ALU in the calc subcell. Part of the controller.



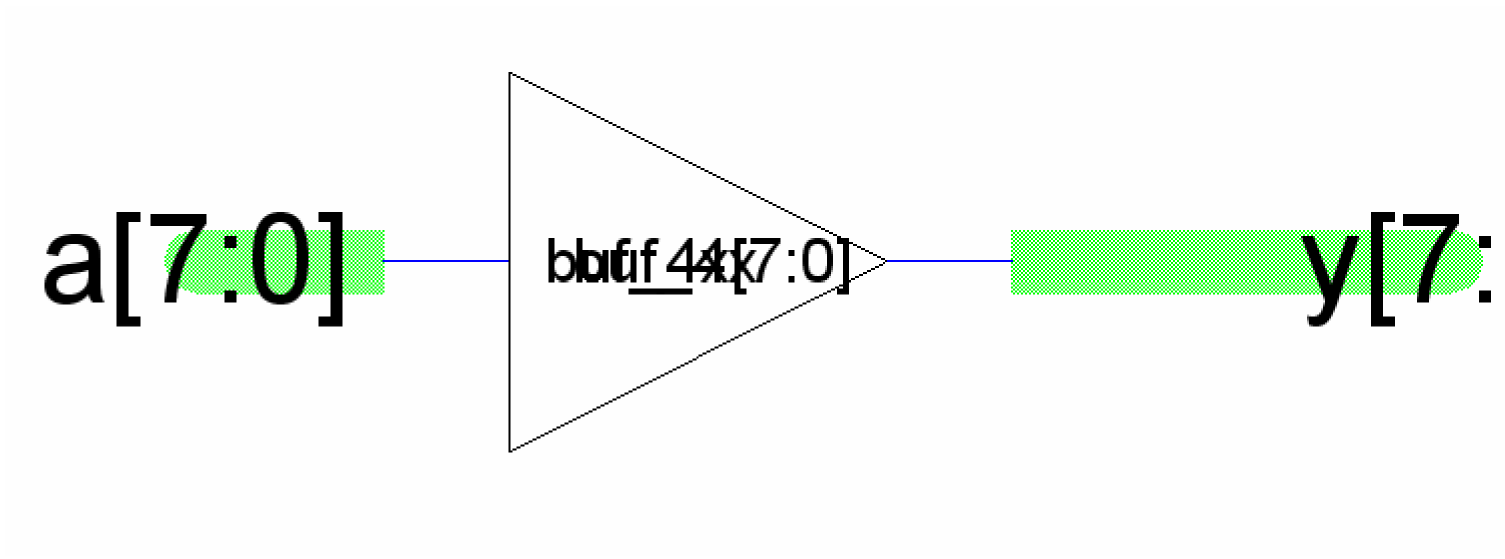
alu_tristate_9 – 9-bit tristate used in the ALU



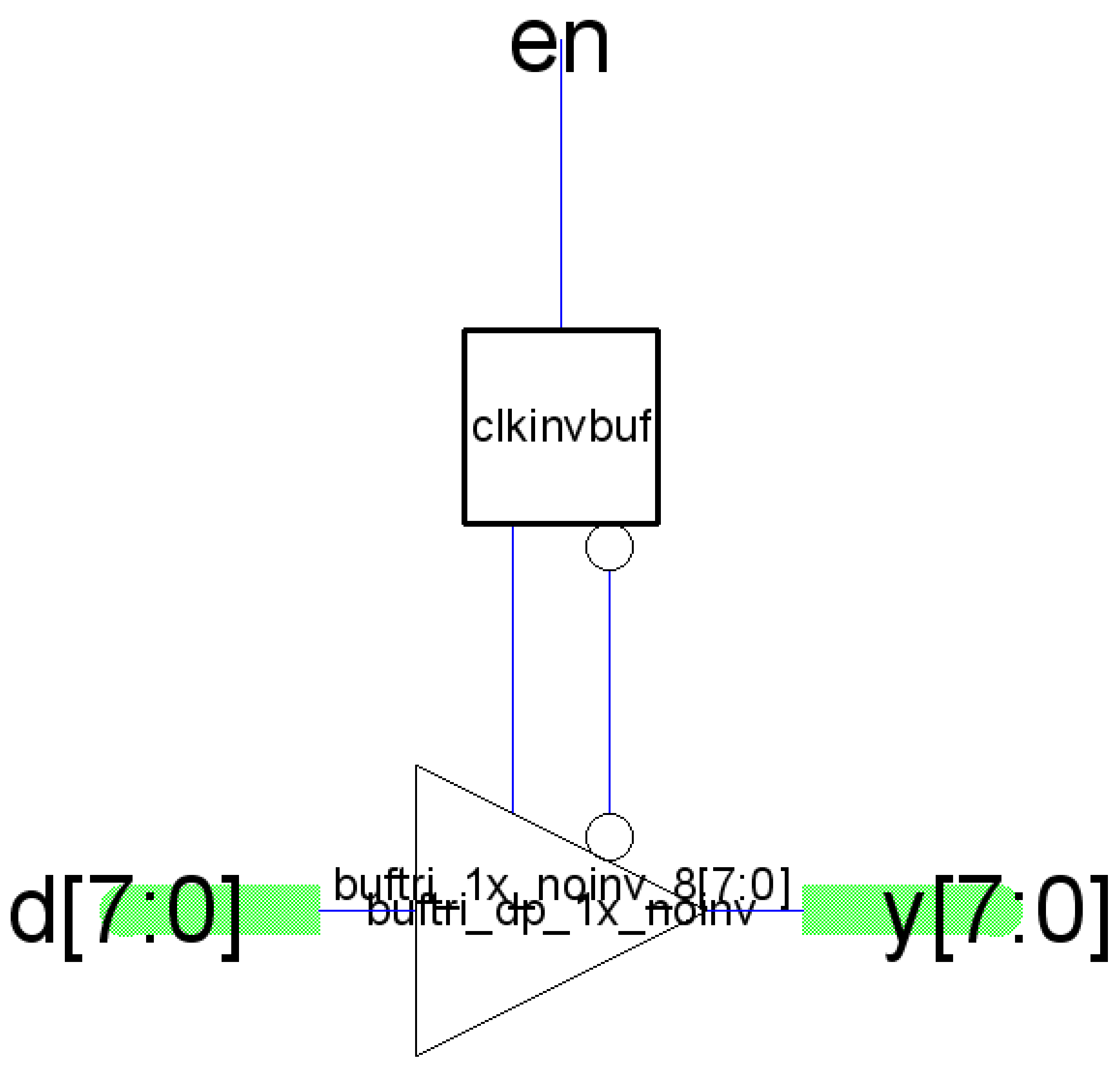
asl – ALU subcell performing the arithmetic left shift function



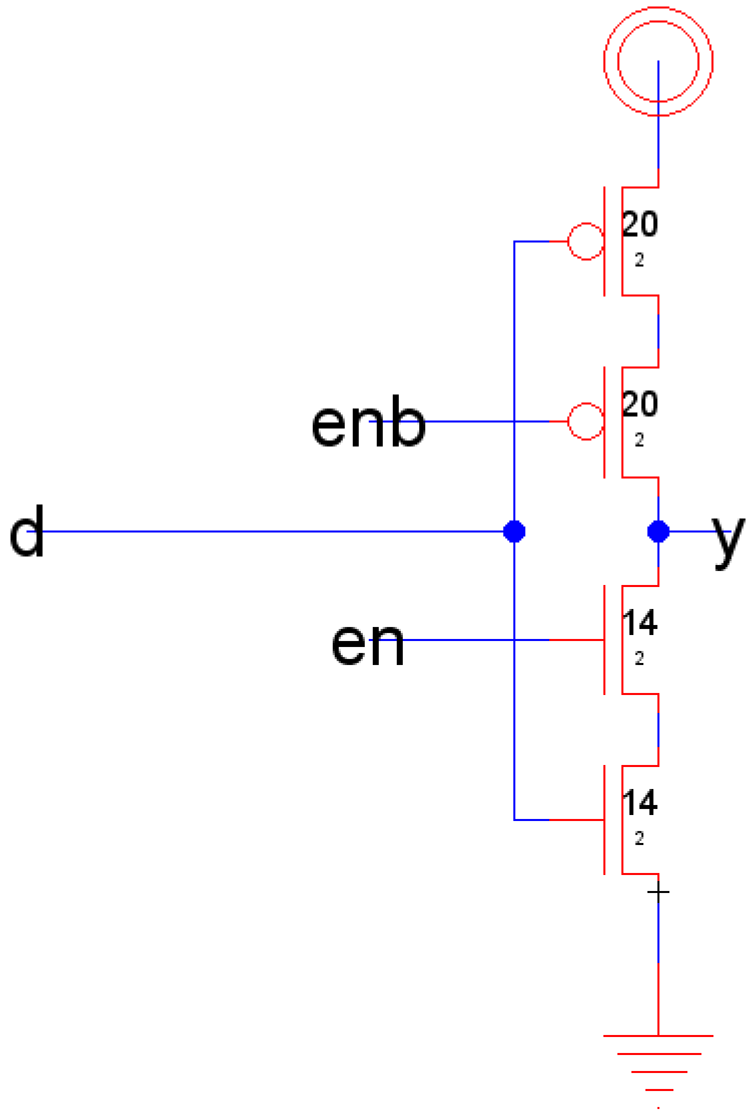
branchlogic – built to determine a branch is taken. If it has, a hard coded value is sent to the ROM.



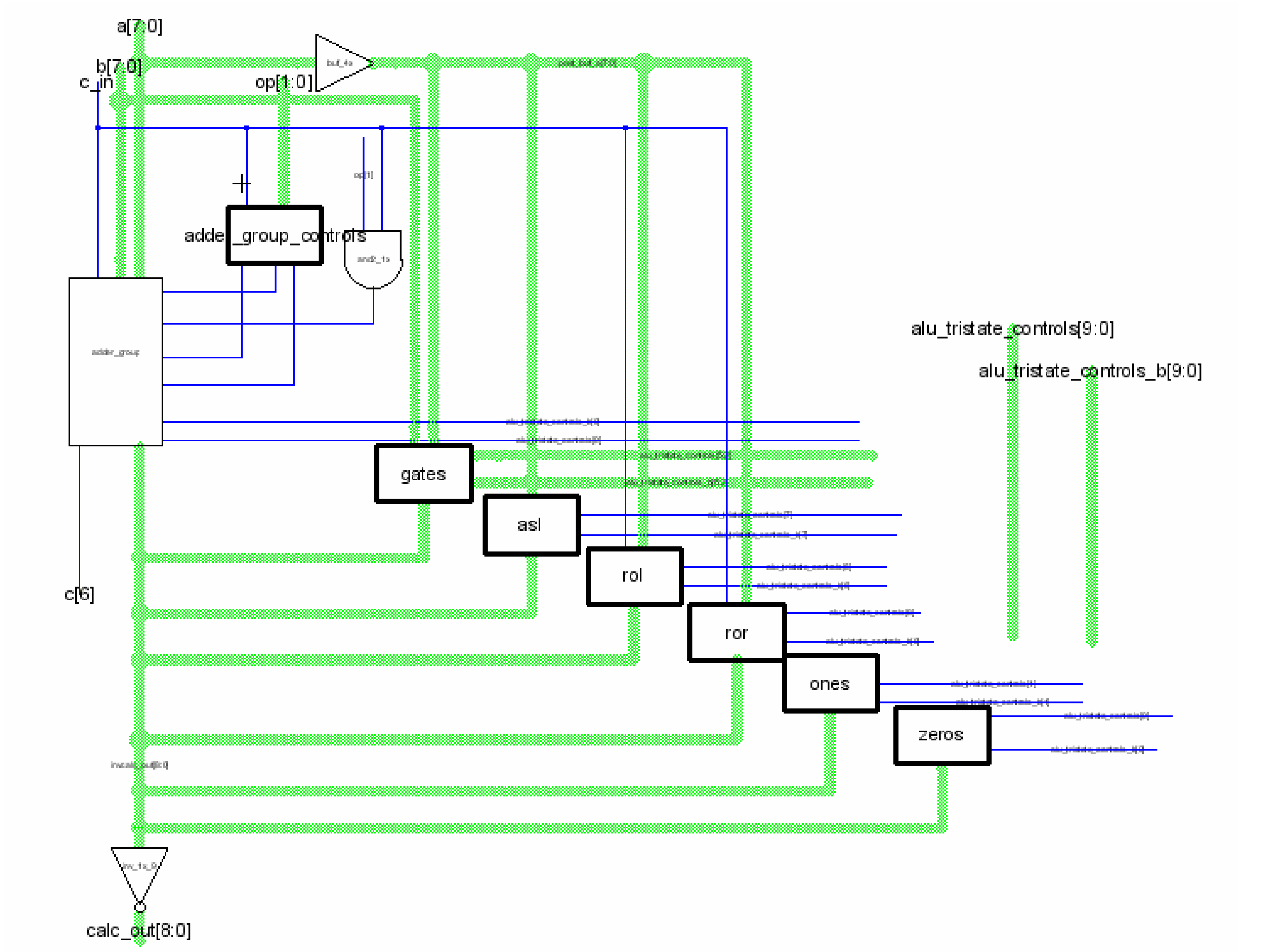
buf4x8 – eight bit 4x buffer



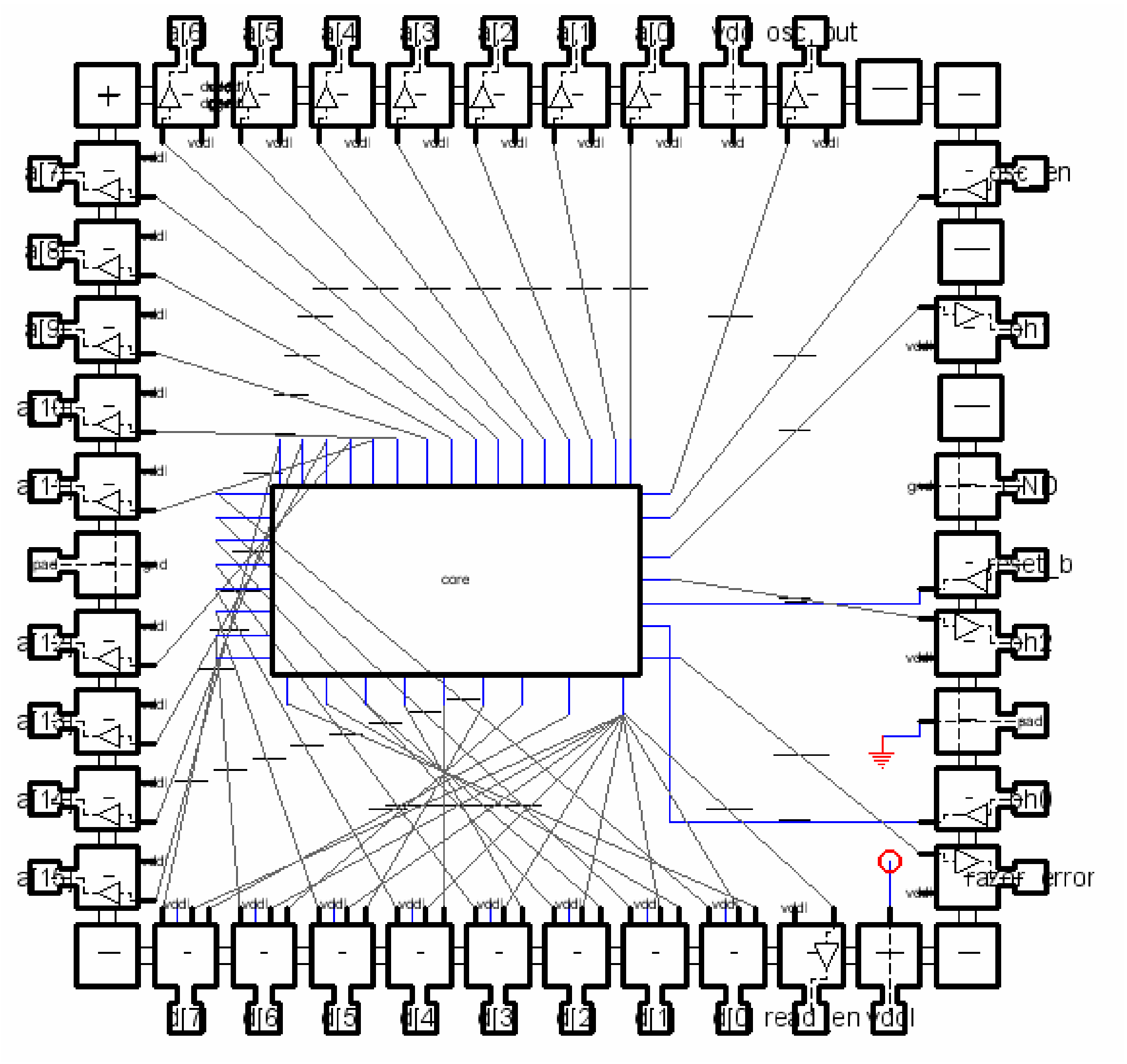
buftri_1x_8_noinv – an 8-bit non inverting tristate buffer



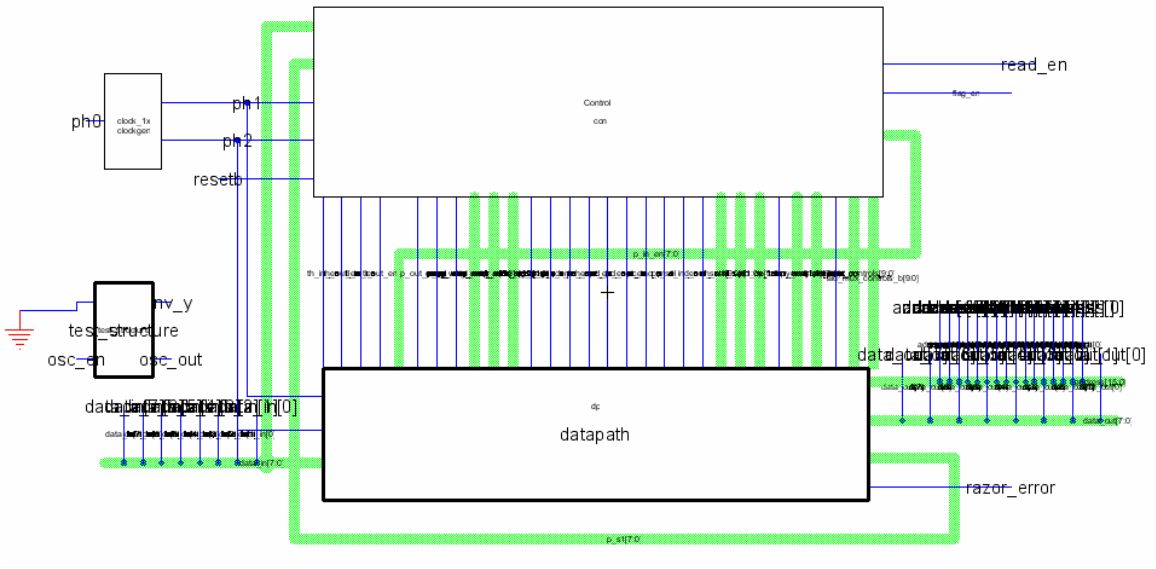
buftri_dp_1x_noinv – a transistor level schematic for the building block of buftri_1x_8_noinv.



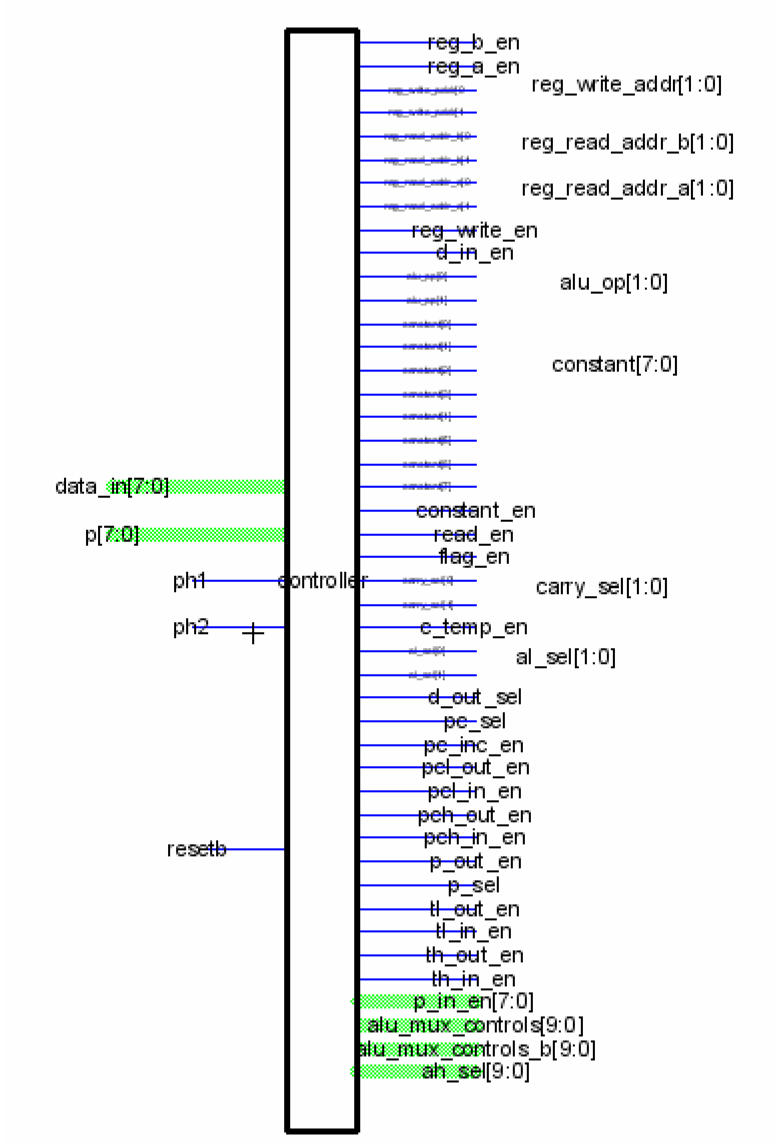
calc – ALU subcell that performs the main calculations. Takes in control signals from the zipper telling it which function to perform. Alu group controls and the and2_1x gate are in the zipper.



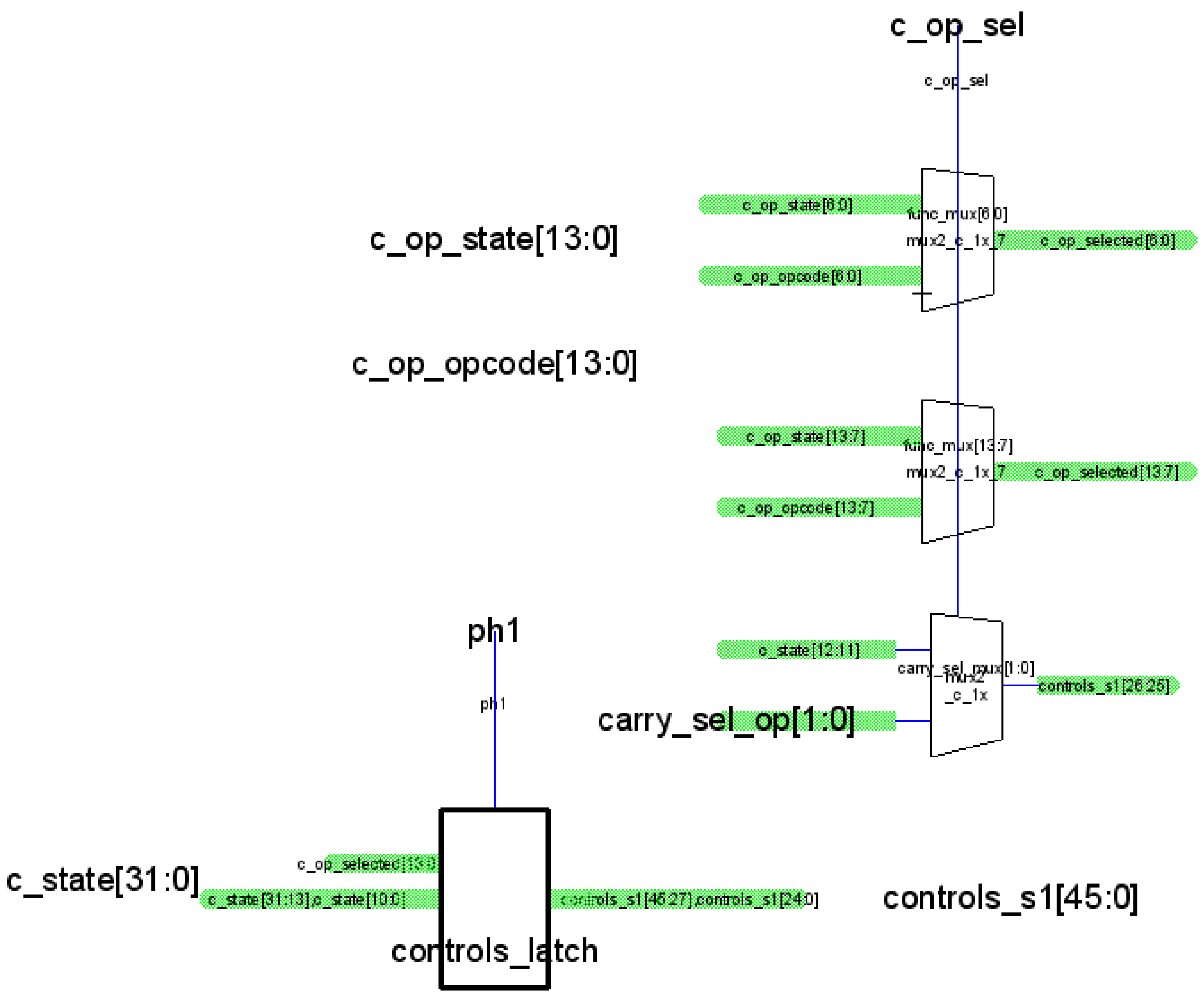
chip – top-level schematic for entire design. Auto-generated from core{sch} and pads.



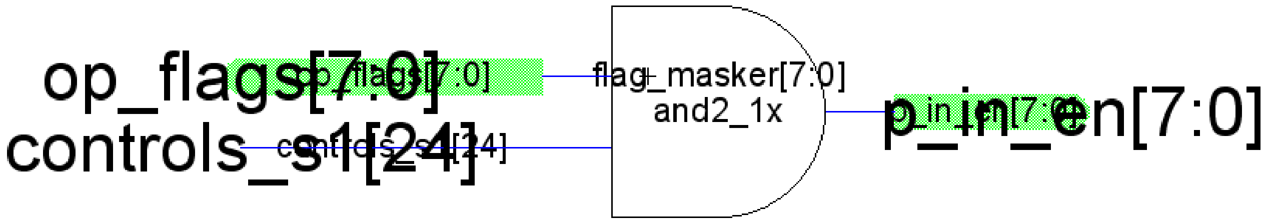
core – combination of controller and datapath. Also includes clock generator for producing two-phase non-overlapping clock and test structure for verifying basic functionality.



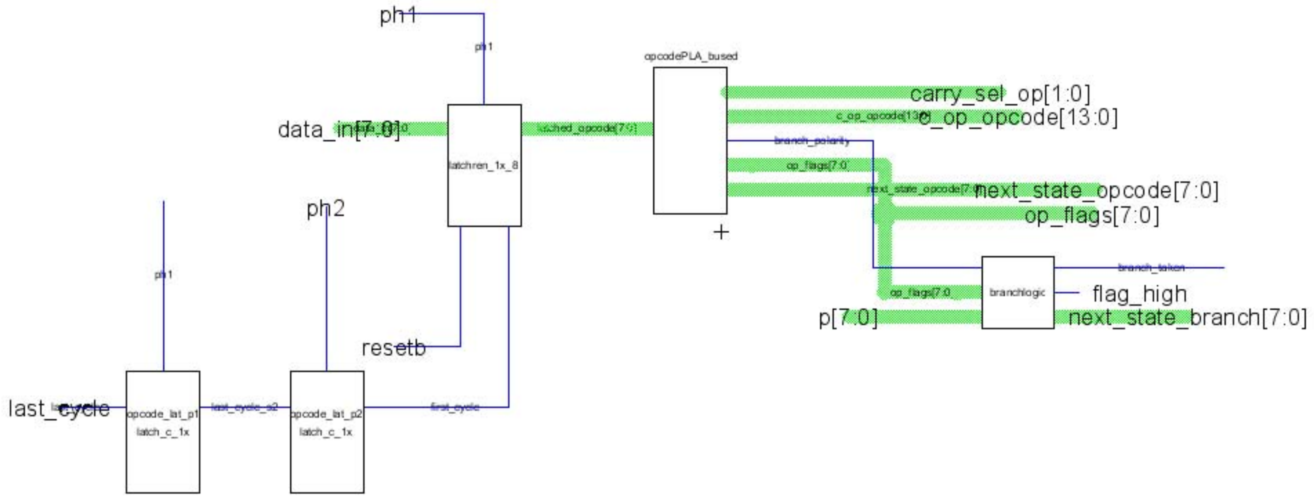
control – top-level schematic for controller block. Same functionality as controller{sch}, but exports are better labeled and icon is more suitable for creating core{sch}.



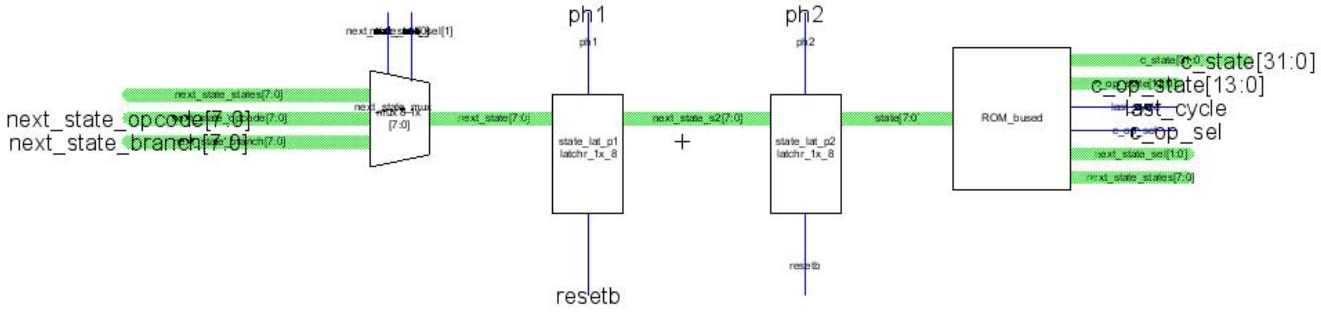
c_control_signals – a cell where the control signals would be selected using the muxes and then outputted through controls_latch. Note that controls_s1[26:25] are not latched, and are the selected carry_sel_op values. Also, within controls_latch, several signals are not latched. This was to save room and energy since these signals would in theory not affect anything if they were changed during the ph2 cycle.



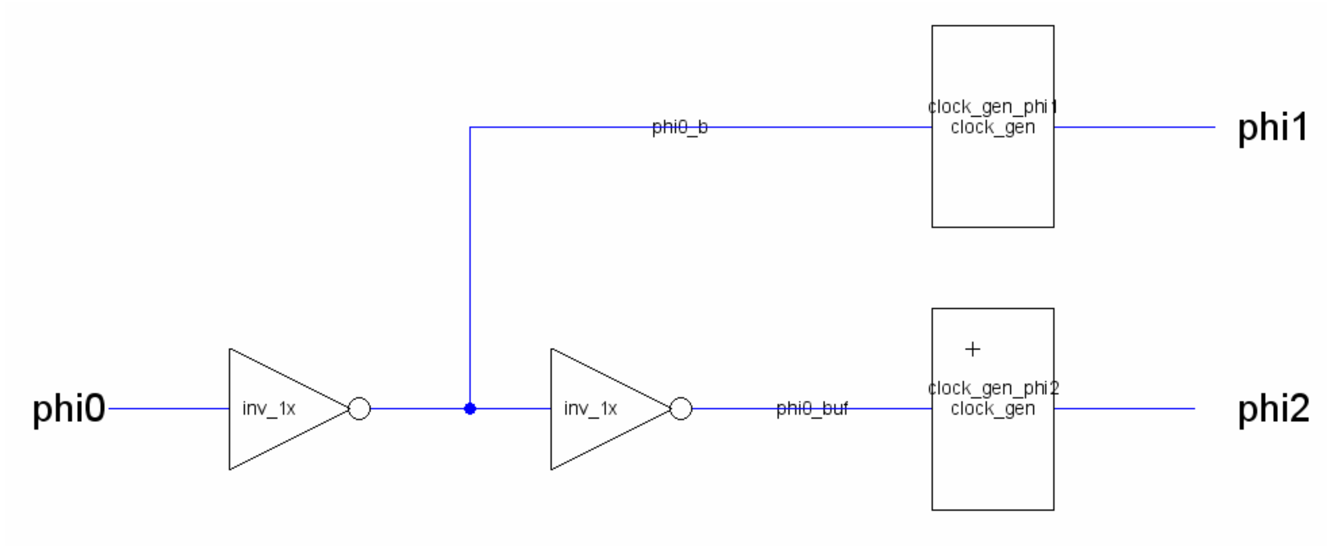
c_flag_masker – this cell takes in the op_flags and controls_s1[24]. Using controls_s1[24] as a masker, it then outputs p_in_en[7:0]



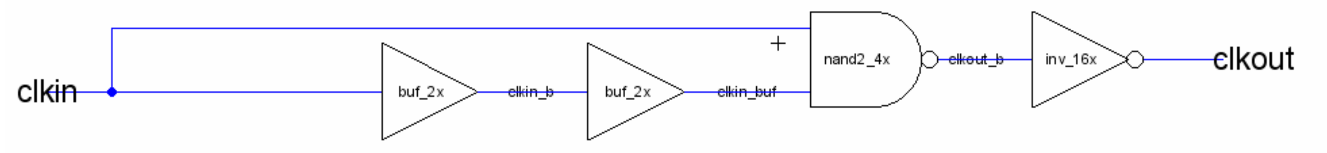
c_opcode – this cell is the controlling structure for the opcode ROM. The latched_opcode[7:0] signal from the latchren_1x_8 cell is the current opcode. It comes from memory by data_in[7:0], and is only written when first_cycle is asserted, a value that comes from the state ROM indicating the previous instruction has come to an end. Outputting from the opcode ROM is several signals, the most important being next_state_opcode, which tells the ROM which opcode state to enter next for a new instruction. Also utilizes branchlogic cell to determine the branch based on p, a set of bits that comes form the datapath.



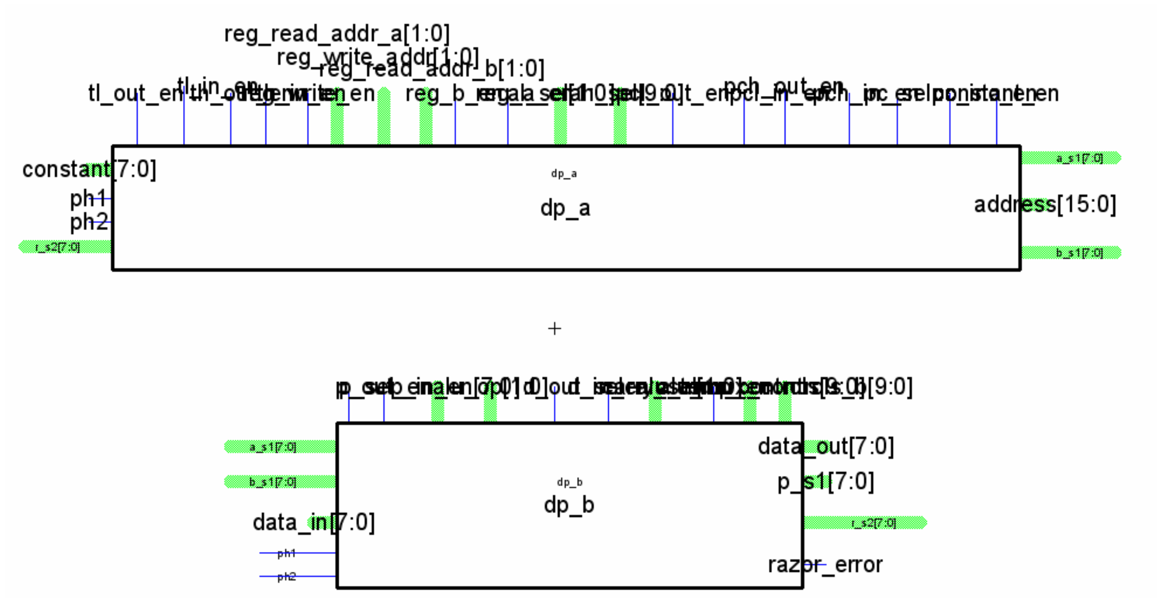
c_ROM – this particular cell is all the necessary components. The most important is the mux, since it chooses which signals the ROM uses as input. Next_state_opcode is used when a new instruction is loaded, next_state_states is when the ROM is progressing through the multiple states of a single instruction, and next_state_branch is used during branching operations.



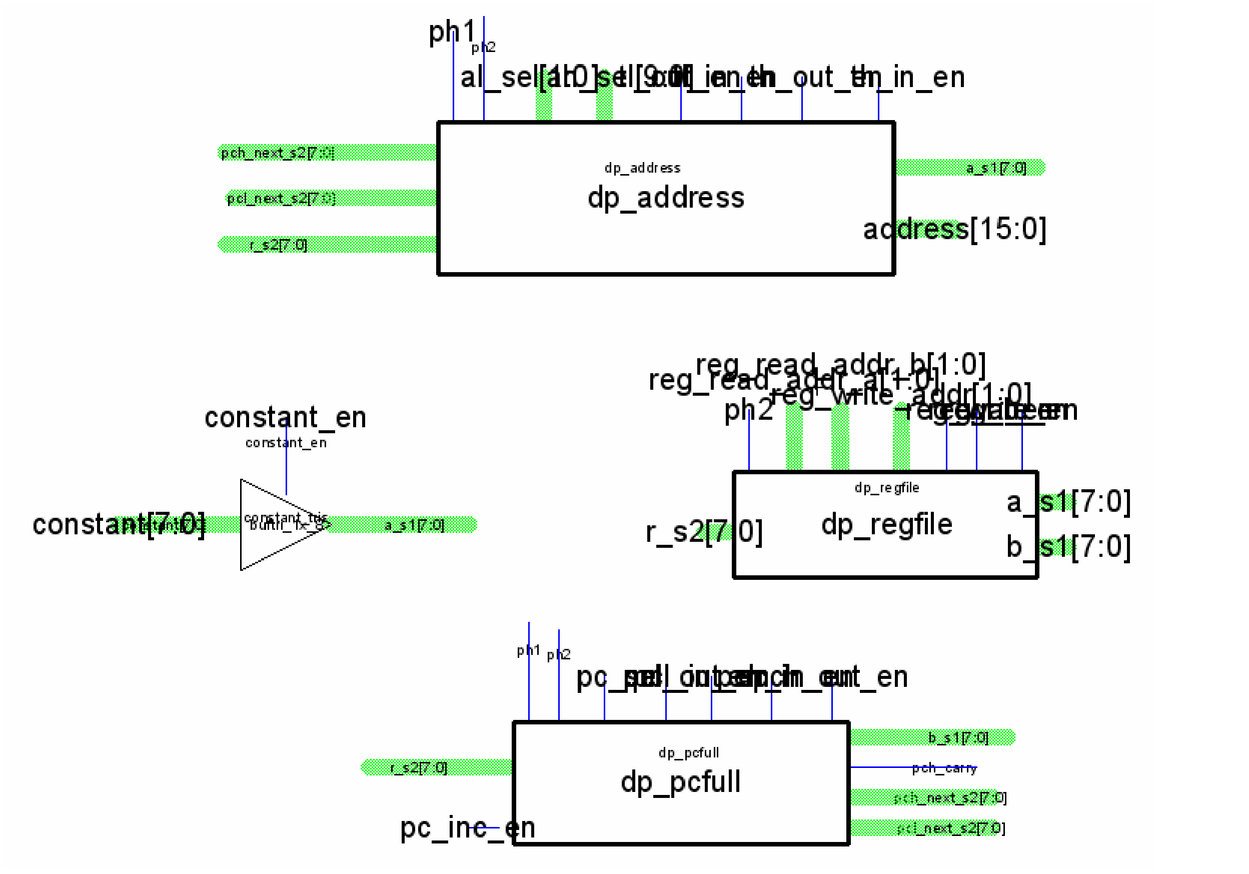
clock_1x – Generates two-phase non-overlapping clock from single-phase clock input.



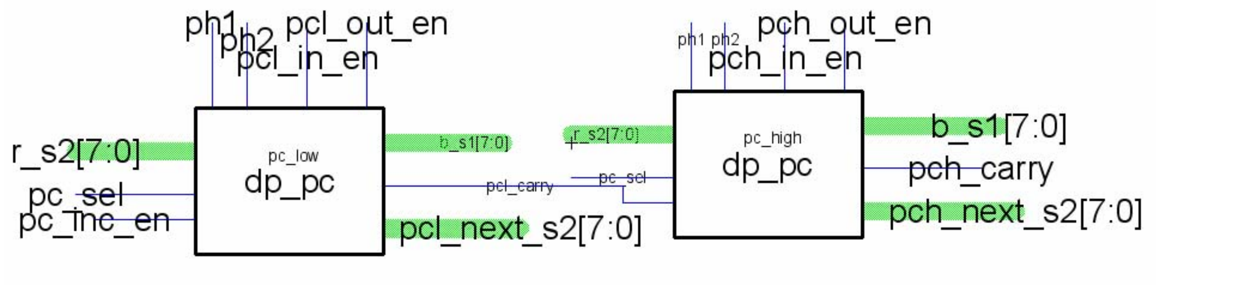
clock_gen – clock generator subcell that produces one of the two clock phases.



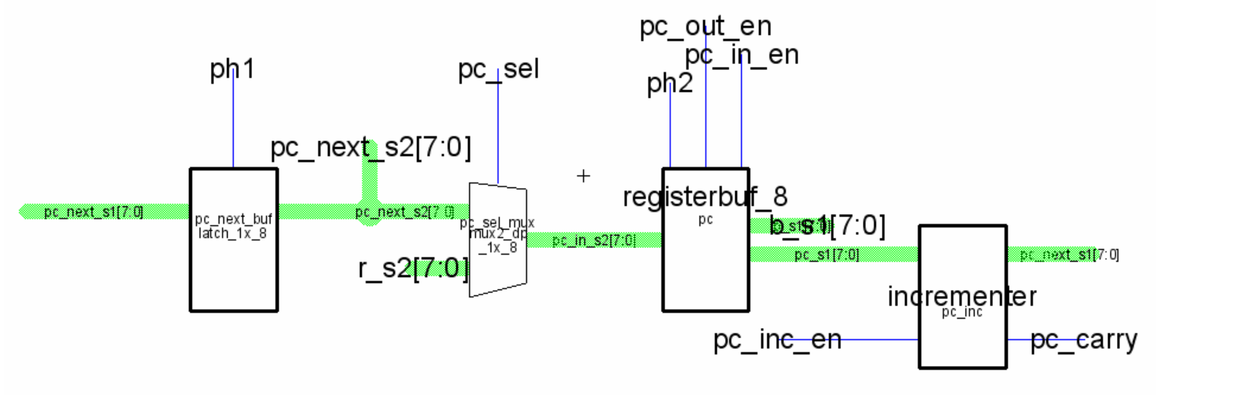
datapath – top-level schematic for datapath



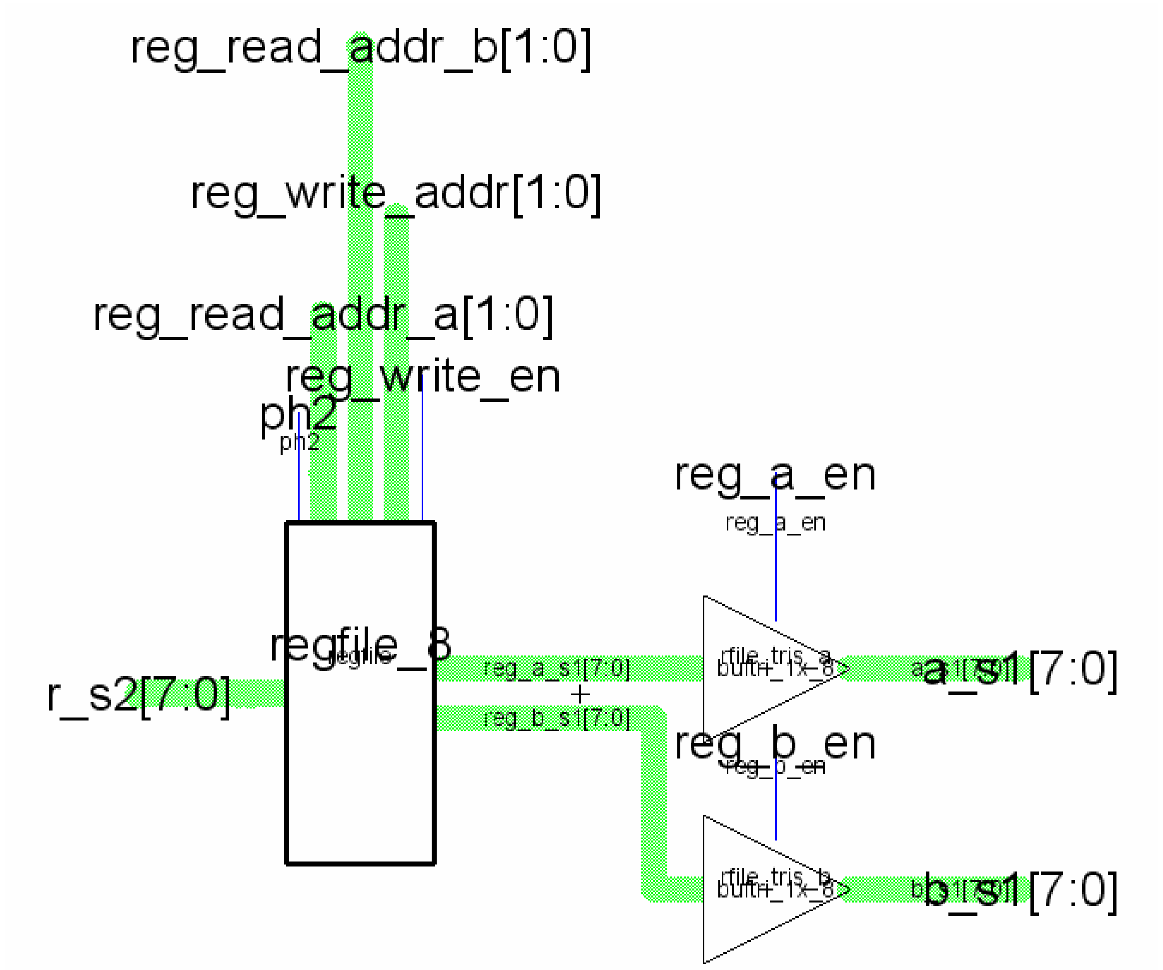
dp_a – half of the total datapath. Includes constant input, program counter, register file, and address determination.



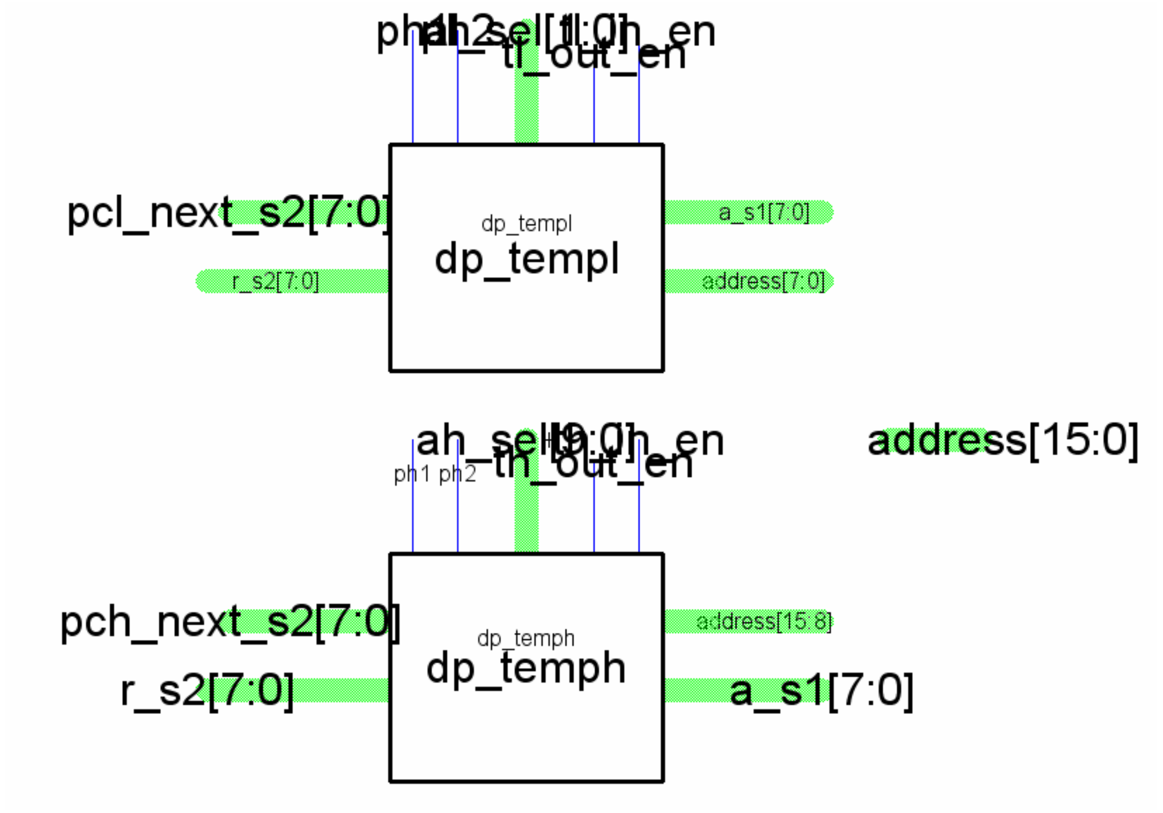
dp_pcfull – computes the program counter



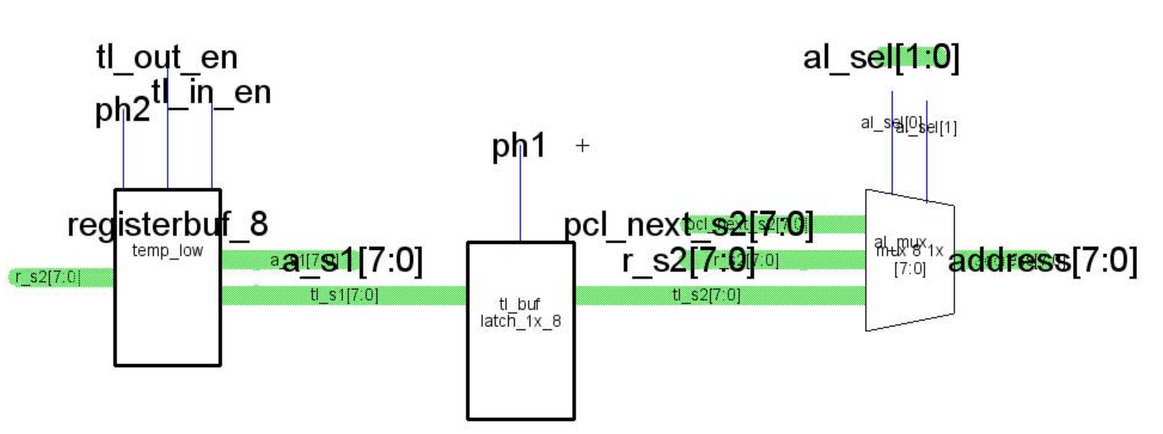
dp_pc – computes one byte of the program counter. Can also write to the b line.



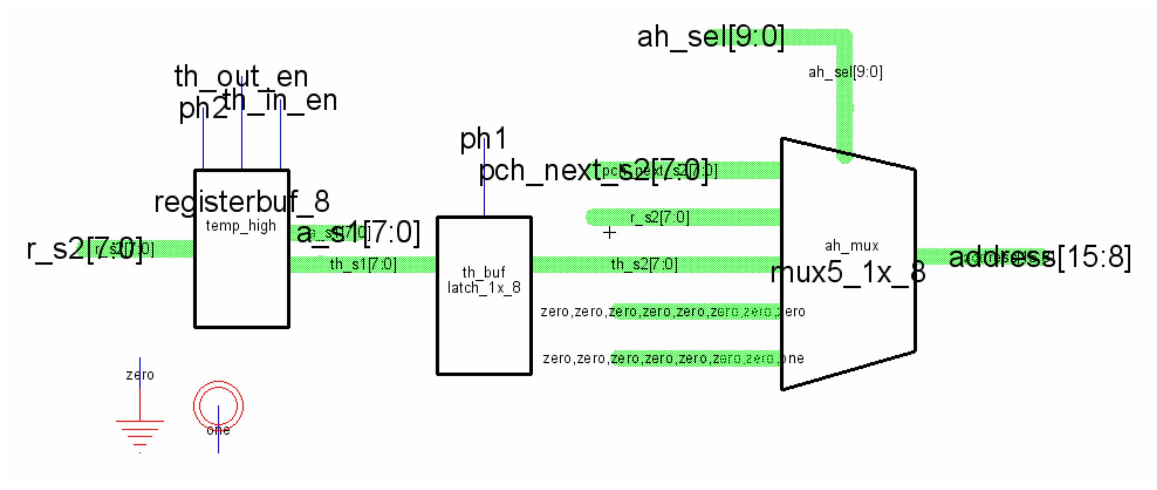
dp_regfile – register file block. Contains the regfile itself and tristates that allow writing onto the a line or b line.



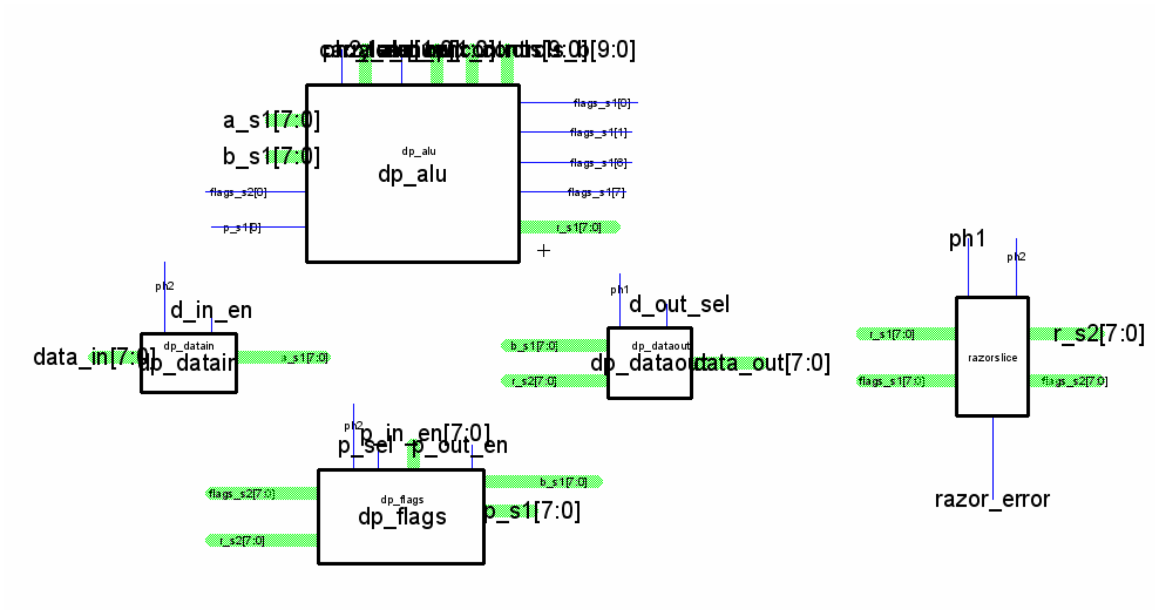
dp_address – computes both bytes of the address line.



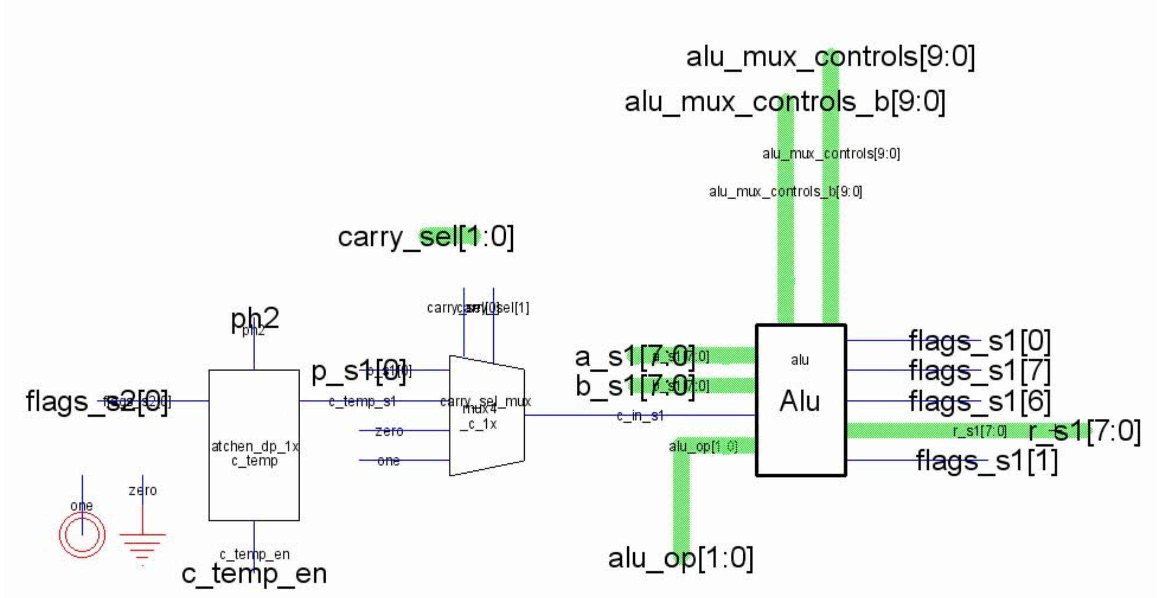
dp_templ – computes the lower 8 bits of the address line.



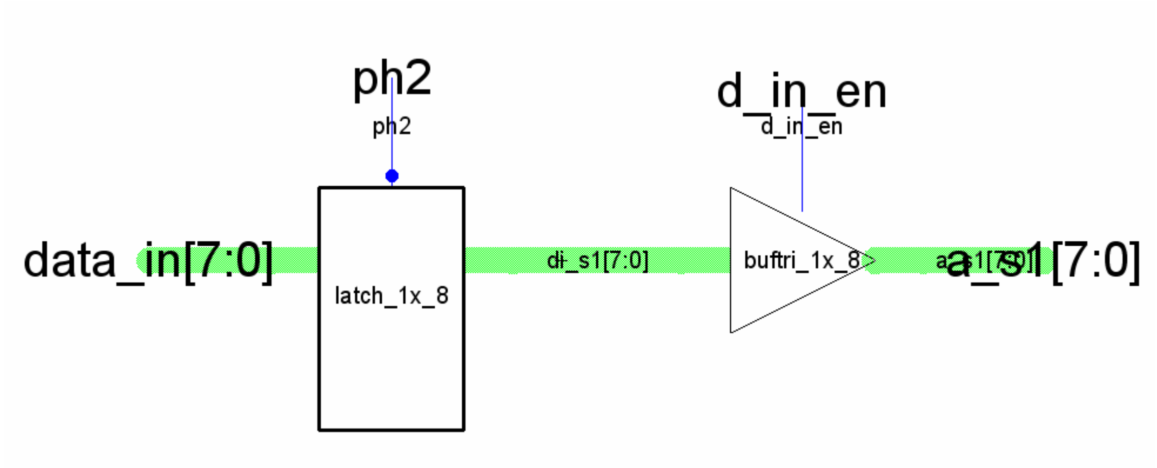
dp_temph – computes the upper 8 bits of the address line



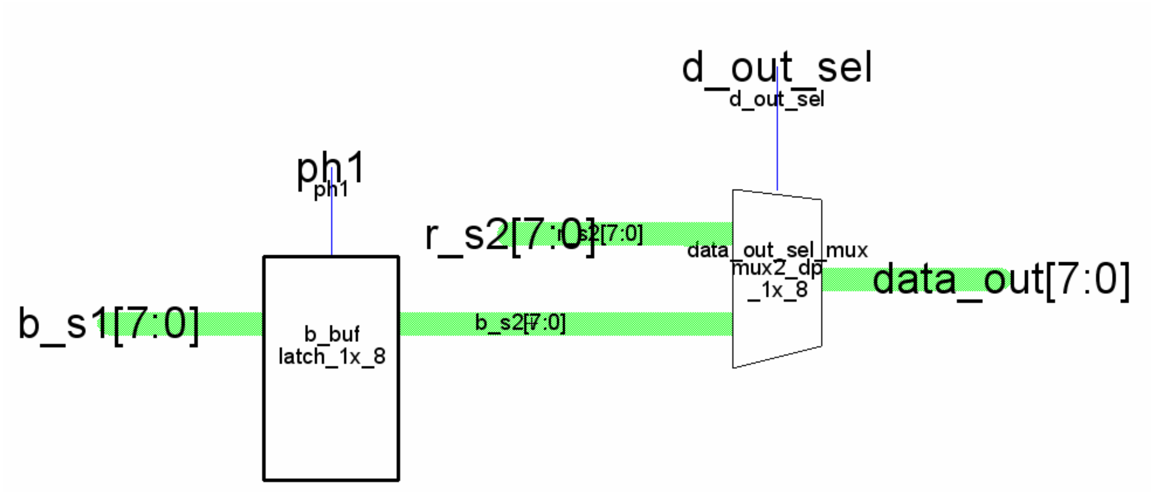
dp_b – half of total datapath. Includes alu, data in and out, flags, and razor latches.



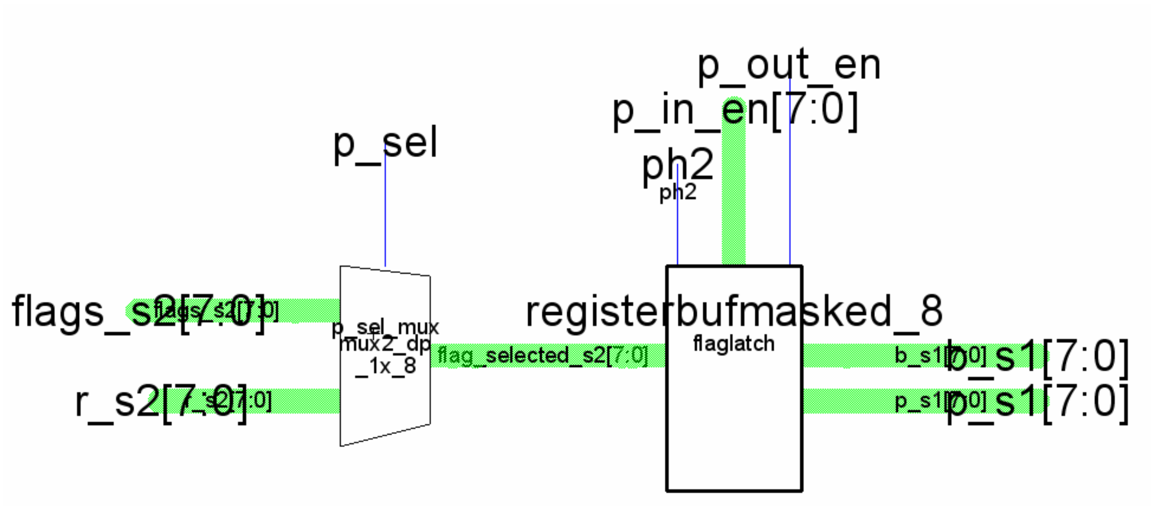
dp_alu – datapath block containing the alu. Also includes a mux and latch for carry in selection



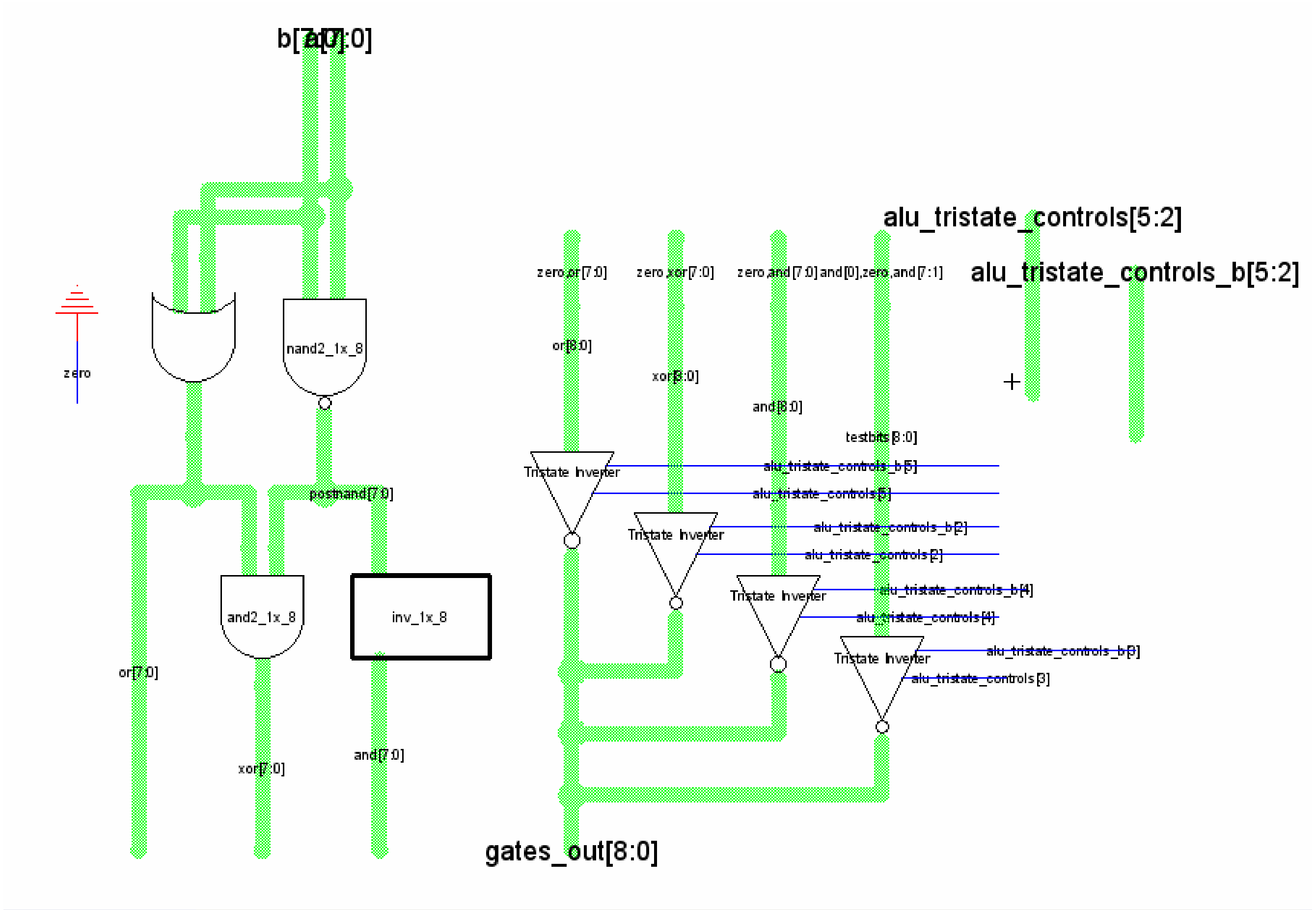
dp_datain – if d_in_en is high, the input data_in will be written onto the a line



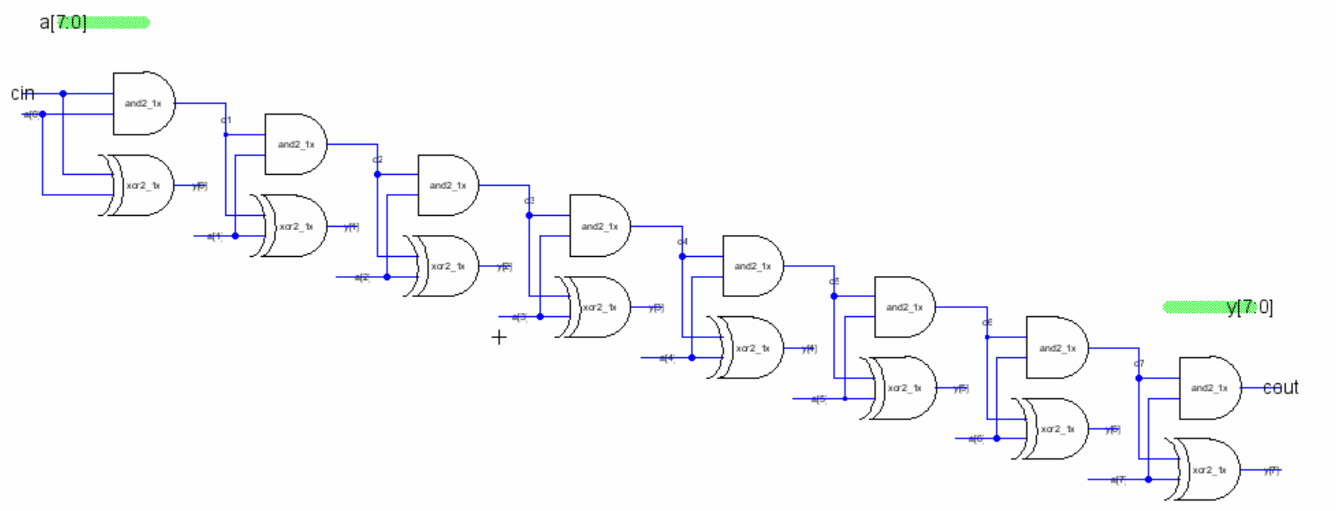
dp_dataout – based on d_out_sel, either the b or r line is written onto data_out. The b line is first made phase 2 stable via a latch.



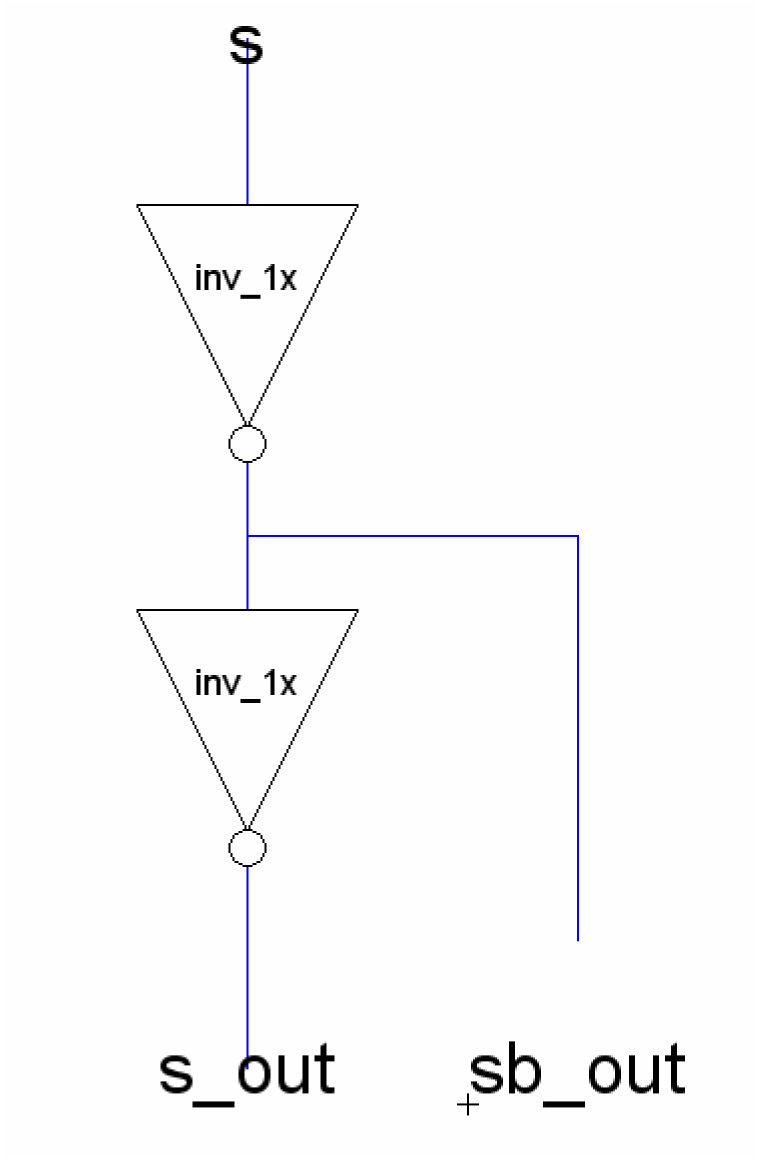
dp_flags – selects either flags_s2 or r_s2 to be written onto the b line. Also uses the flags to compute the p signal to the controller.



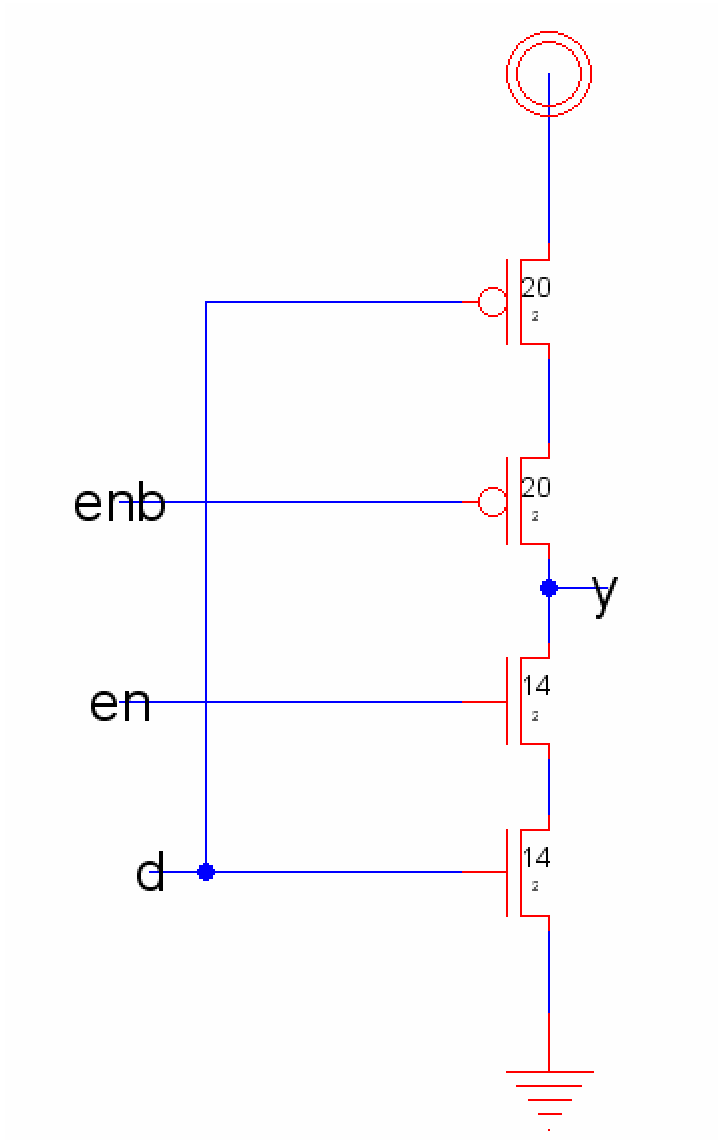
gates_9 – ALU subcell which generates the or, xor, and, and testbits functions for the alu. The output is chosen by tristates which receive their control signals from the controller.



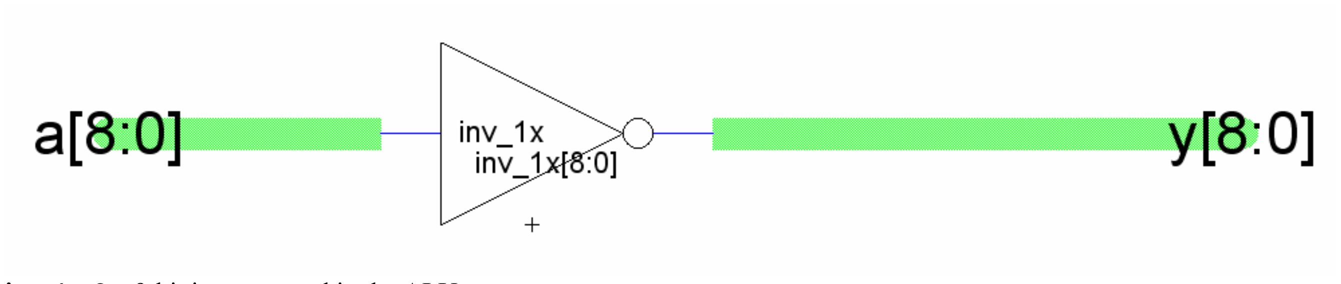
incrementer – Basically a half adder where one of the inputs is only one bit. Has the effect of incrementing a if cin is high. Used in the datapath to increment the program counter. The cout signal connects the lower and upper bytes of the program counter.



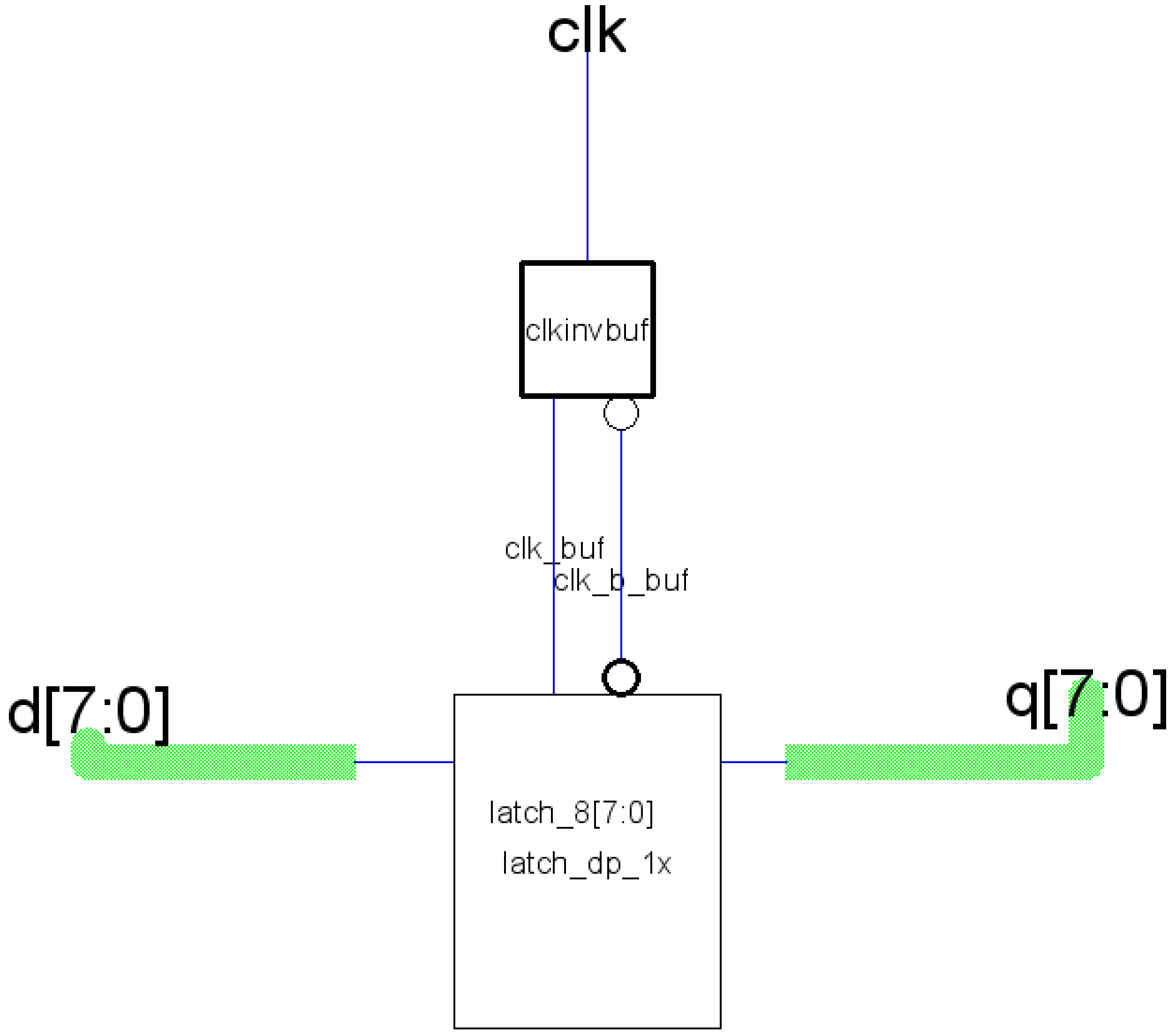
invbuf_1x – combined buffer and inverter cell.



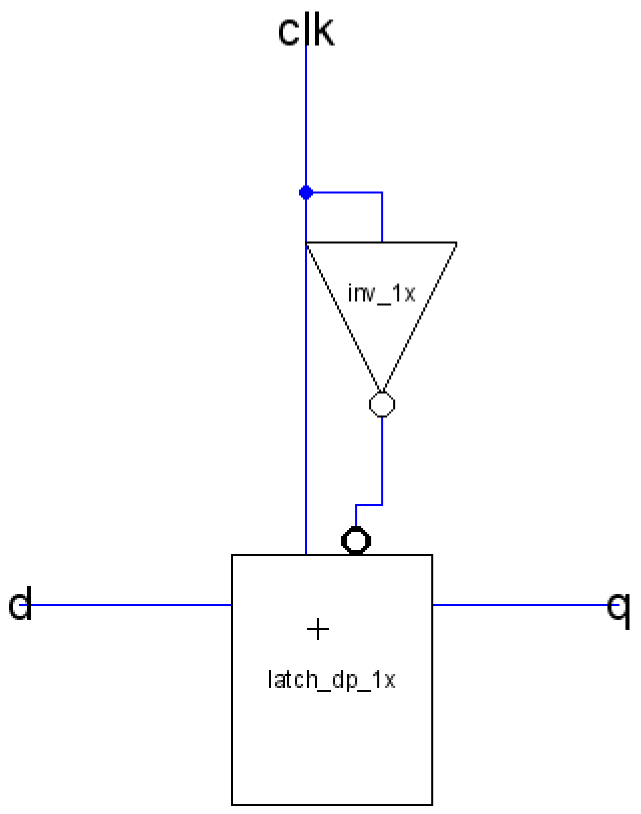
invtri_dp_1x – tristate inverter, sized for use in the datapath.



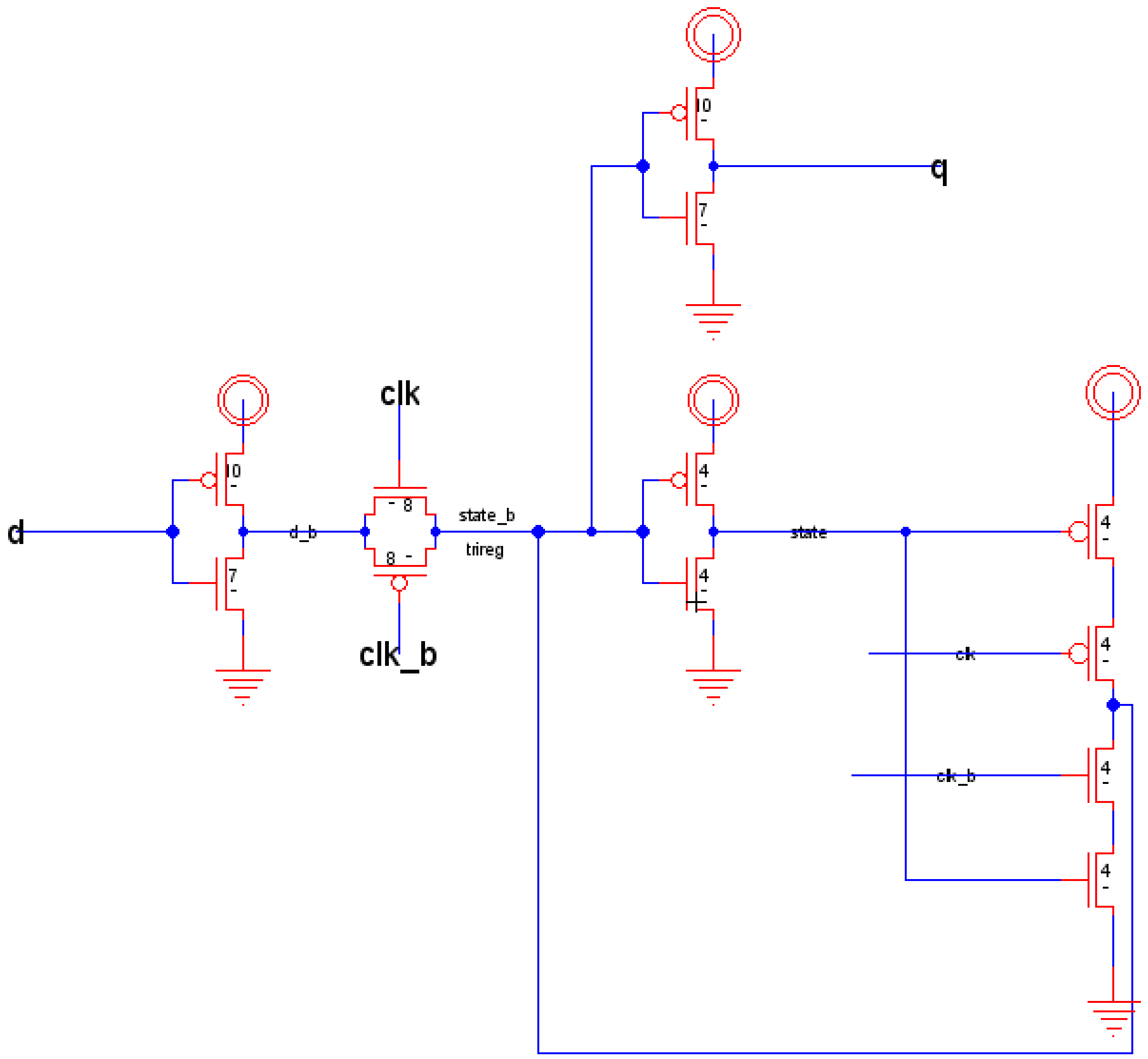
inv_1x_9 – 9-bit inverter used in the ALU.



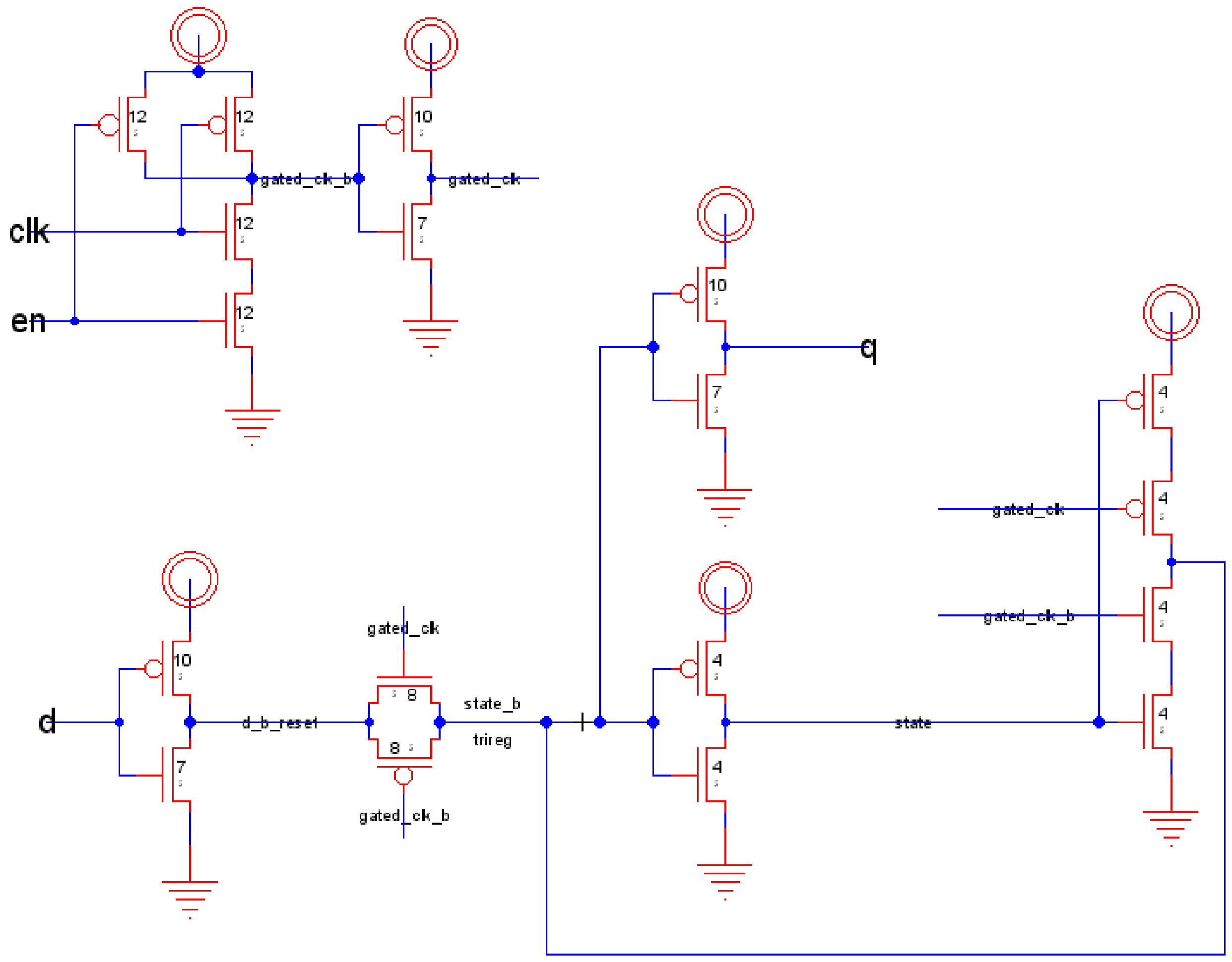
latch_1x_8 – designed as an 8-bit latch, it instantiates eight latch_dp_1x versions and utilizes a buffered and inverted clock from clkinvbuf to reduce the load on clk.



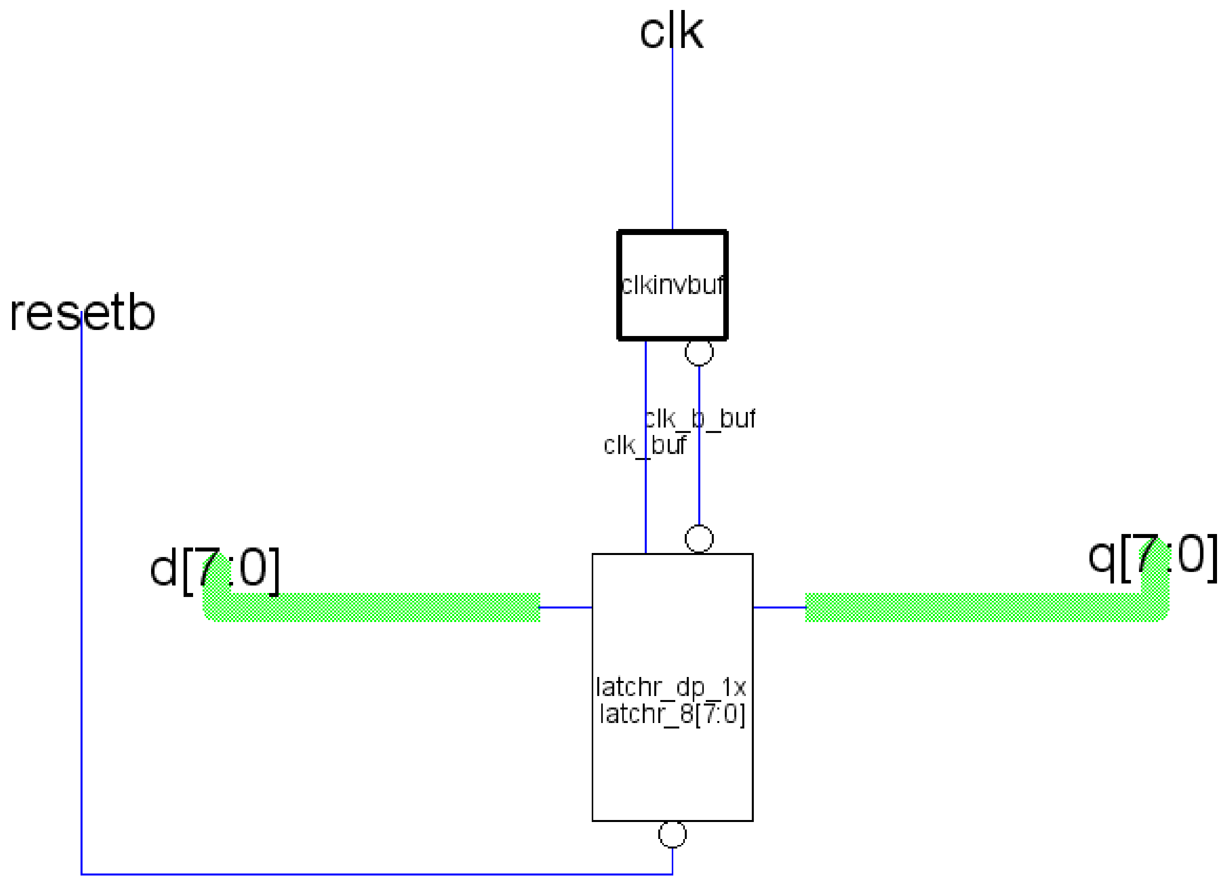
latch_c_1x – instantiates latch_dp_1x but also feeds it a generated inverted clock. Designed to make a since latch in the controller more modular.



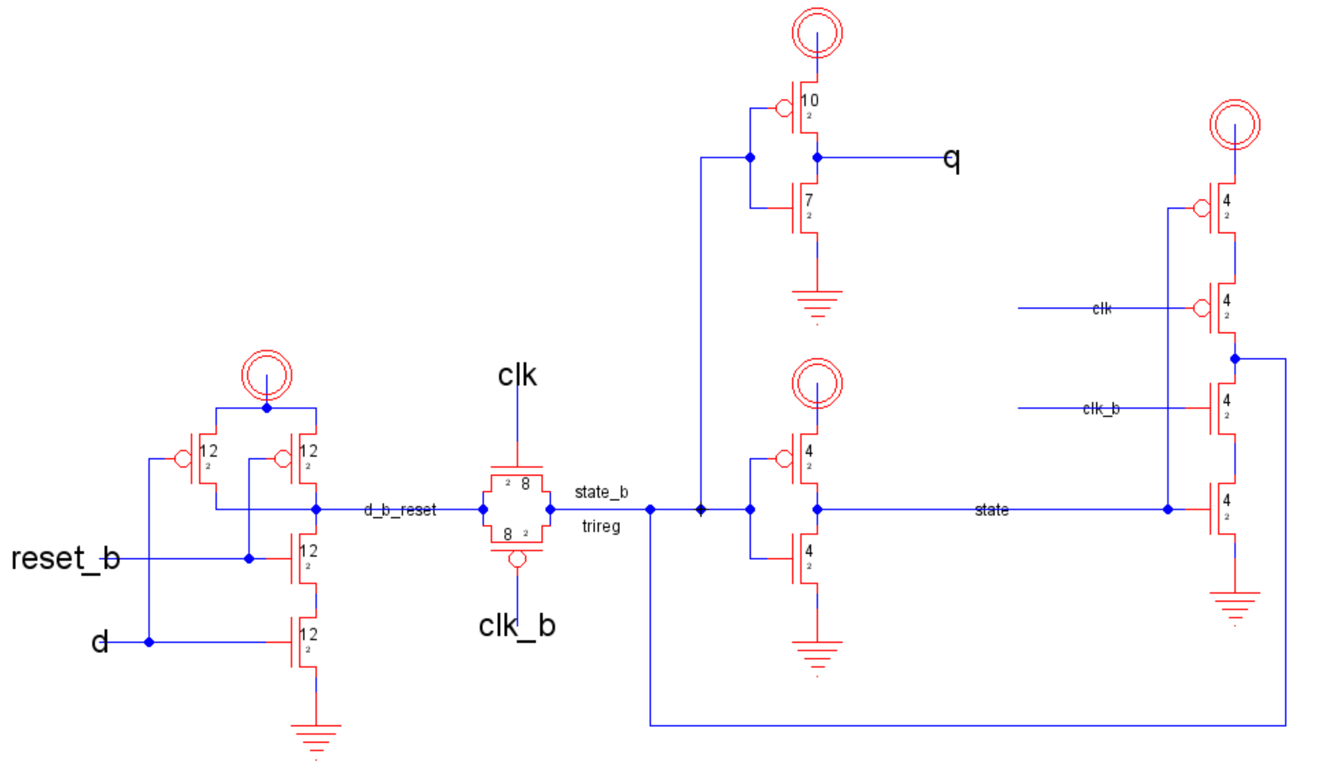
latch_dp_1x – the transistor level schematic for the latch. Utilizes synchronous reset and an input and output inverter to stabilize the signals and prevent back driving.



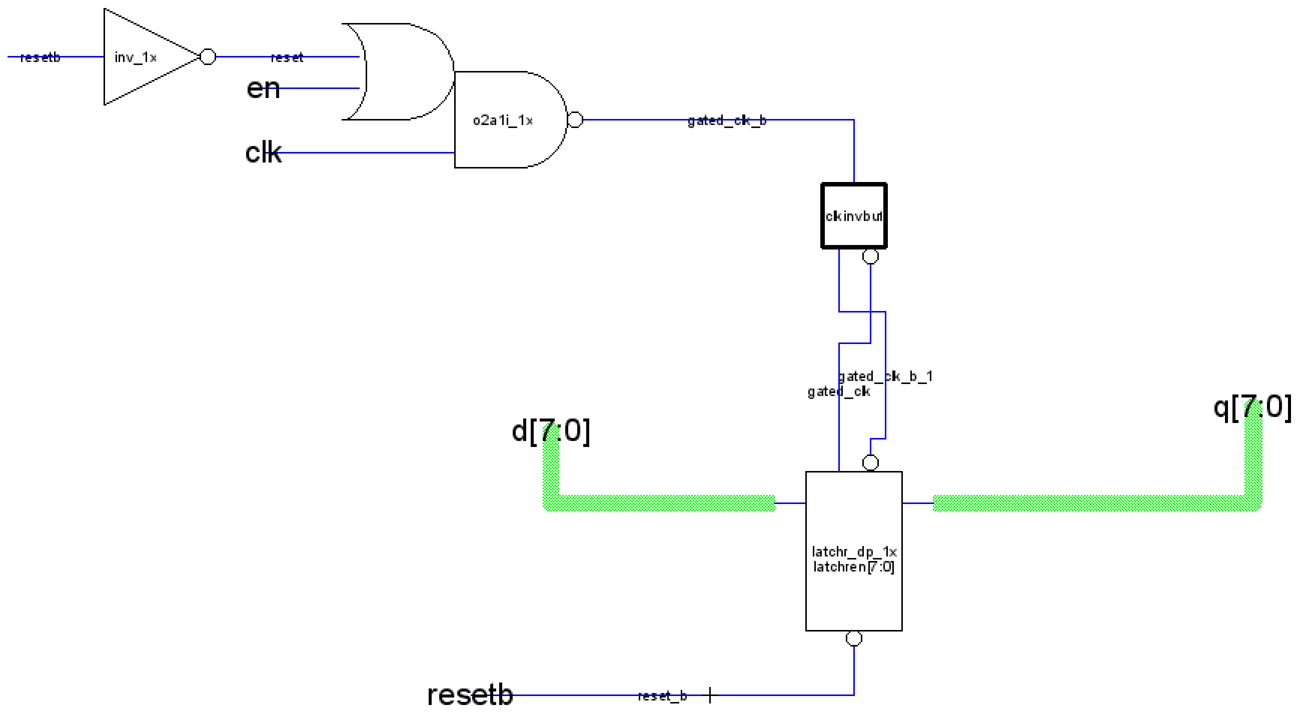
latchen_c_1x – and enabled latch. Since this is the controller version, it generates its own gated_clk signal.



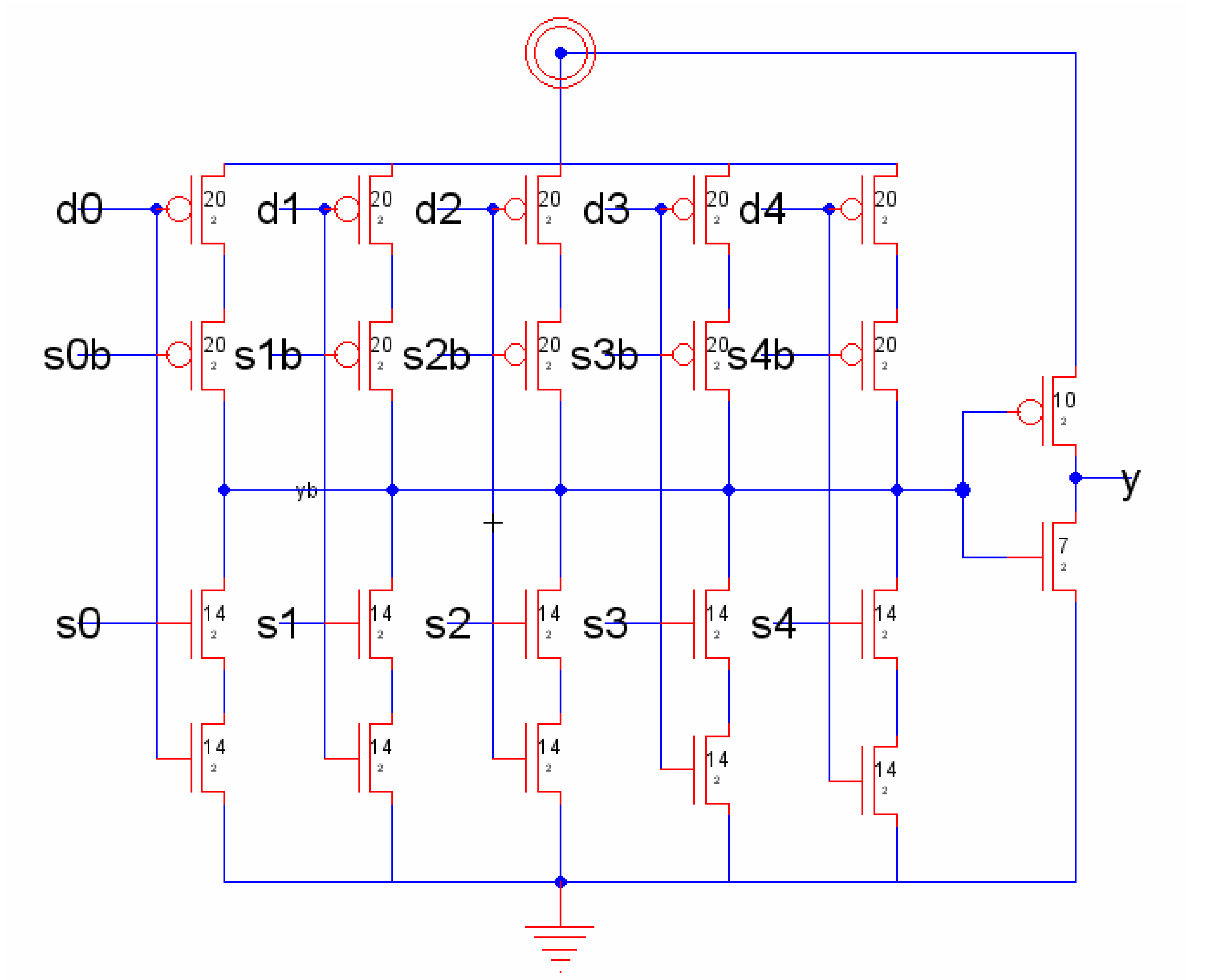
latchr_1x_8 – an 8-bit latch with reset. Takes in resetb since the chip is given resetb.



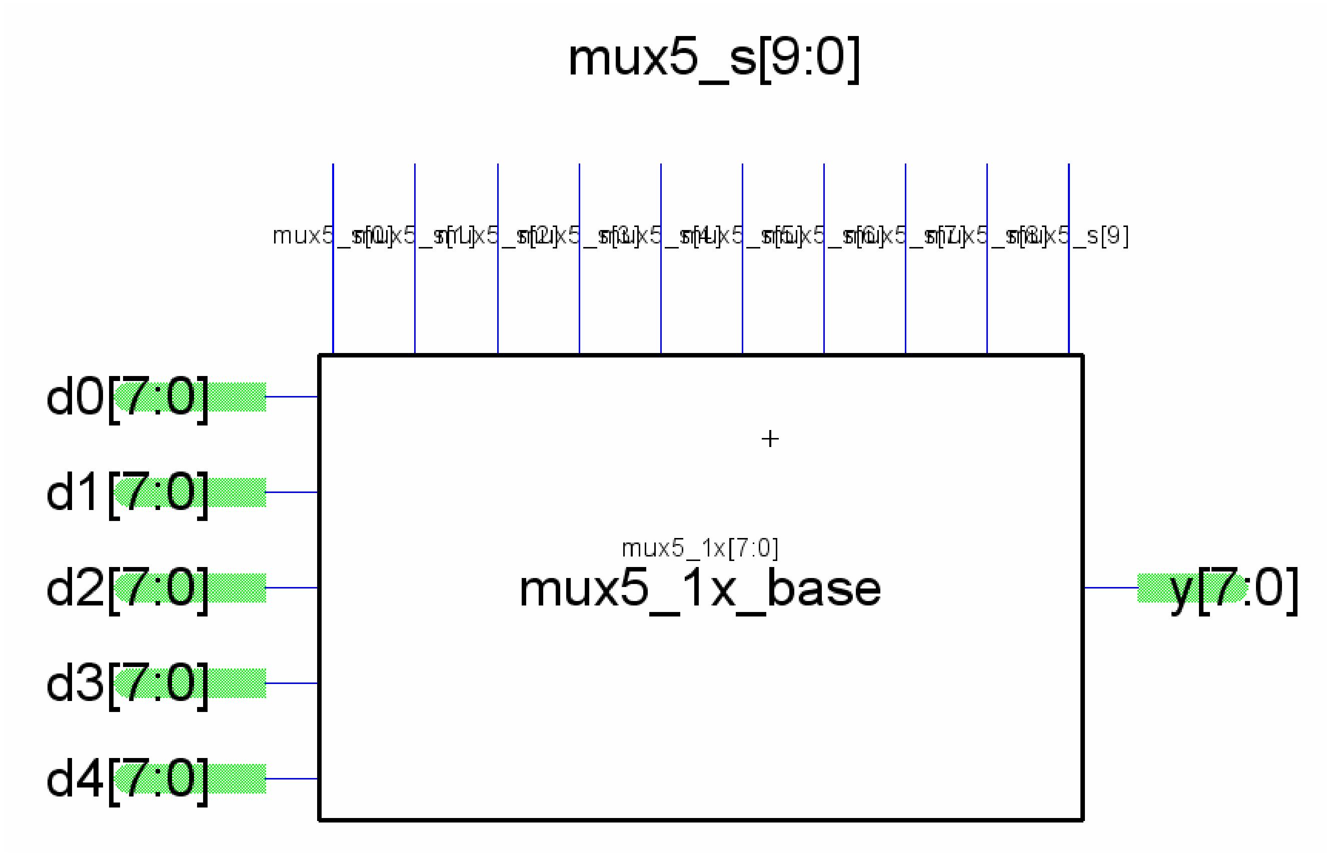
latchr_dp_1x – the resettable latch. Since this is a synchronous reset, it only resets when reset_b is high and clk is high.



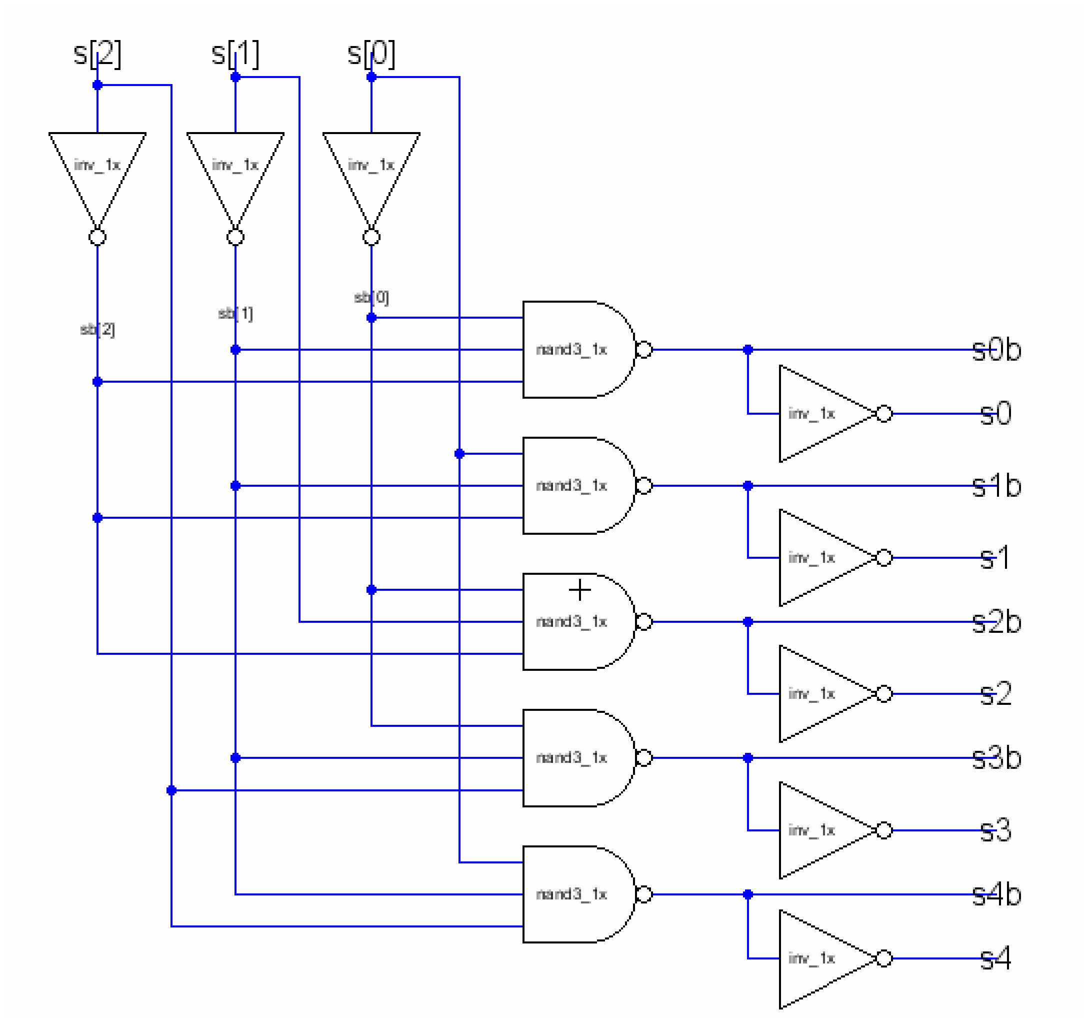
latchren_1x_8 – a resettable enabled 8-bit latch. Uses gated clock and a resettable latch set. The or-and gate ensures that even if enable is low, reset can affect the gated clock.



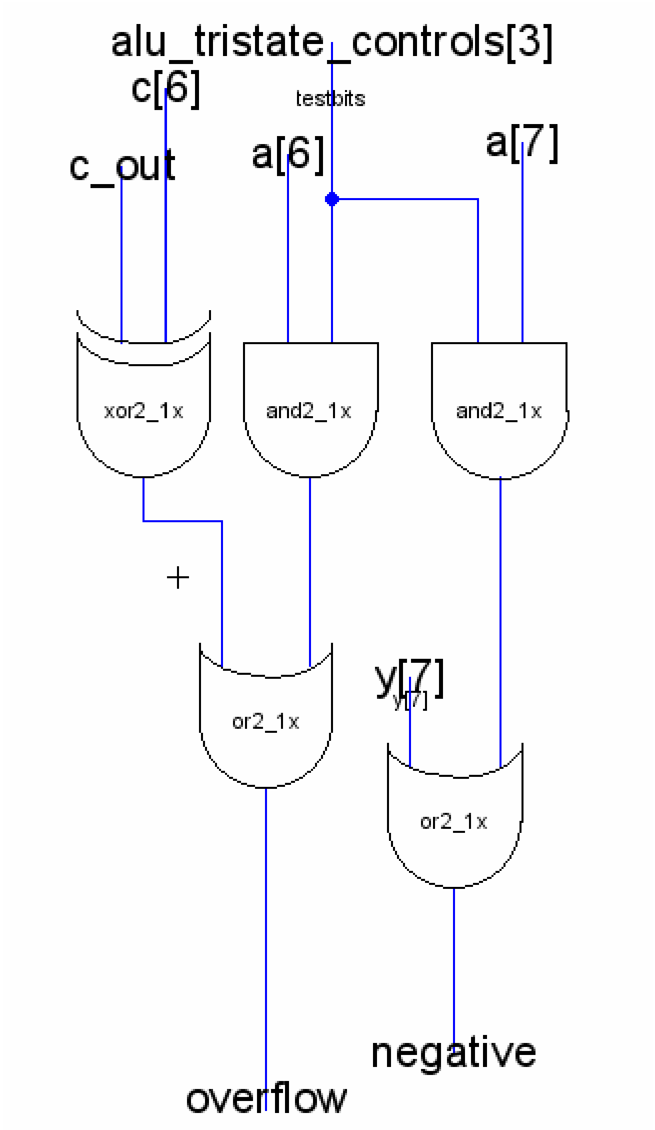
mux5_1x_base – 1-bit, 5 input mux. Uses one-hot encoding, and must be provided normal and inverted control signals. The mux5_decoder block provides the control signals. This is to save transistors on the 8-bit version of the mux, since only one encoder block is needed regardless of the number of bits.



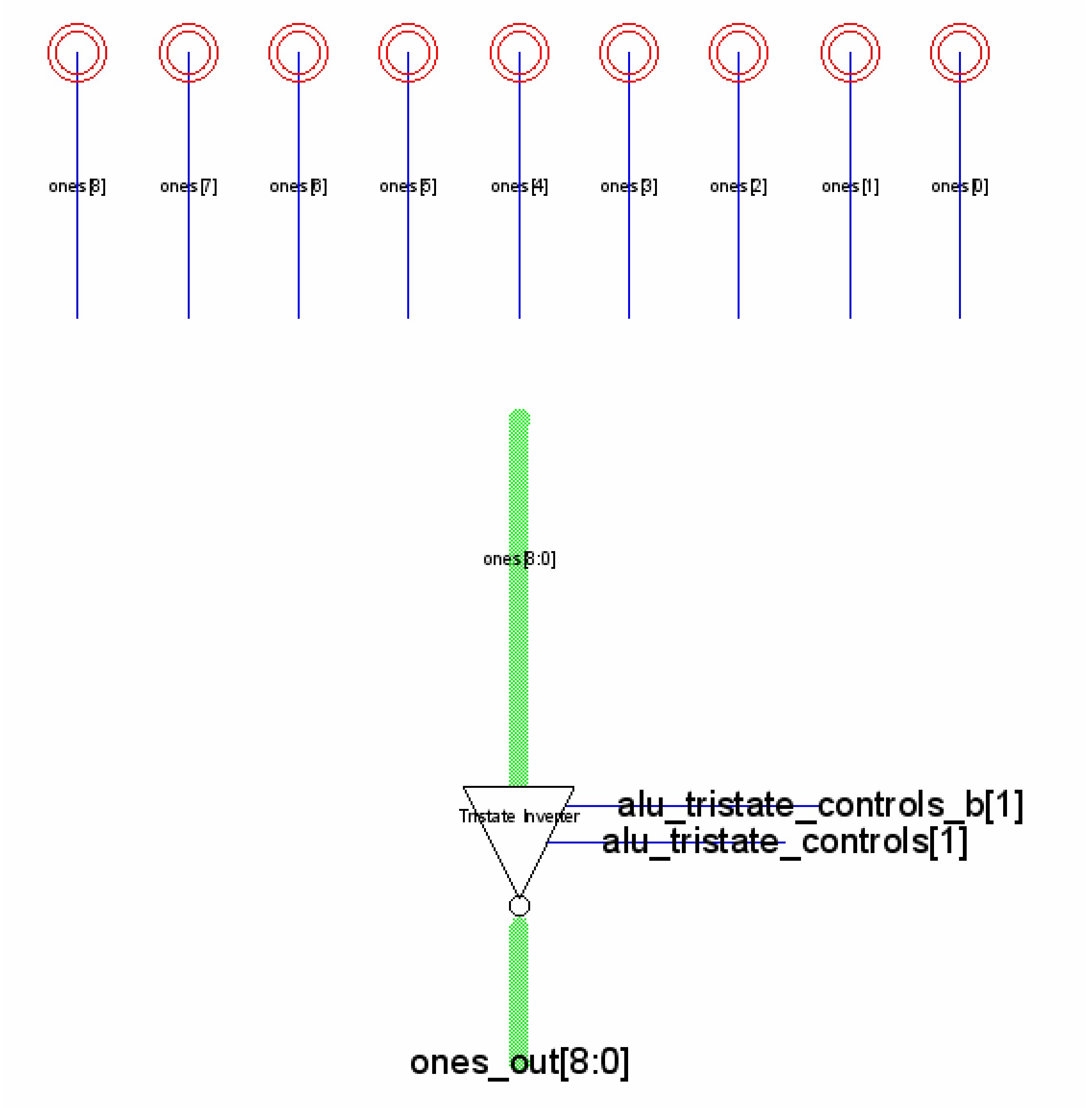
mux5_1x_8 – 8-bit version of the mux5_1_base. The same control signals can go to all 8 copies, thus saving gates.



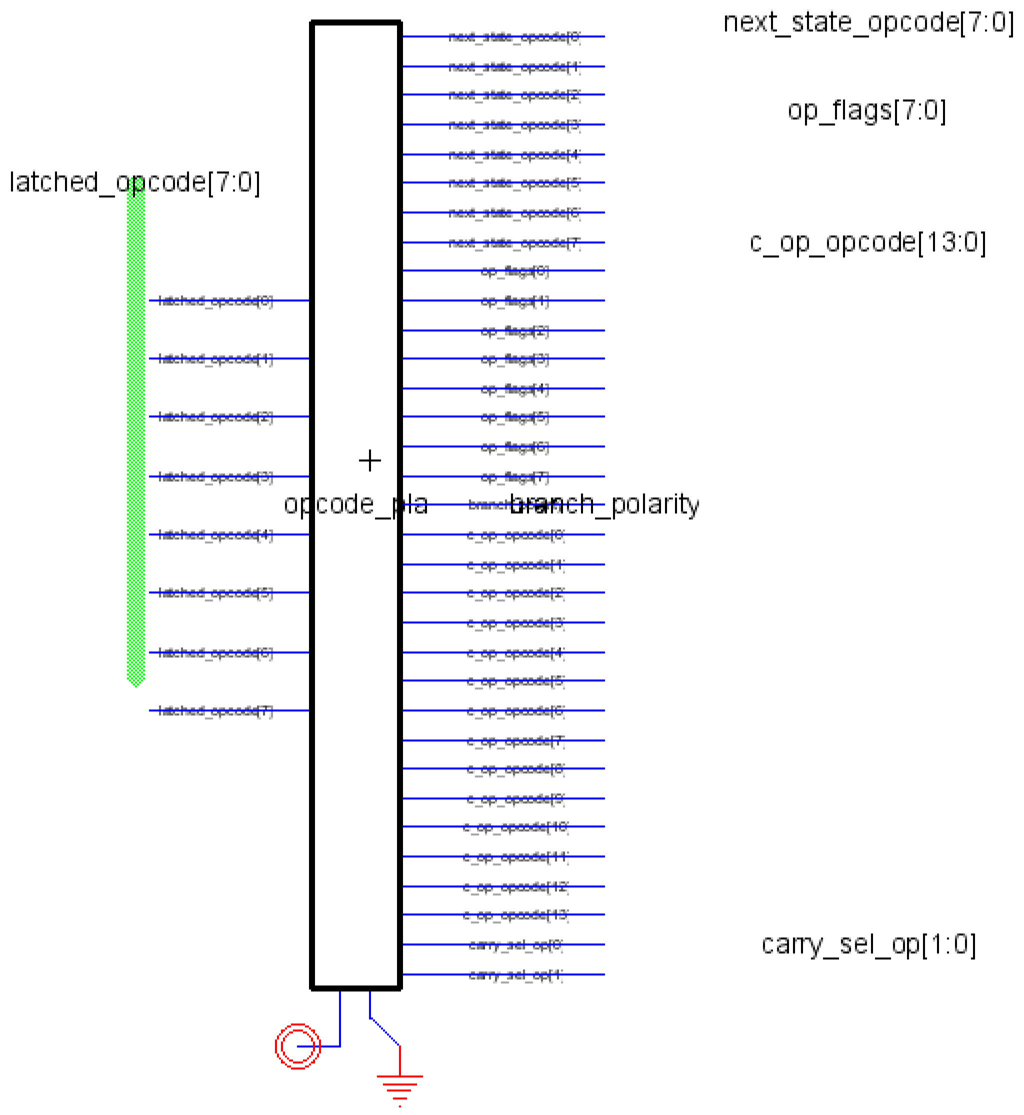
mux5_decoder – Takes the standard mux select signals and converts them to a 4-bit one-hot signal, with inverted values. This allows the mux5_1x_base schematic to be simpler. This cell is part of the controller, while the mux5 base is in the datapath



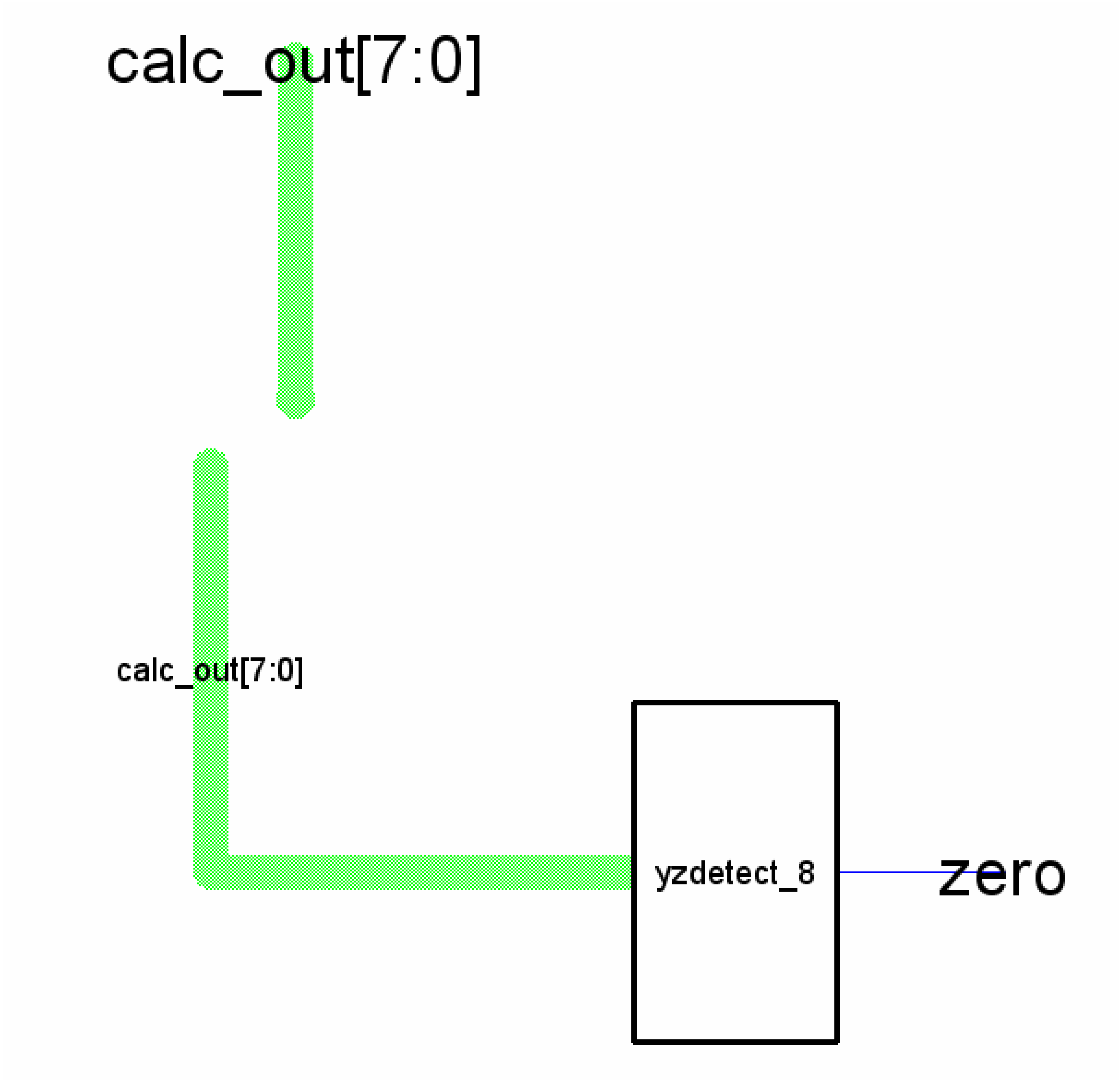
negative_overflow_gen – generates the negative and overflow outputs of the ALU, is in the zipper of the ALU.



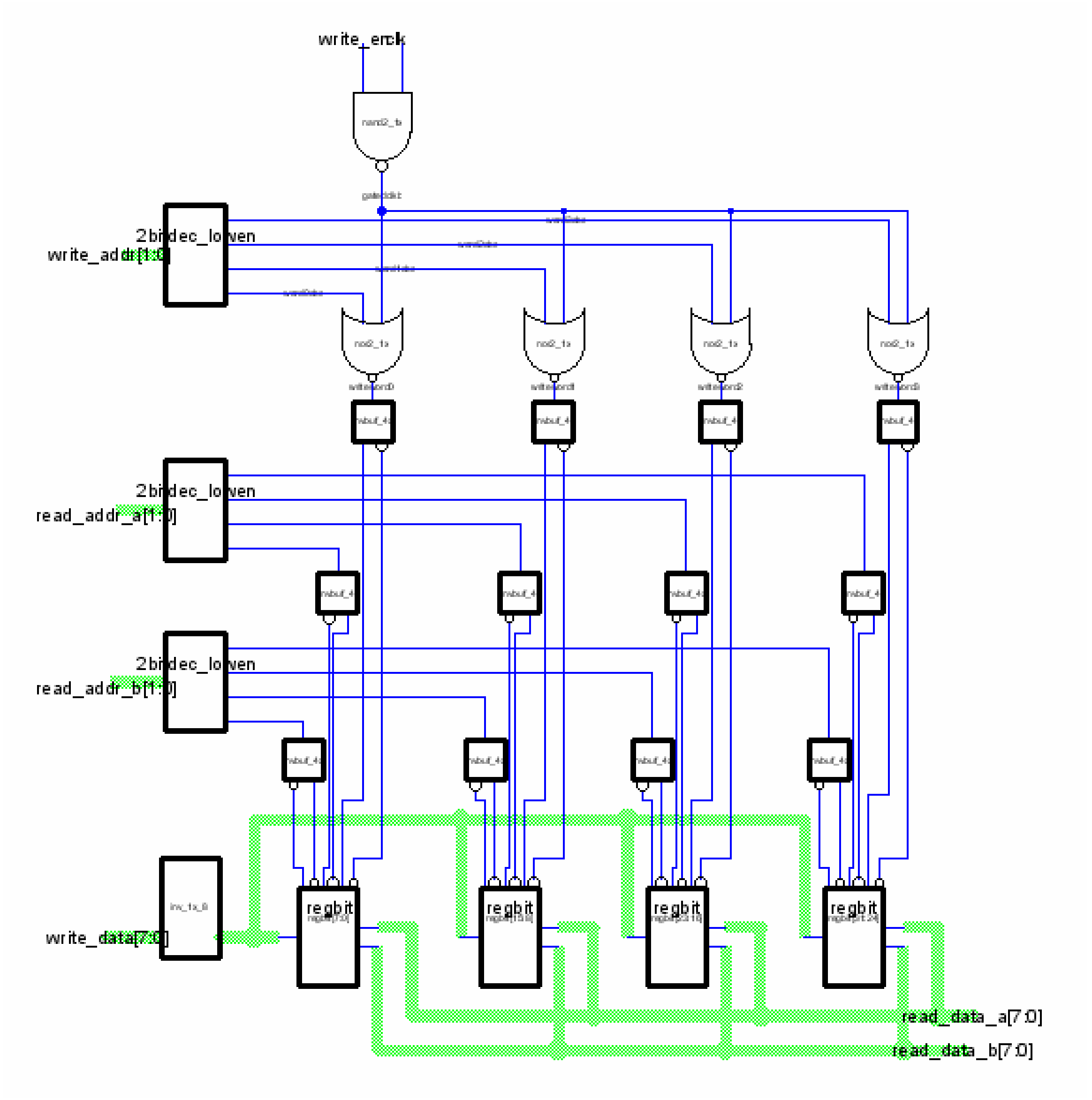
ones – ALU subcell. Generates nine high bits.



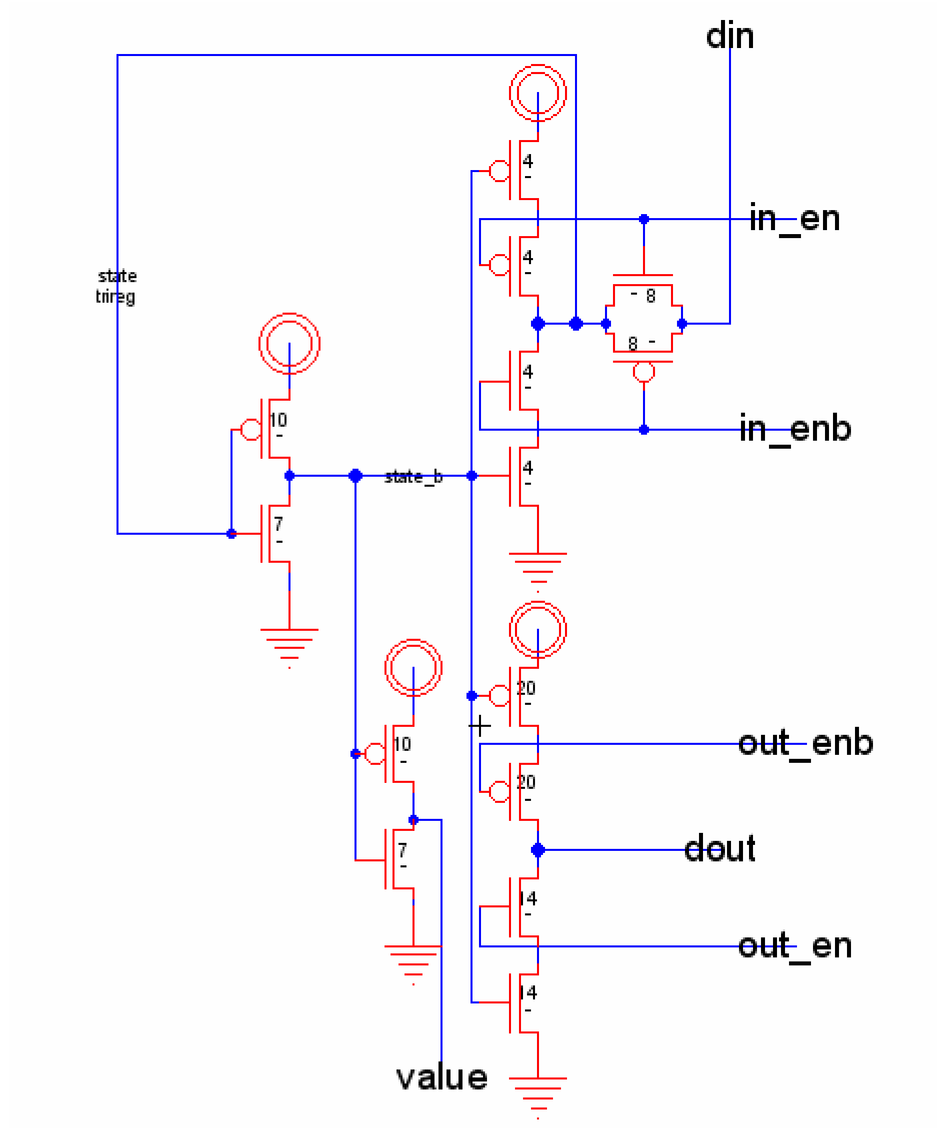
opcodePLA_bused – the sole purpose of this cell is to group the input and output signals of the opcode ROM into more manageable bus lines and naming.



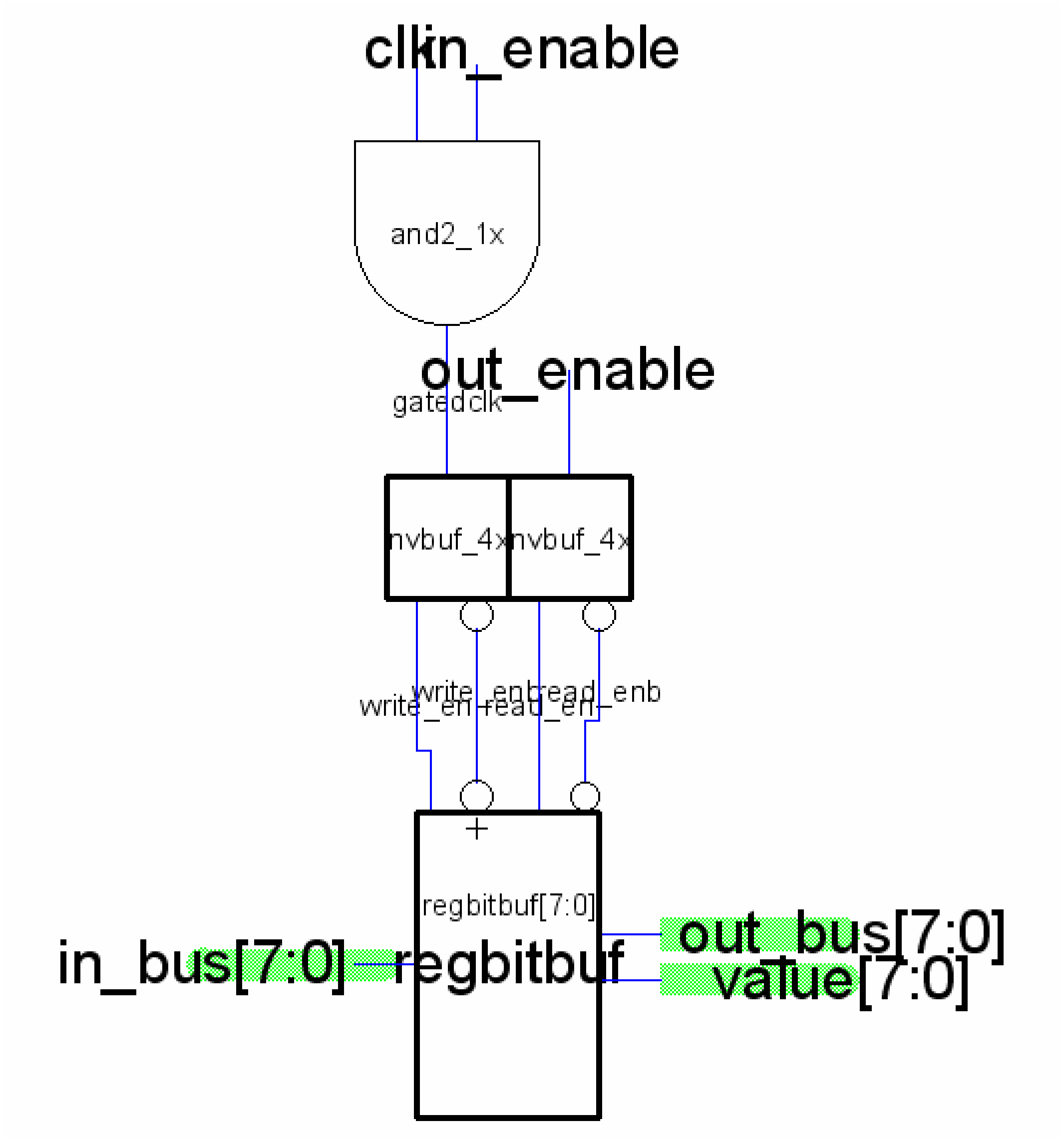
post_calc – generates the zero output of the alu (high if output is 8 bits of 0)



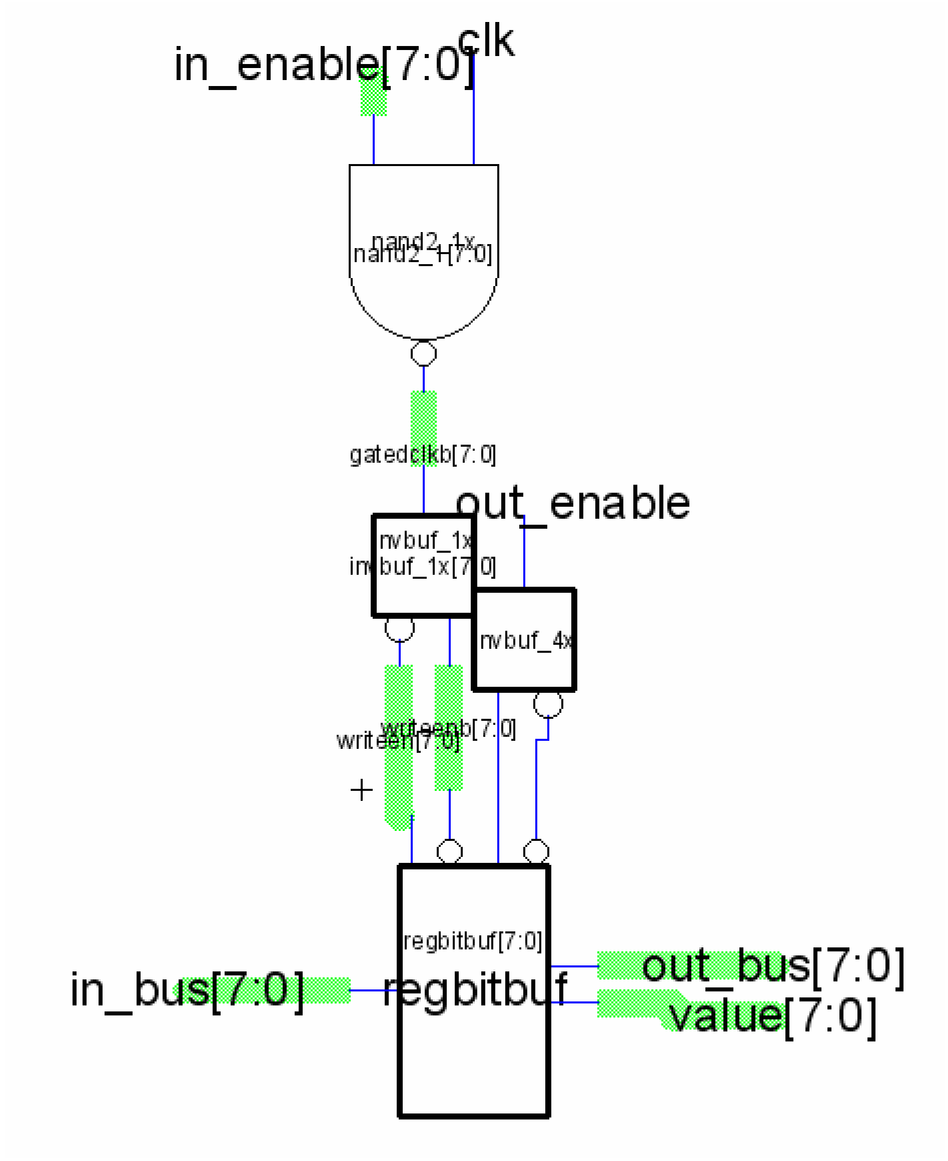
regfile_8 – 8-bit register file. Data to the a and b lines is controlled by tristates in the dp_regfile cell.



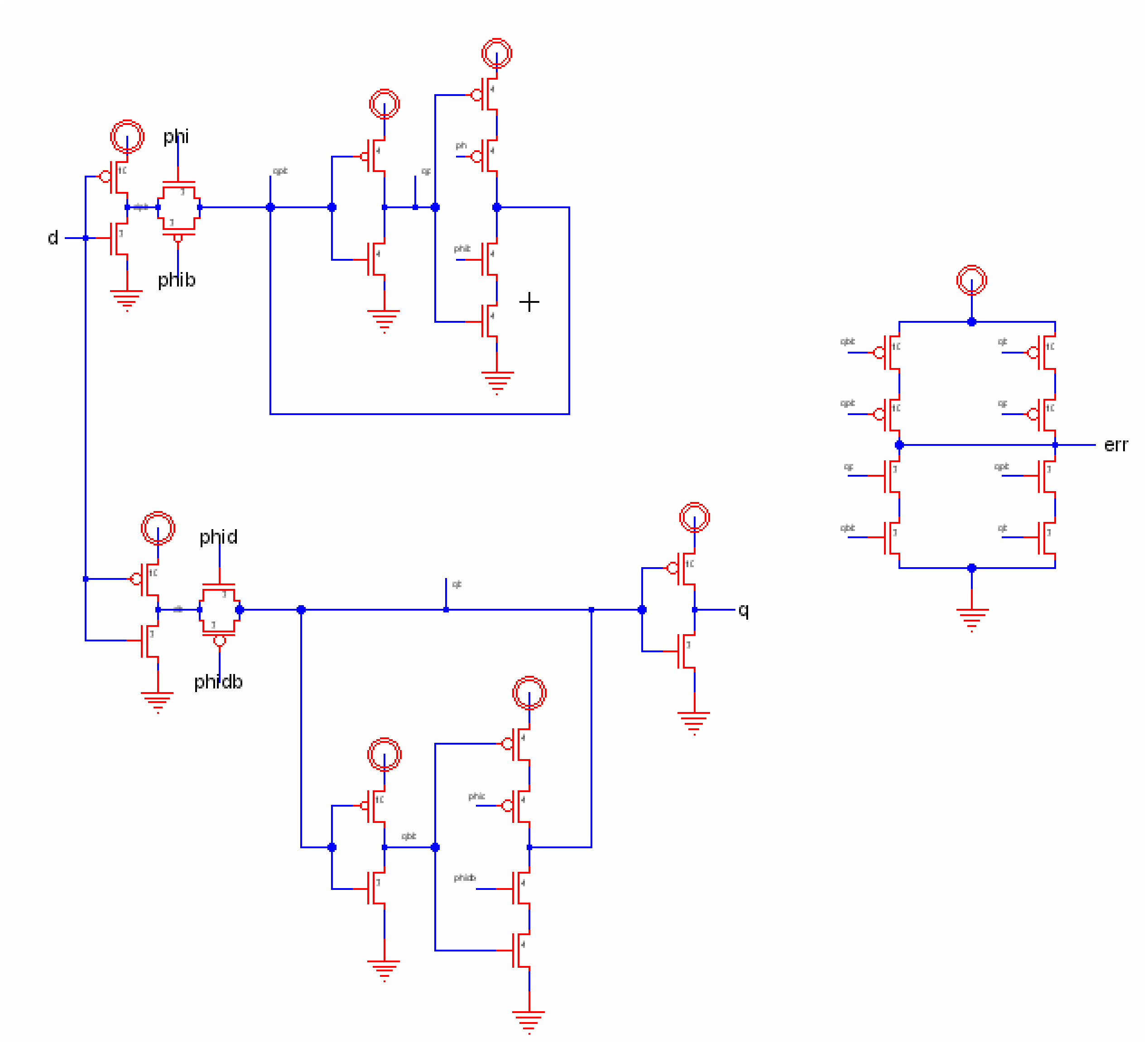
regbitbuf – buffered 1-bit register cell, used in the non-register file registers.



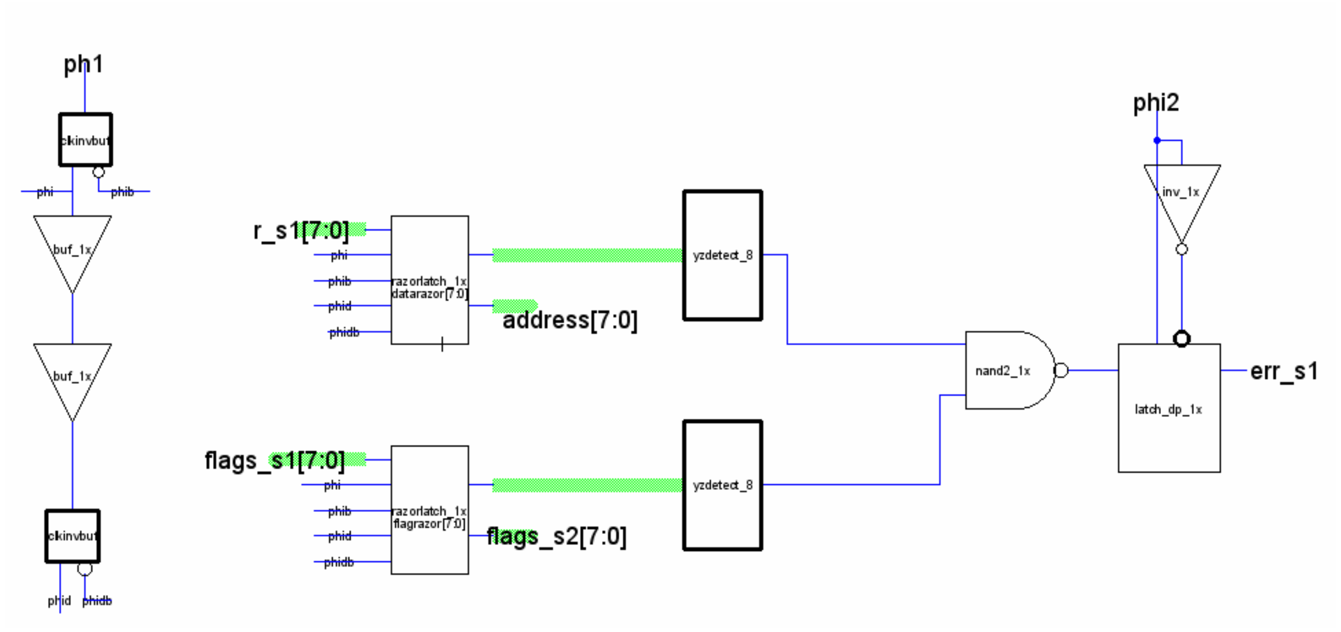
registerbuf_8 – 8-bit buffered register. Gated clock is used to save power.



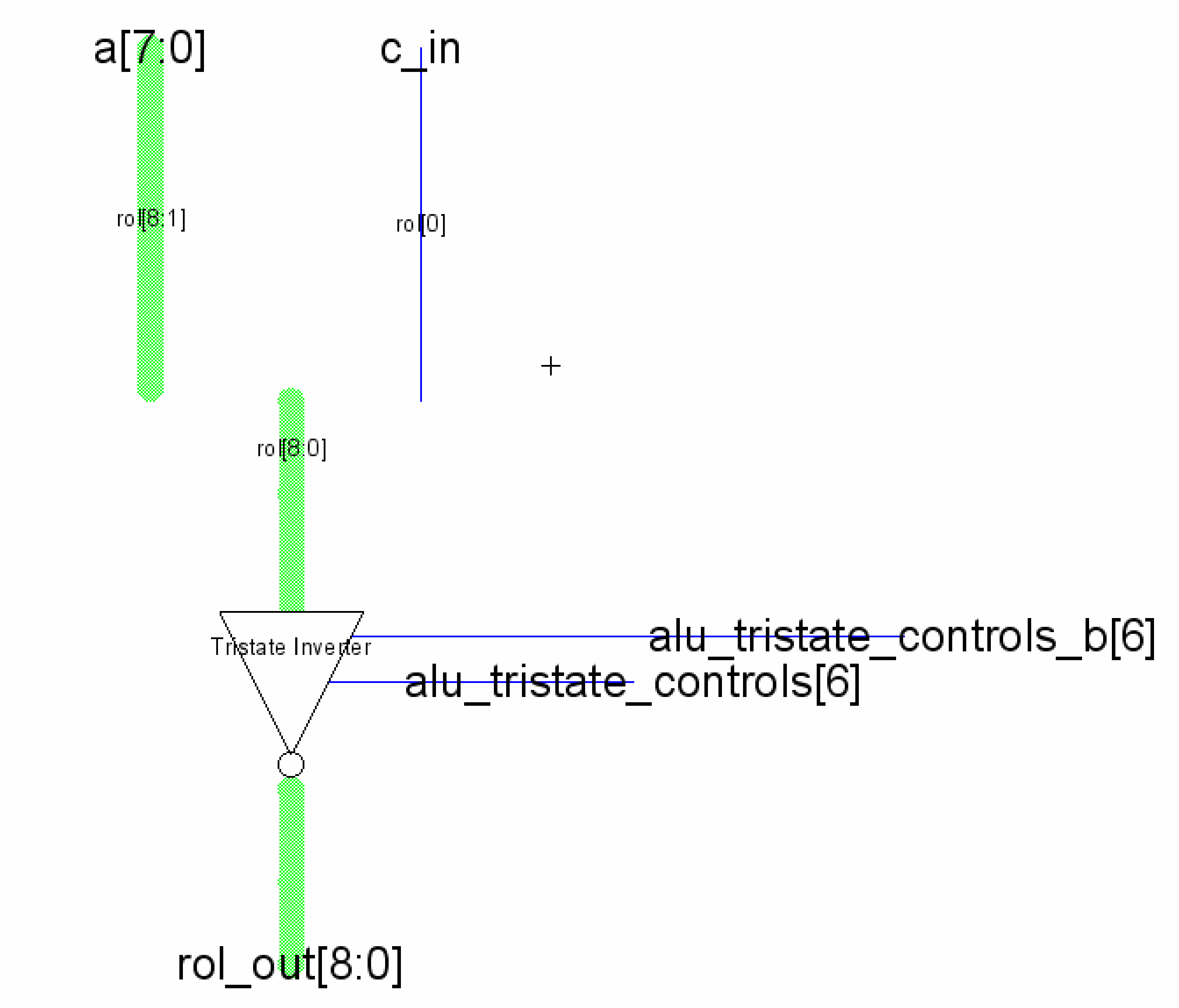
registerbufmasked_8 – masked version of the 8-bit buffered register cell.



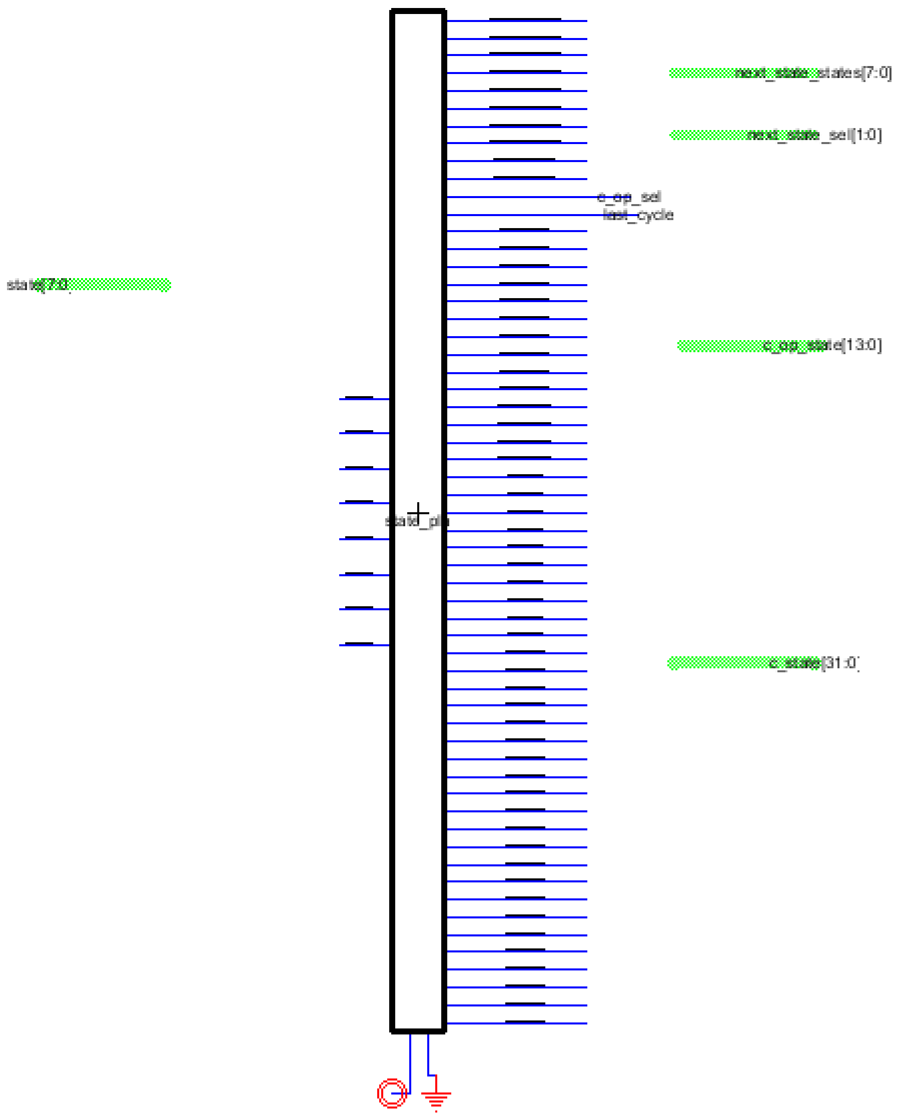
razorlatch_1x – Experimental latch that outputs an error signal when at the edge of failure. This is done by delaying the clock to a second latch, and comparing the outputs of the two latches to see if they are different. If they are, then that slight delay is enough to cause the latch to fail, meaning the chip is just on the edge of failure.



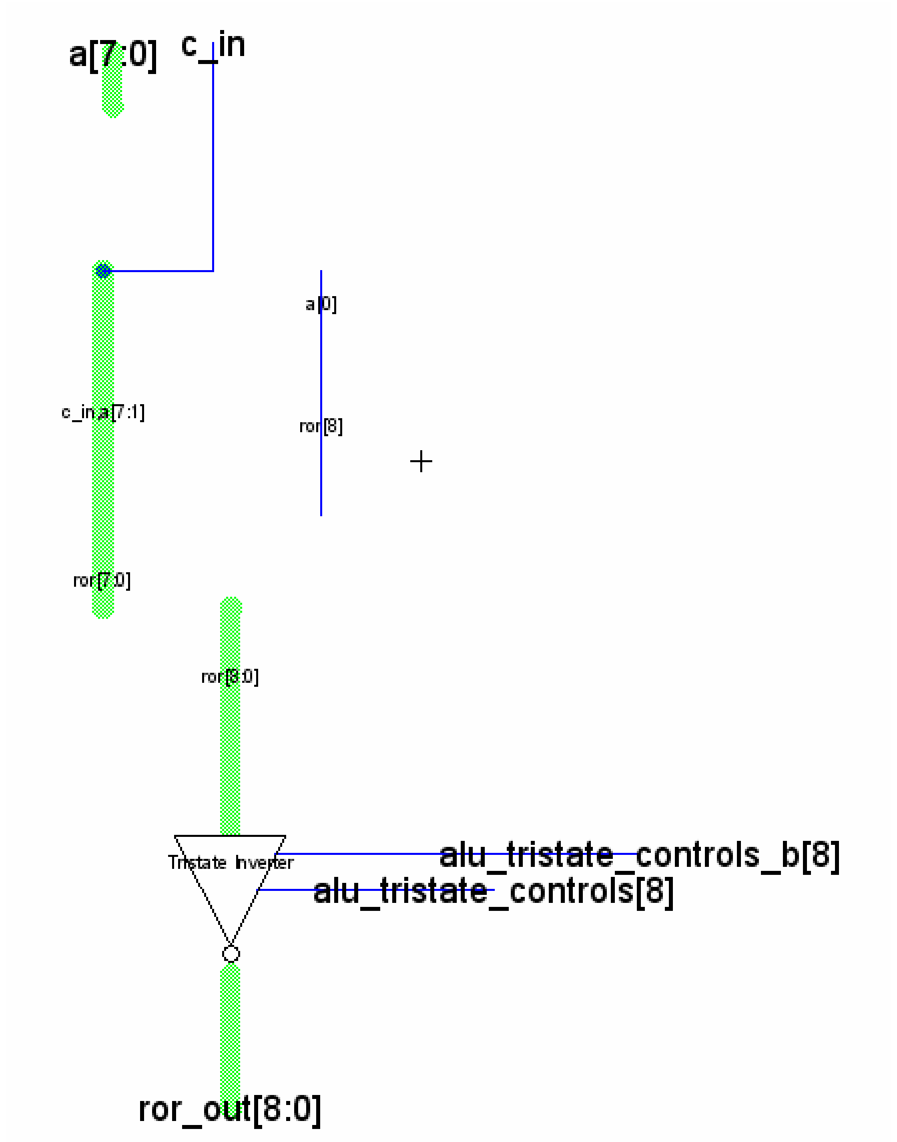
razorslice – Combines two razor latches to create one error signal which indicates if either is failing.



rol – ALU subcell that performs the left rotate function.

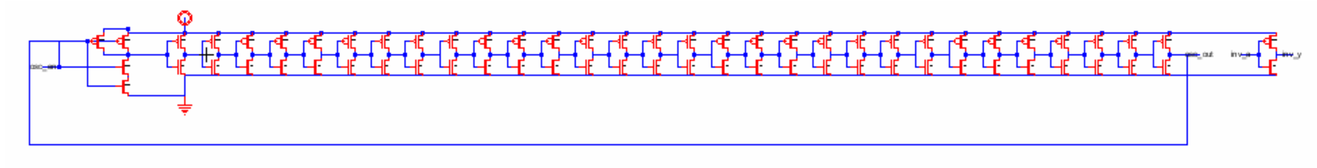


Rom_bused – the sole purpose of this is to group the signals in and out of the ROM into more manageable bus lines for the schematic.

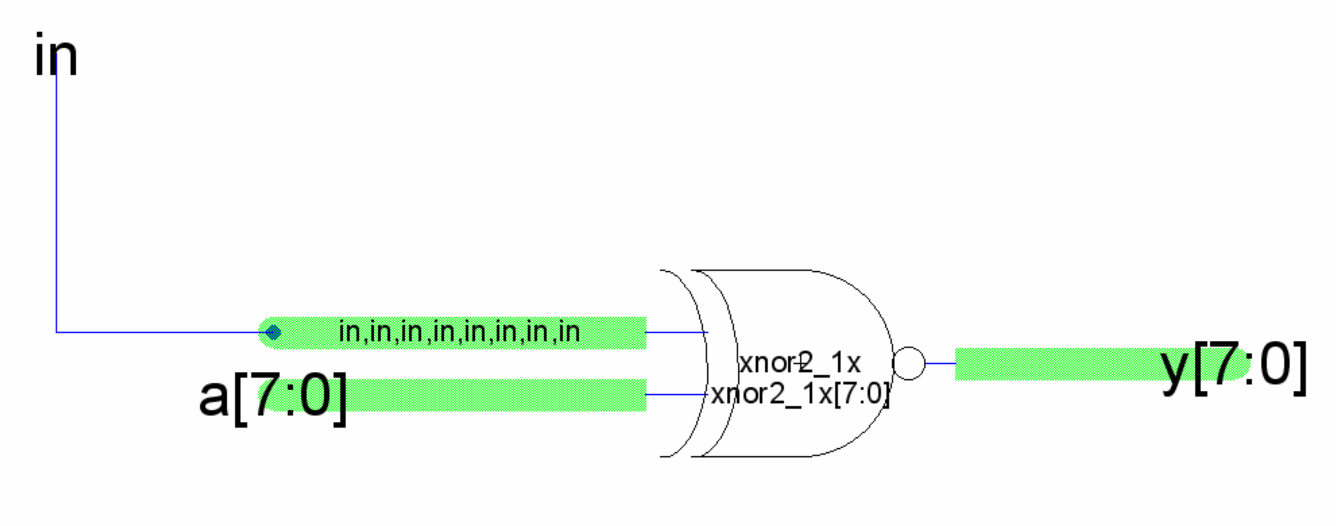


ror – ALU subcell that performs the right rotate function.

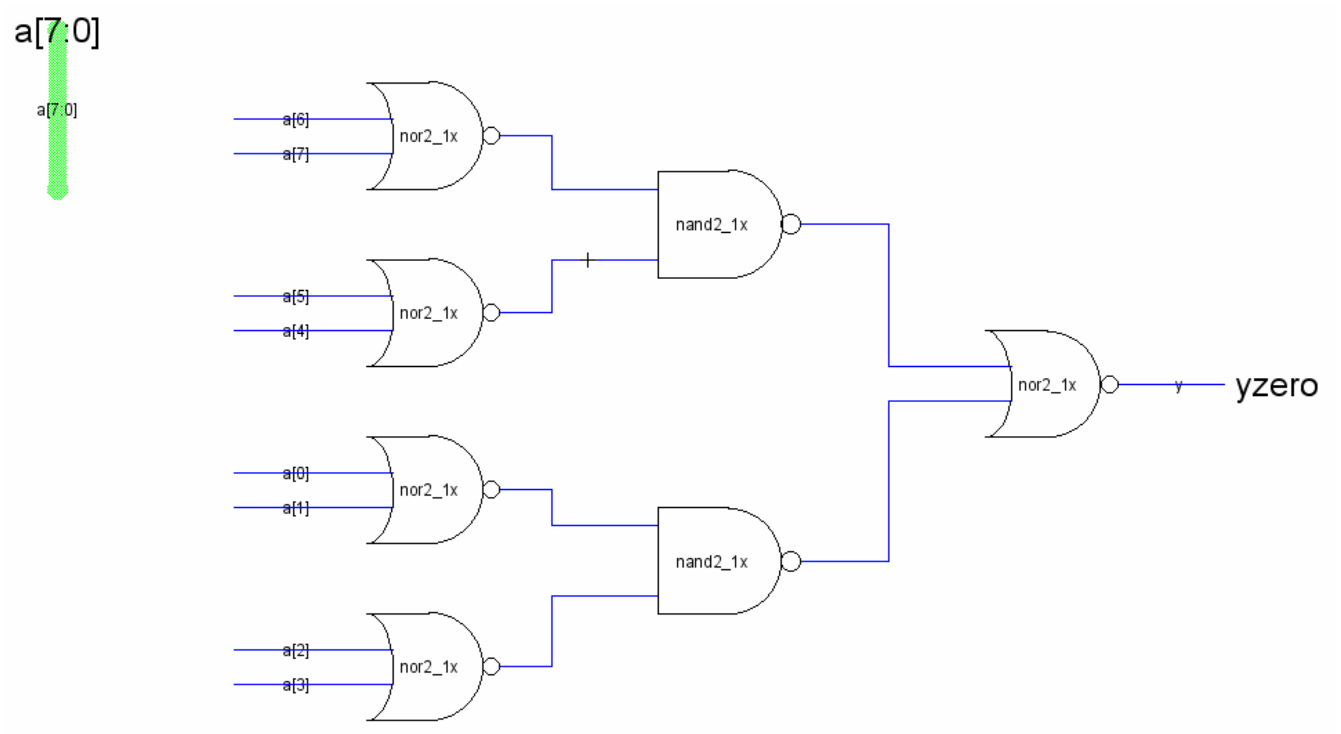
.



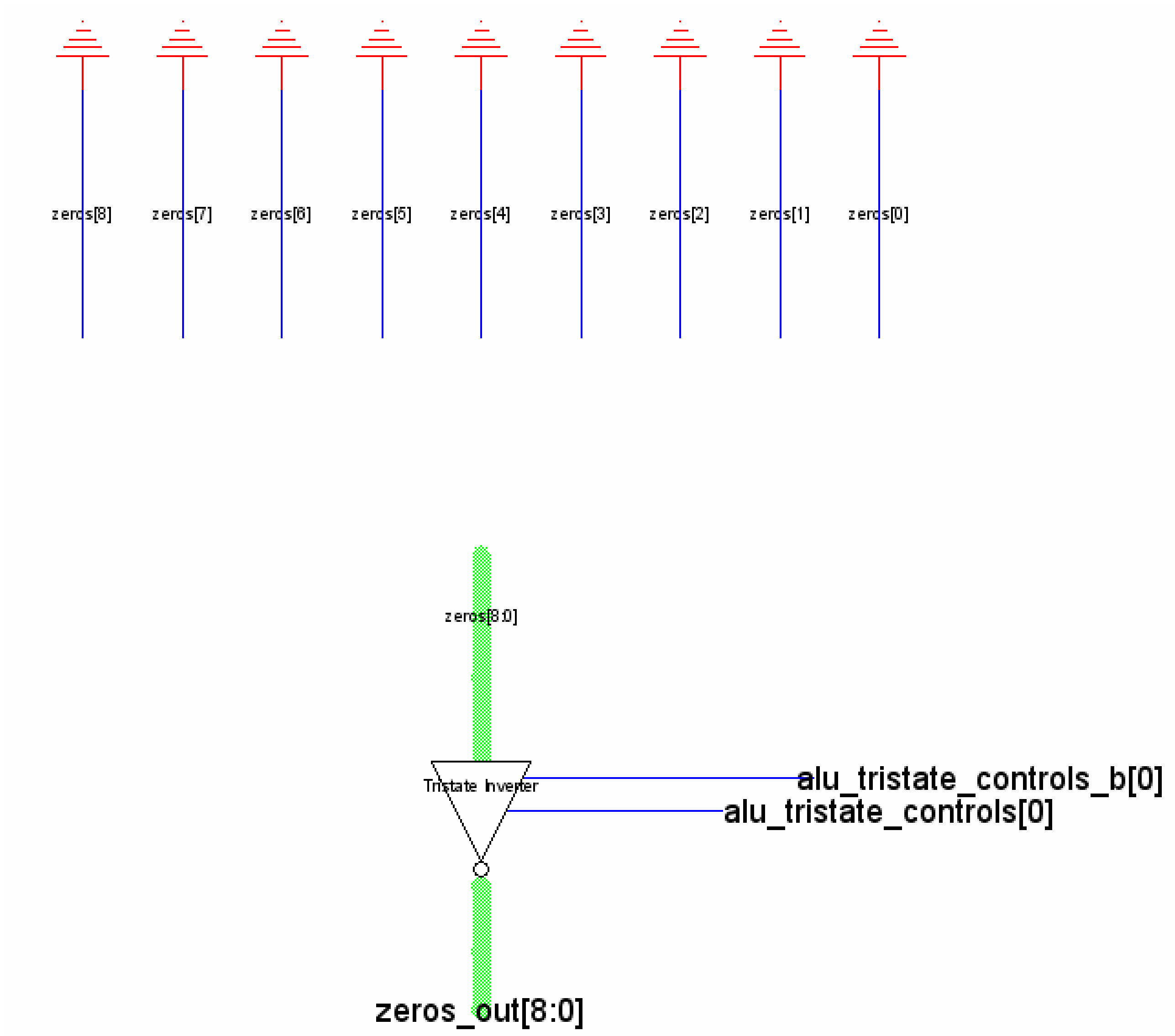
test_structure – an inverter and ring oscillator used to test basic chip performance, in case of problems with the manufactured chip. Due to pin constraints, the inverter is not used.



x_nor_constant – specialized XNOR gate used in the ALU.

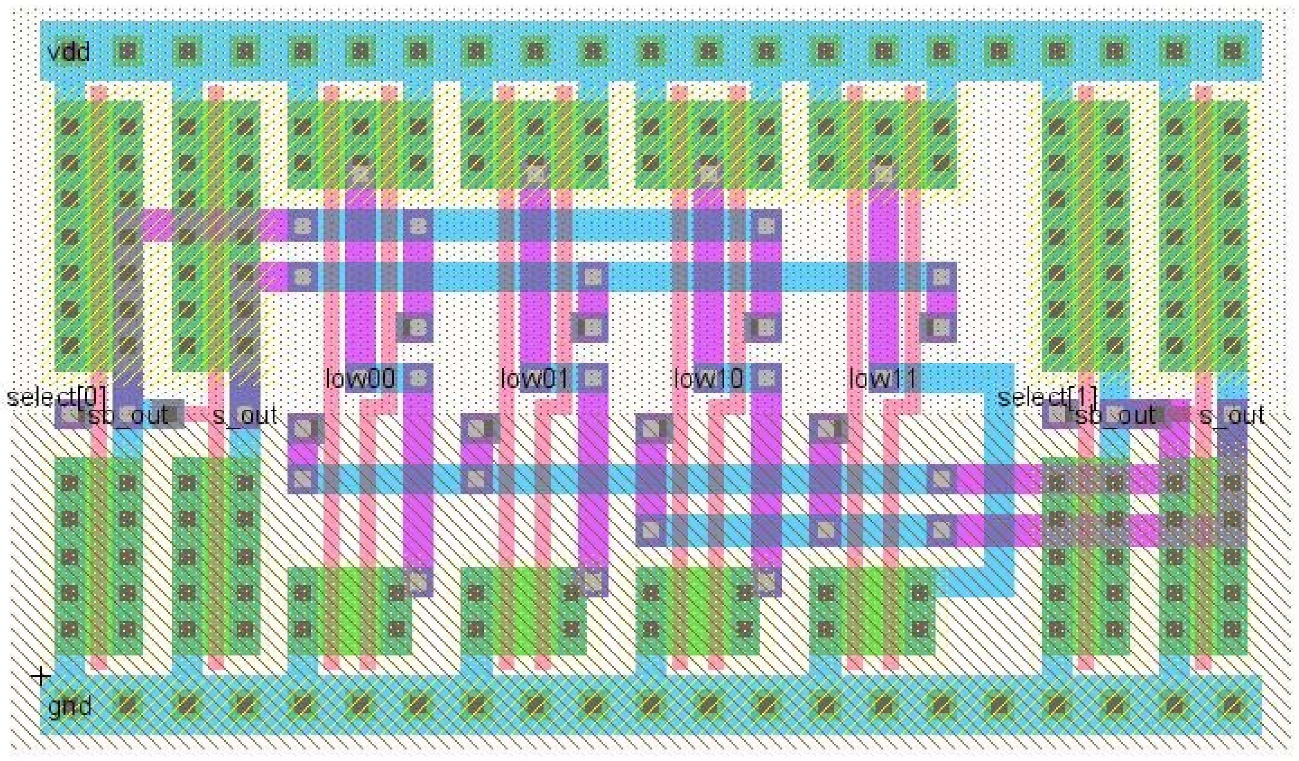


yzdetect_8 – zero detector. Output high if input bits are all low, is laid out such that it fits into a single wordslice

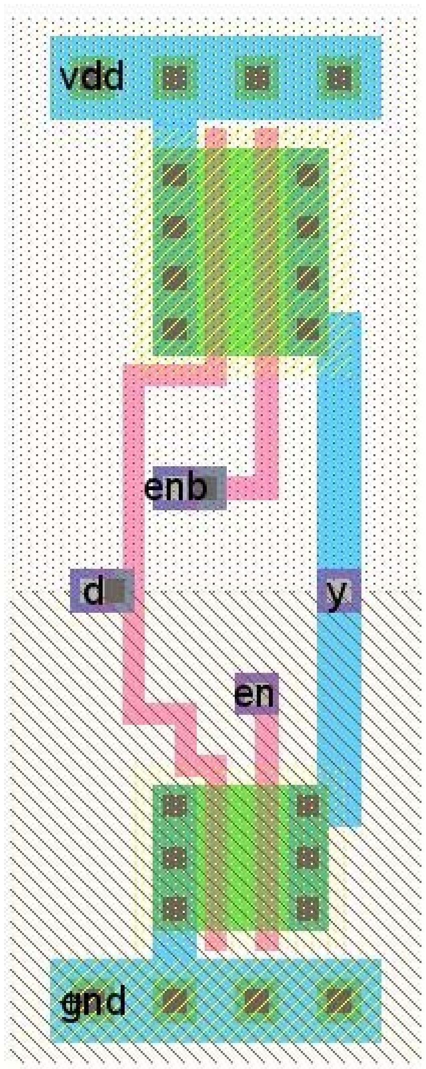


zeros – ALU subcell. Generates nine low bits.

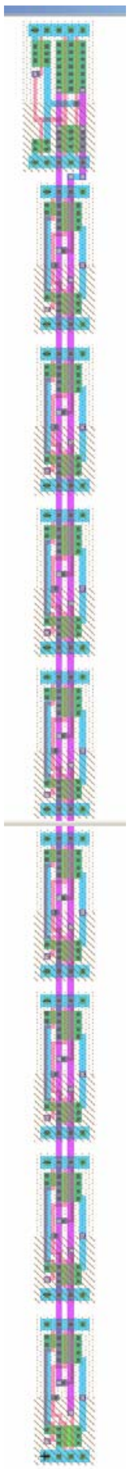
Appendix: Layout



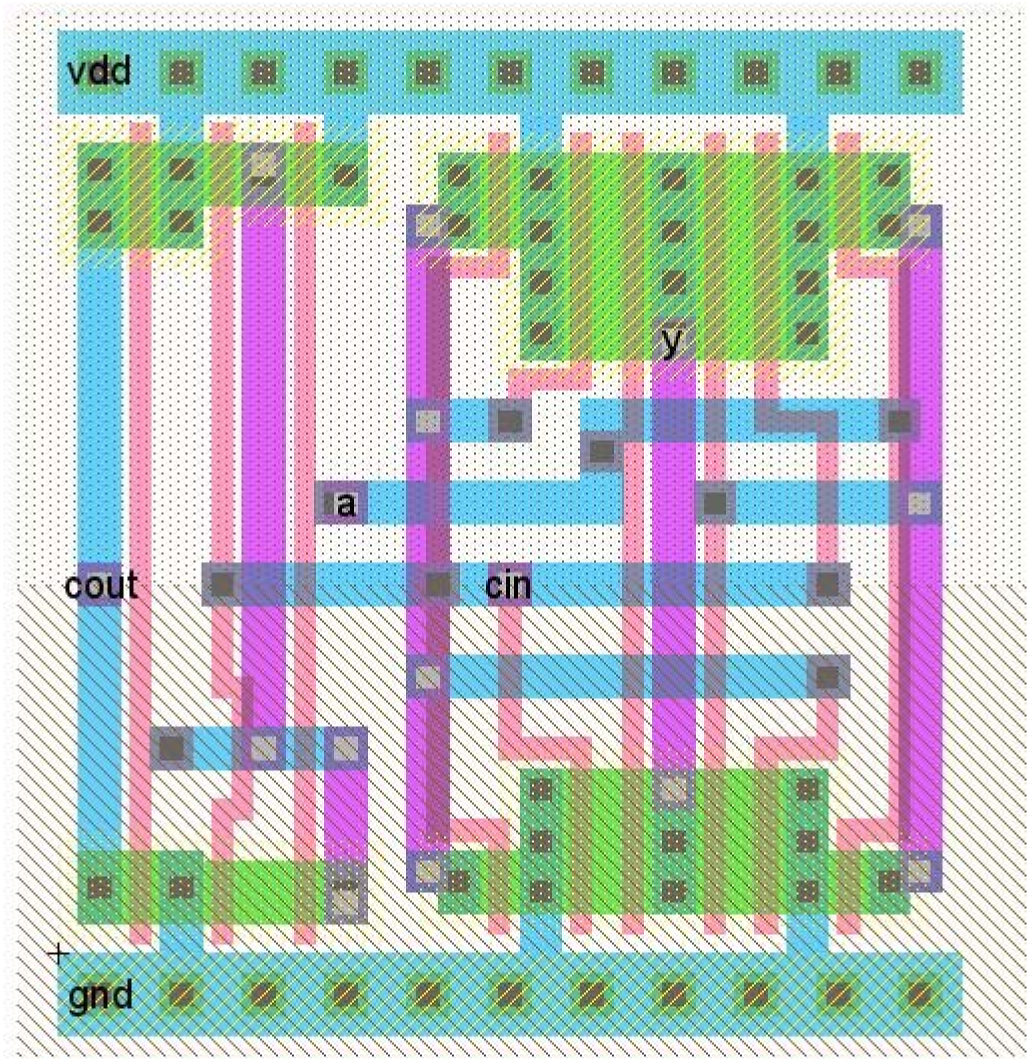
2bitdec_lowen - 2-bit decoder used in regfile_8



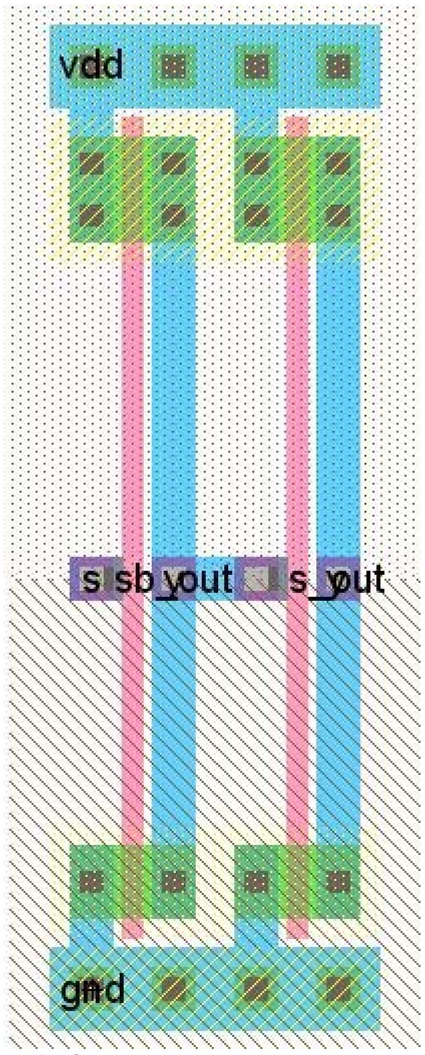
buftri_dp_1x_noinv - non-inverting tristate buffer used in buftri_1x_8_noinv



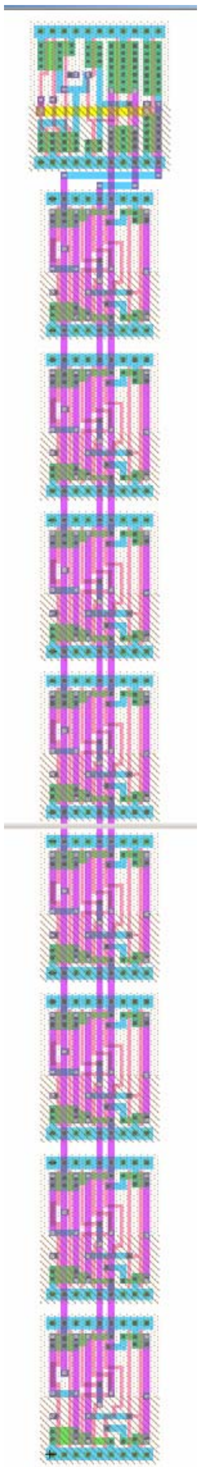
buftri_1x_8

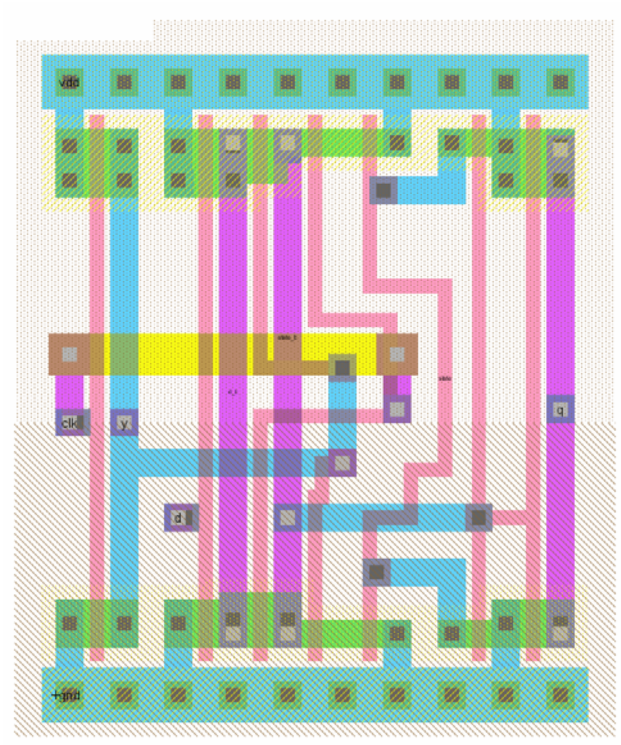


incrementer_1x – a full adder used to create incrementer, an 8-bit ripple carry adder

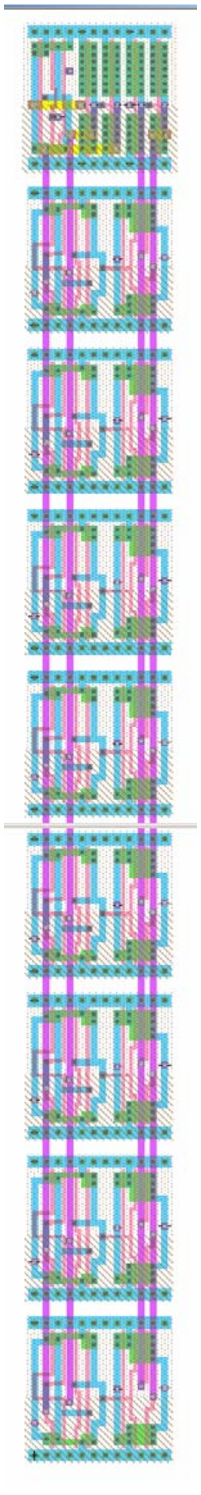


invbuf_1x - a combined buffer and inverting buffer


latchren_1x_8

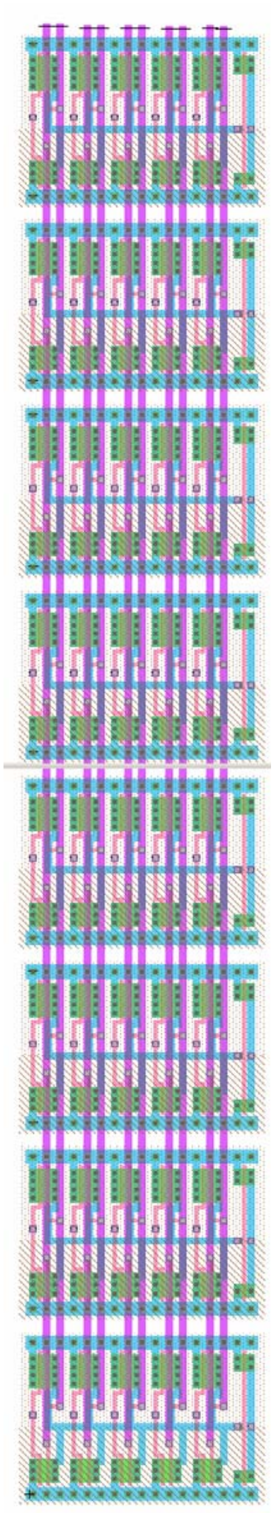


latch_c_1x

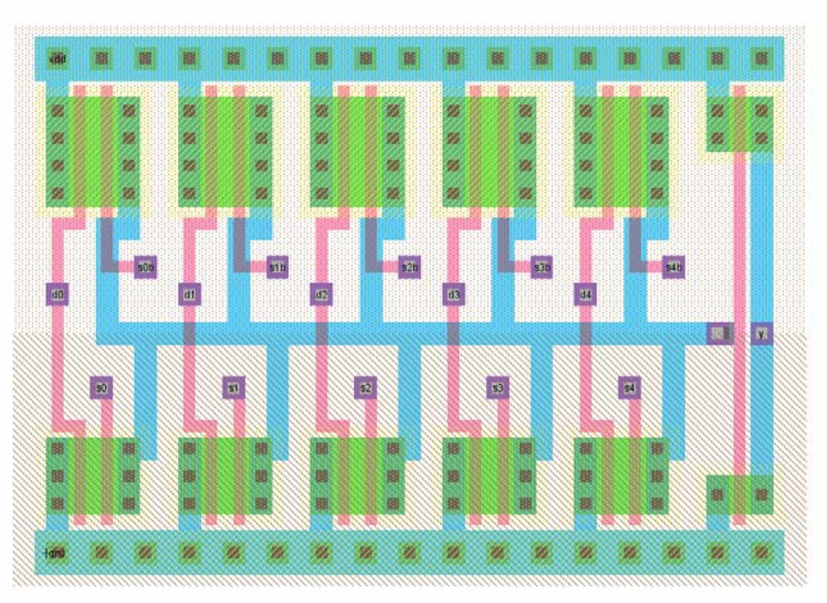


registerbuf_8

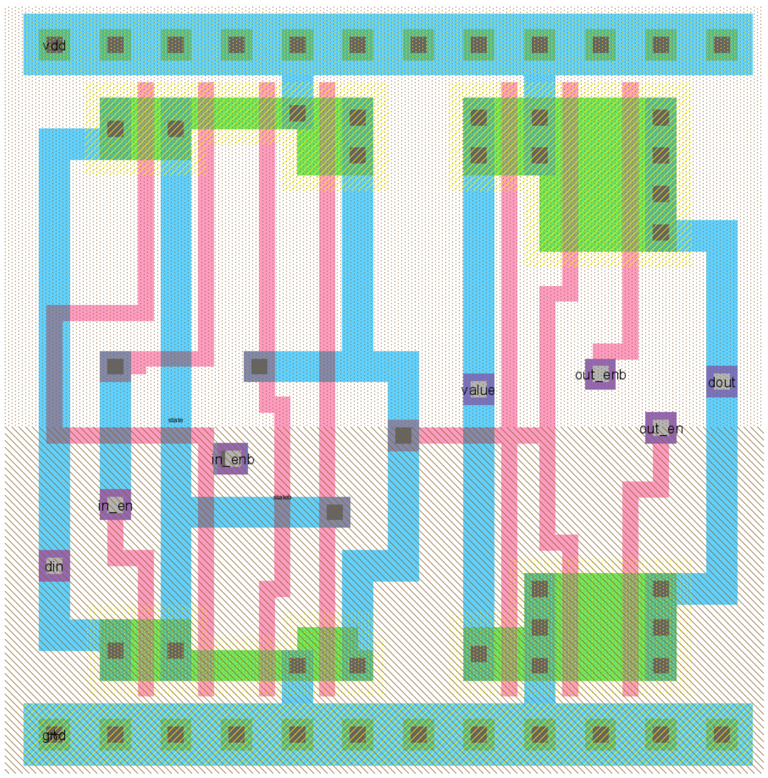
The mux5 was originally built as one piece, but it was decided it would be simpler to split it into a base and a controller. The base would incorporate the 8 bitlines and connect to the mux5 controller which was moved up from the datapath into the controller.



mux5_1x_8

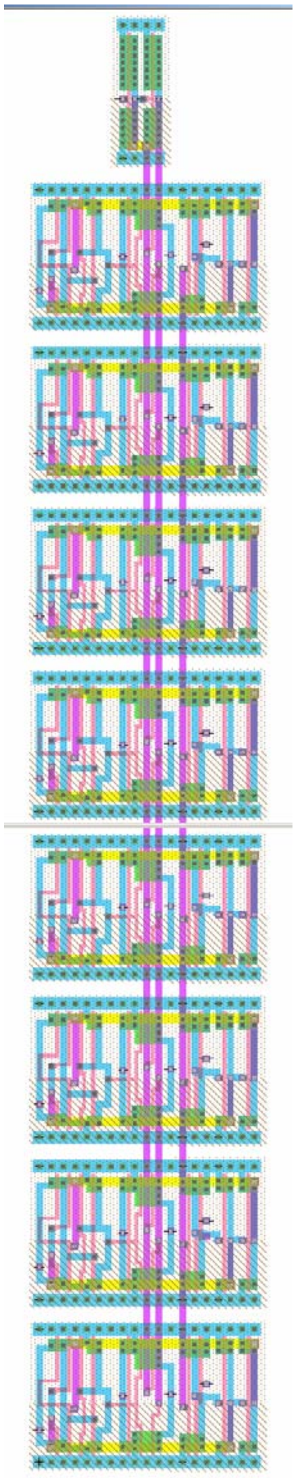


mux5_1x_base

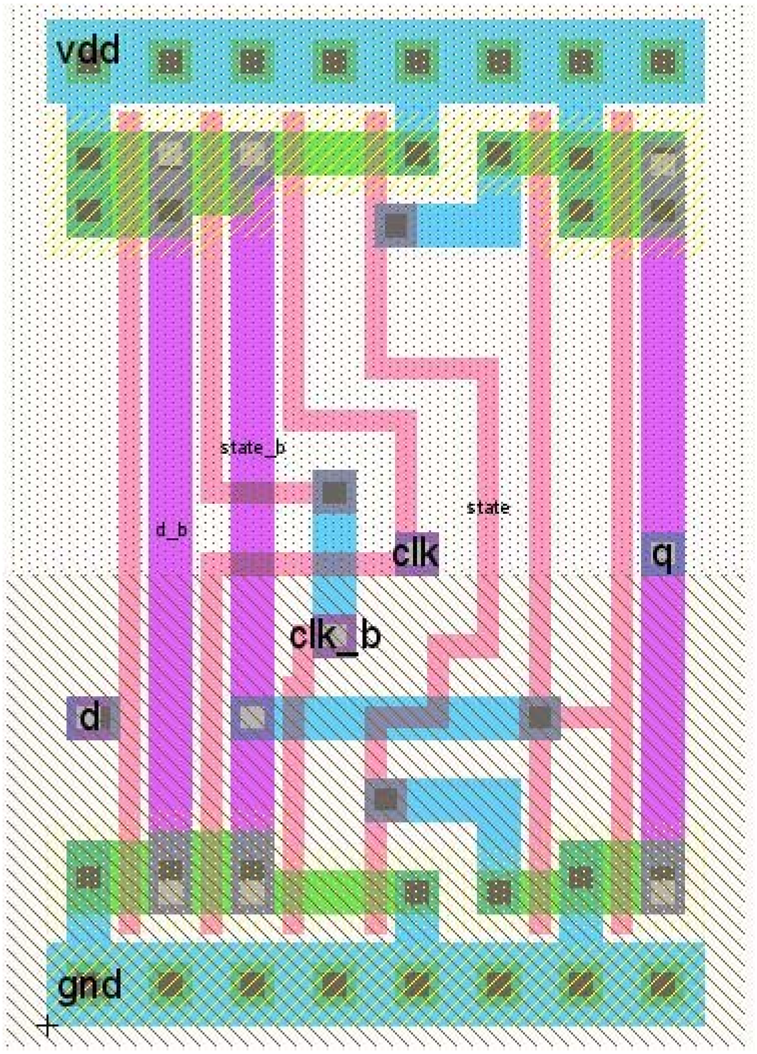


regbitbuf

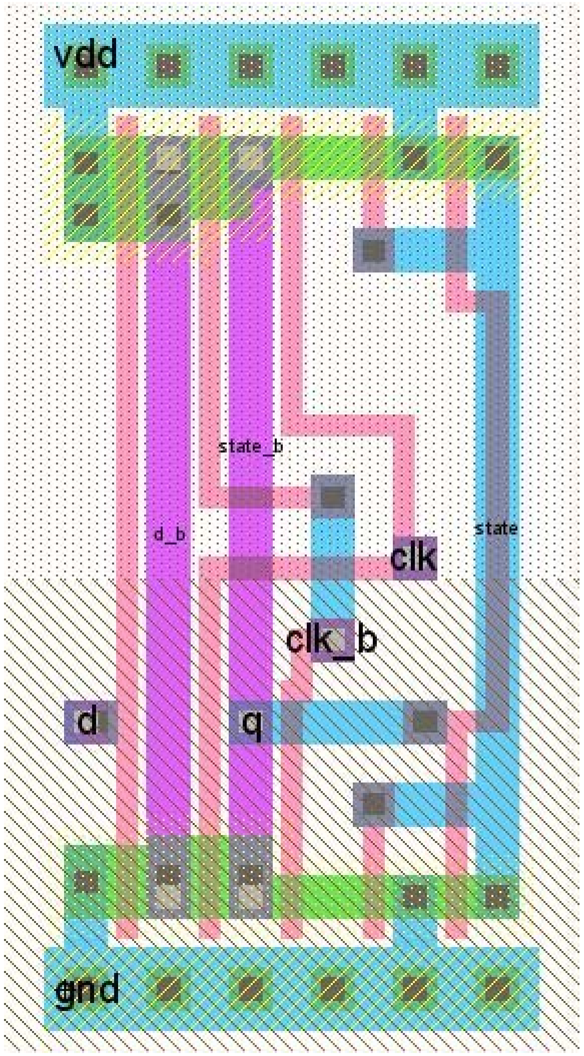
One of the goals was to save metal3 tracks for final wiring of the datapath. However in some cases, like the registerbufmasked_8, free metal3 tracks were used to complete the necessary wiring.



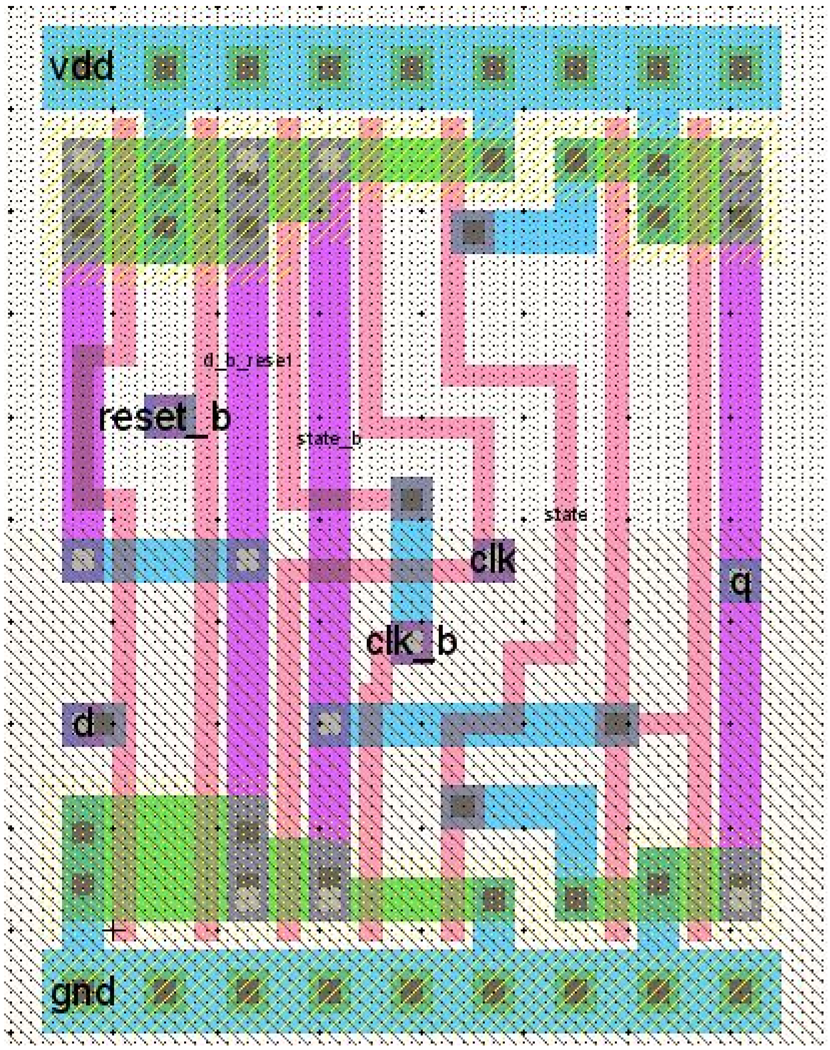
registerbufmasked_8



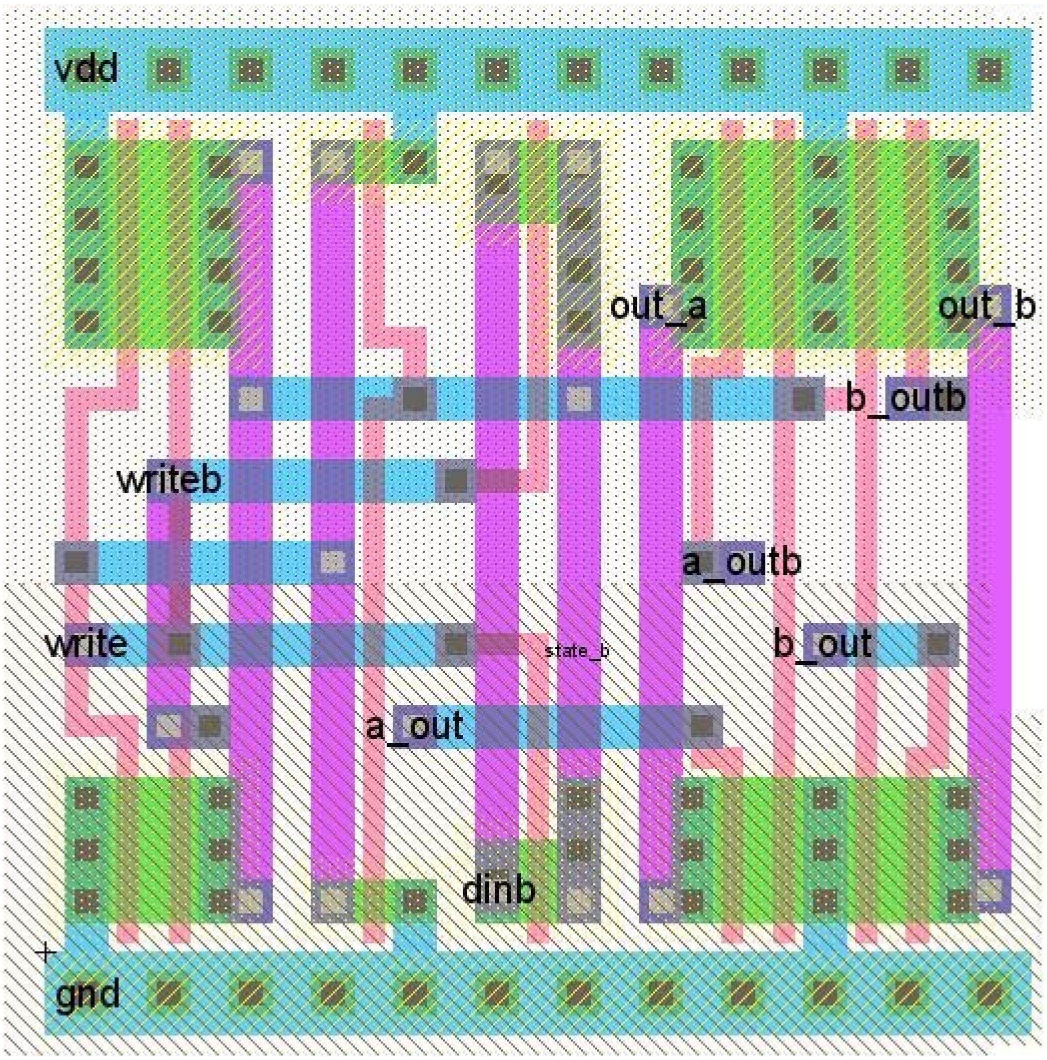
latch_dp_1x - a simple latch that is used in latchen_1x_8, latch_1x_8, latch_1x_7_clkb, latch_1x_8_clkb, and latch_c_1x



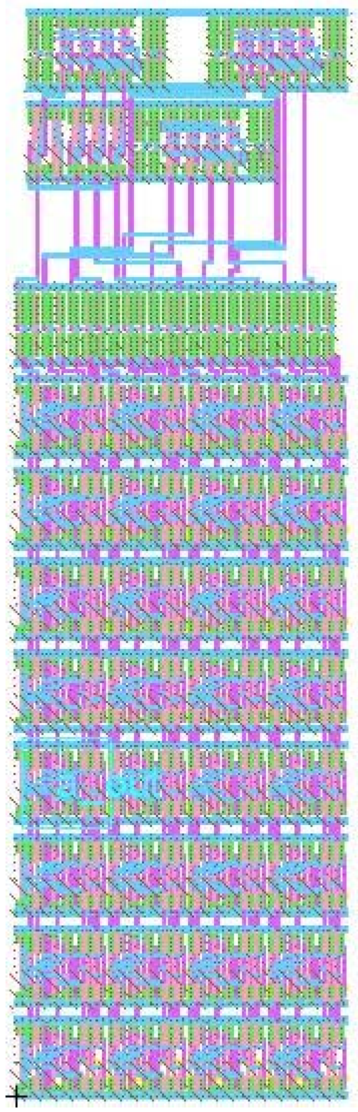
latch_dp_1x_noinv - an alternate version of latch_dp_1x where the output is not inverted; this is used in dp_datain to save datapath space by removing a pair of consecutive inverters

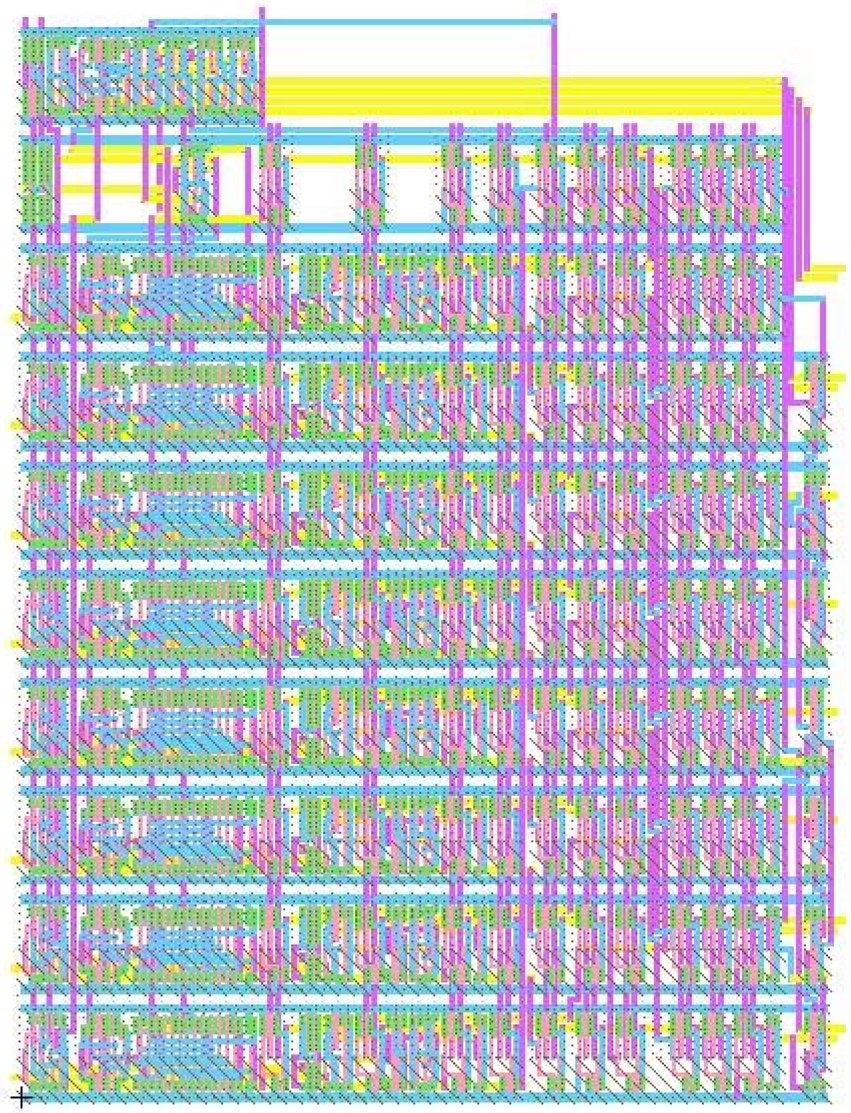


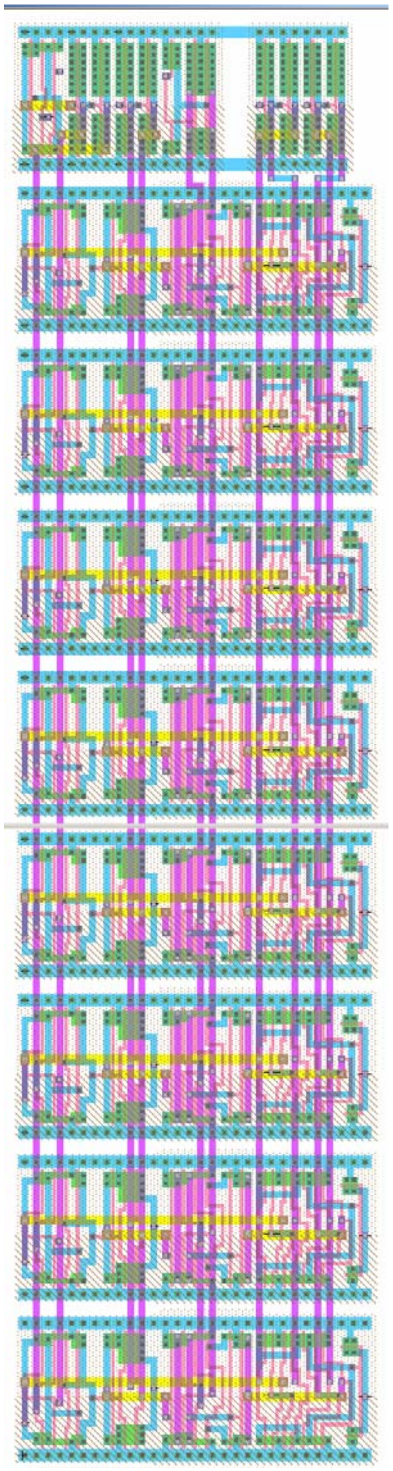
latchr_dp_1x - a latch with a reset; used in latchr_1x_8 and latchren_1x_8

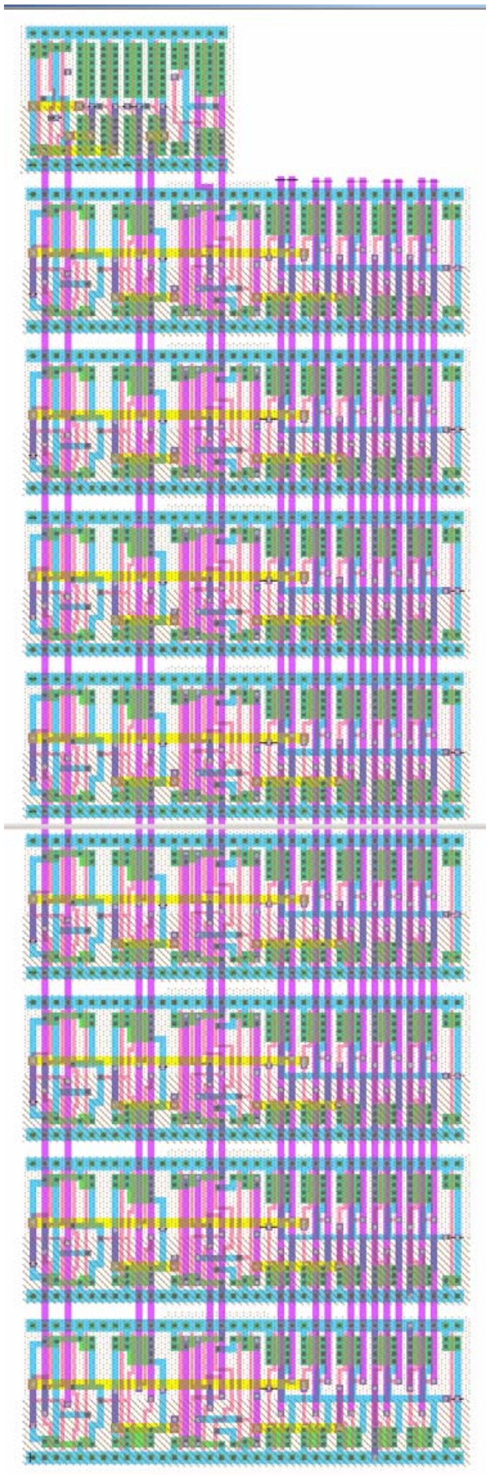


regbit - a 1-bit register; used in regfile_8

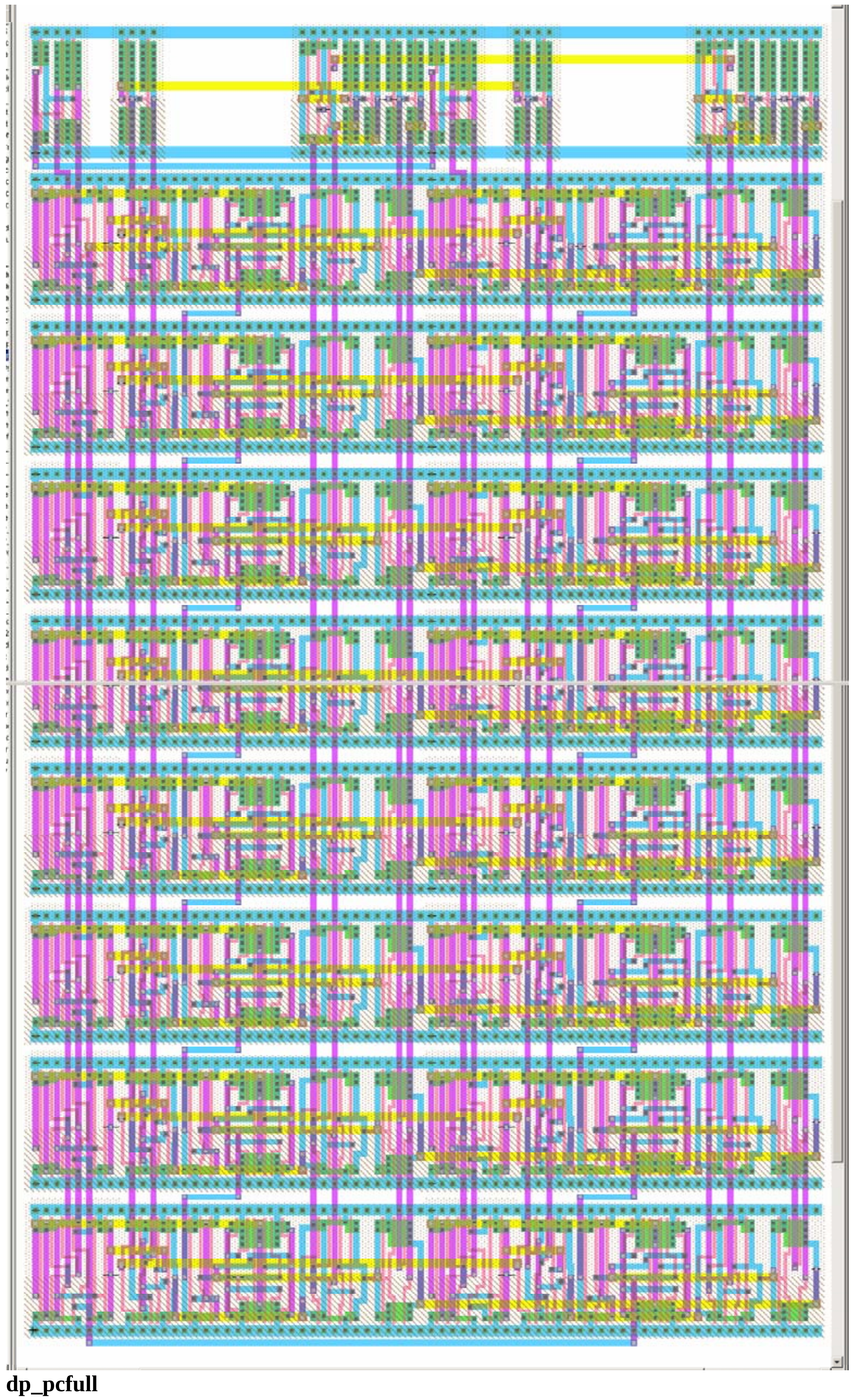

regfile_8 - a register file; the upper four bit-lines contain the control logic for the module, these submodules were placed in the zipper to minimize the overall width of the regfile

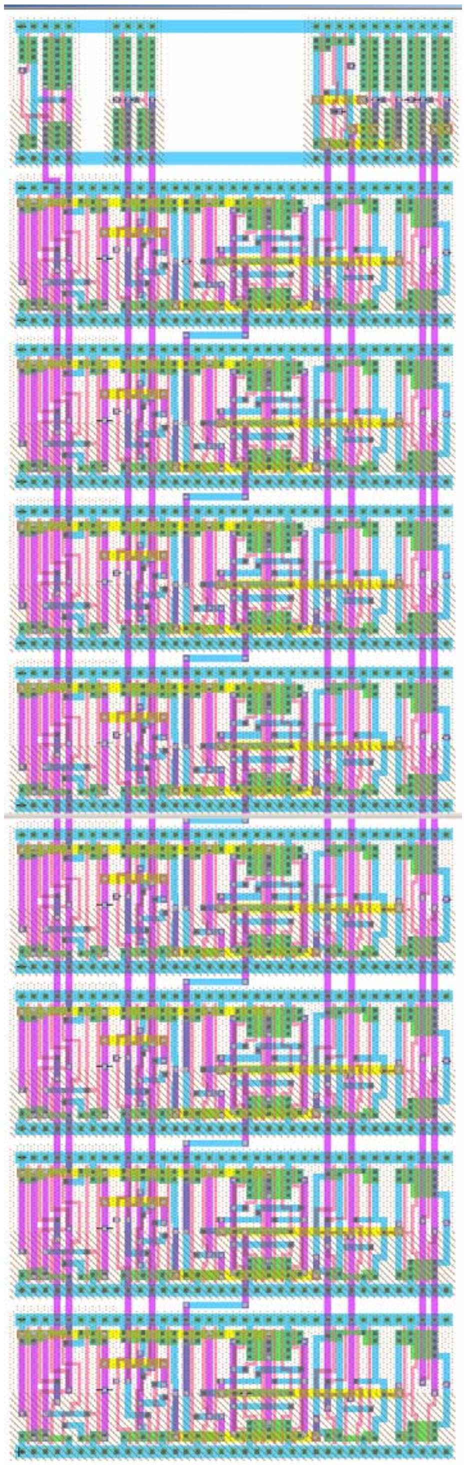

ALU


dp_templ

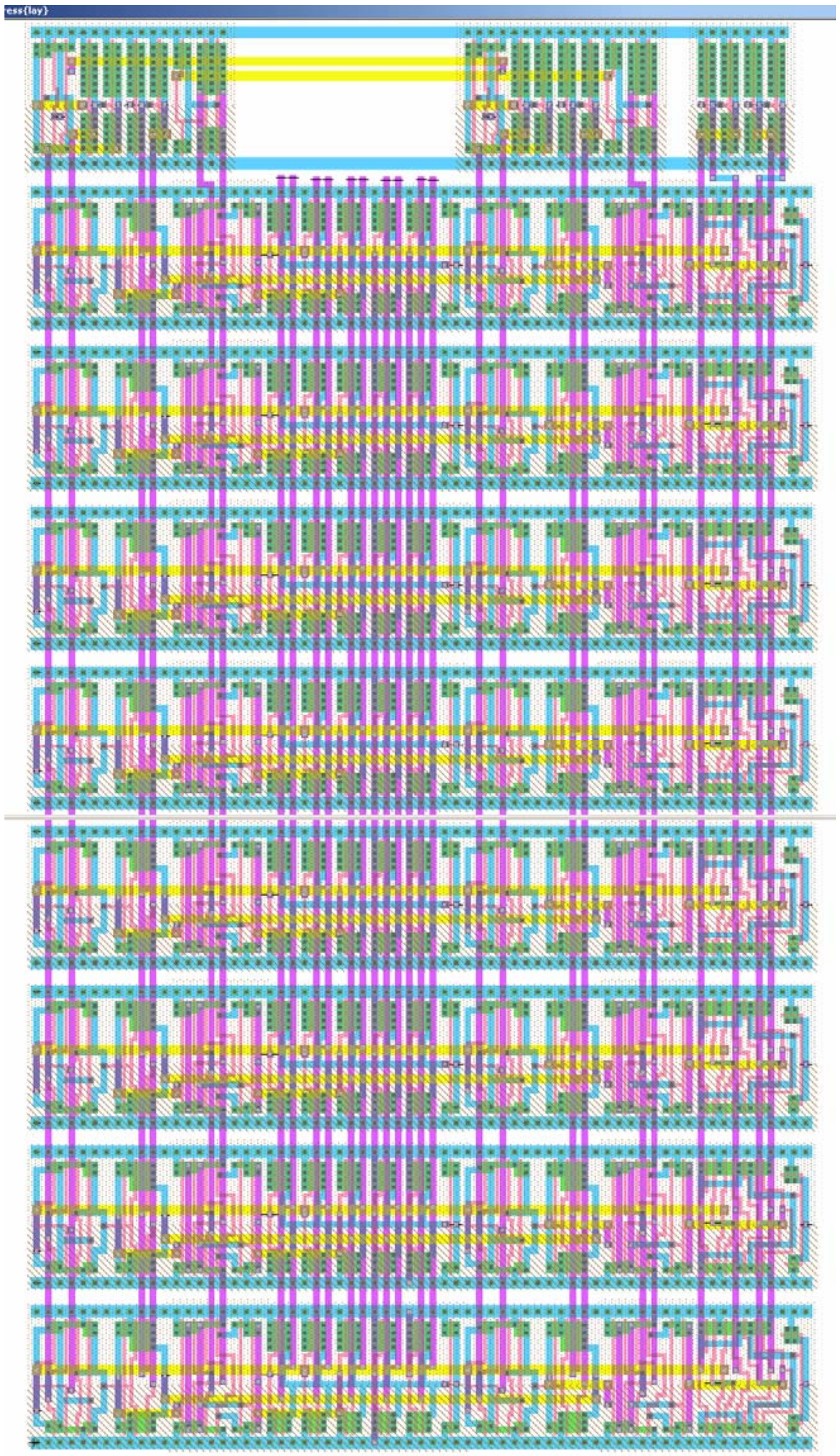


dp_temph

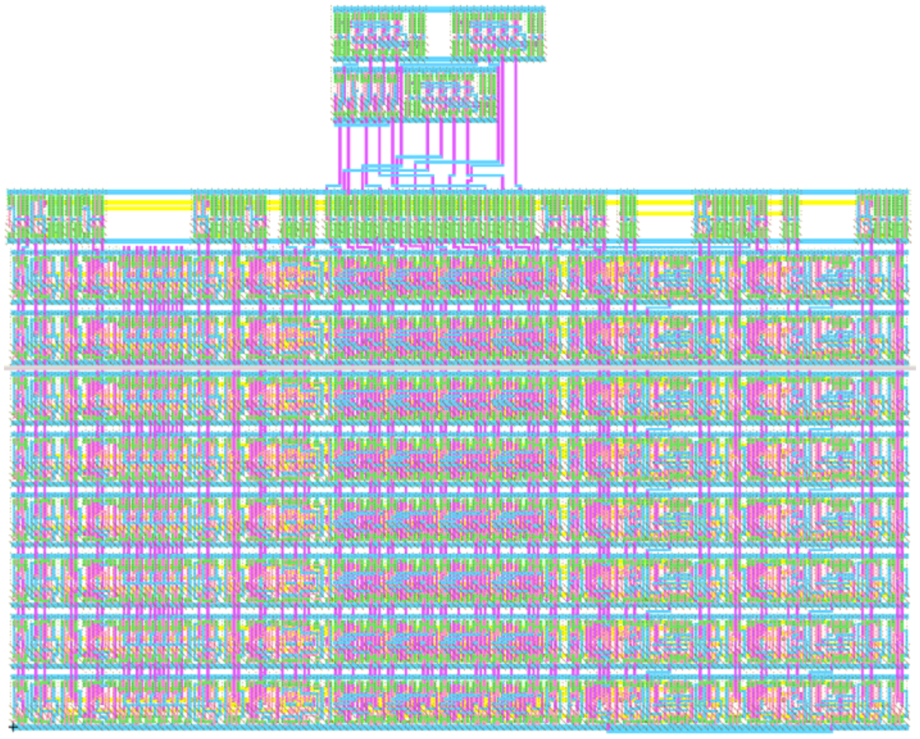

dp_pcfull

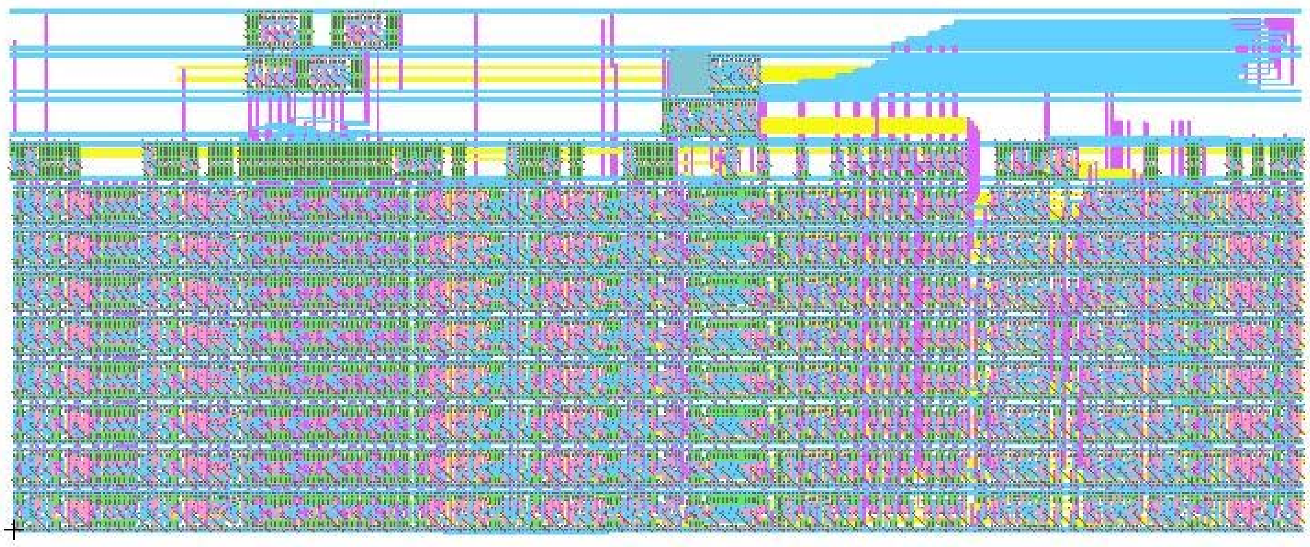


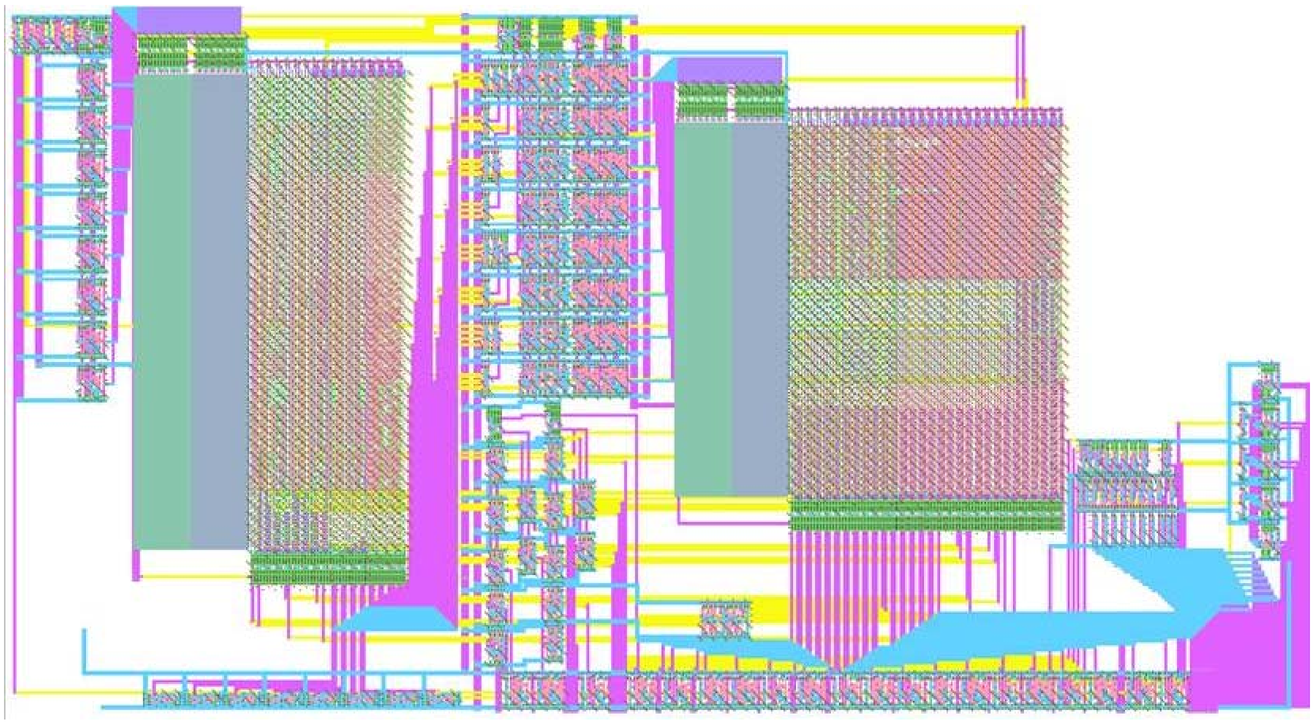
dp_pc

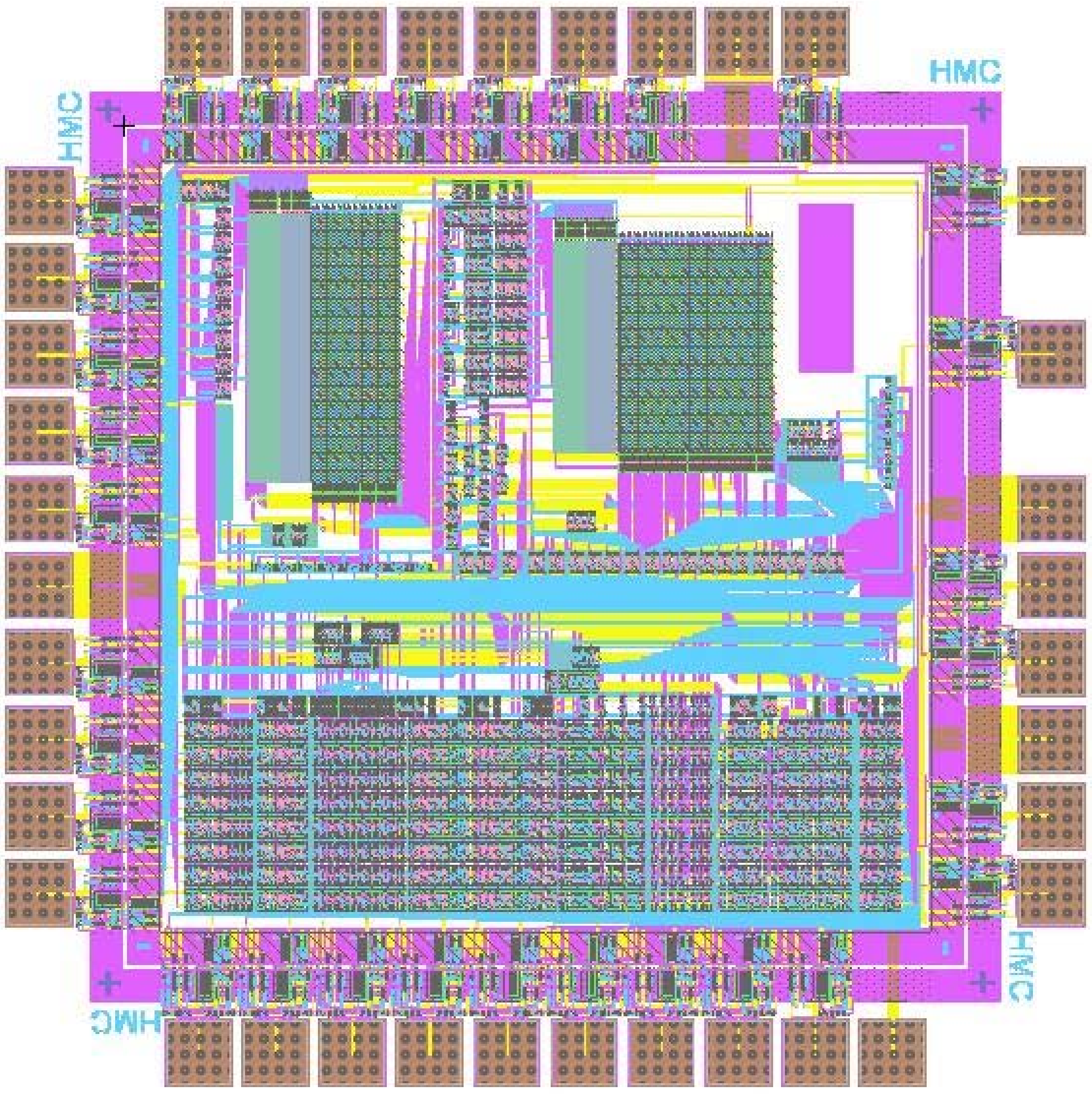


dp_address


dp_a


Datapath


Controller



Full Chip

Appendix: Microcode Assembler

The microcode is written in a human-readable table (Appendix: Microcode Assembler: 6502.ucode). That table breaks the microcode into sections called "states" and each state makes up several lines in the ROM, one of which is run each cycle. The lines describe the values of the control signals relevant to each cycle. When the assembler runs, it sets up all the control signals with default values, reads the correct values for the signals out of this table, and writes the system verilog line describing that cycle to standard output for each cycle. The output is captured and redirected to a file, (6502.ucode.compiled, not included) that is then included in a case statement in the state_pla module of control.sv. Setting default values in the assembler allows unimportant signals to be left out of the table to enhance readability.

The microcode eventually forms the state ROM, but because the states are generalized, some cycles require instruction-specific control signals. This occurs at most once per state, so when a line in the human-readable microcode table specifies "func" for any of the signals, the assembler sets a signal that draws upon the opcode ROM for a subset of the control signals. The opcode ROM contains one entry for every defined opcode in the 6502 architecture and controls the behavior of the chip during the cycles that depend on the specific instruction being executed.

The opcode ROM is generated from the instruction table (Appendix: Microcode Assembler: instrtable.txt) which contains all the entries for the ROM in a human-readable form. The compiler for this table does not set defaults and assumes that every signal is listed in each entry. A python script (Appendix: Microcode Assembler: instrtable2opcodes.py) translates the entries into the system verilog lines included in the opcode_pla module of control.sv. It leaves the state label as text, however and a second script (Appendix: Microcode Assembler: opcode_label2bin.py) uses comments in 6502.ucode.compiled to replace these labels with the appropriate indices into the state ROM.

On Microsoft Windows systems /src/generate_plas.bat will run all three scripts on the appropriate output automatically.

Microcode Assembler: ucasm.py

```
# ucasm.py
# microcode assembler for 6502
# tbarr@cs.hmc.edu, 28oct2007
# hjustice@hmc.edu
# kmarsh@cs.hmc.edu
#
# This code reads in 6520 microcode formatted as in /src/ucode/6502.ucode
# and assembles it into system verilog lines for inclusion into a case
# structure (used to define a ROM).  It uses ordered dictionaries to store
# the values of the control signals and writes them to the standard output
# when it finishes.
#
# The ordered dictionaries are initialized with default values for each
# signal, so the microcode file needs only specify relevant signals.

import sys
from odict import OrderedDict

BASE_STATE = 11

class ParseError(Exception):
    pass

def int2bin(n, count=8):
    """returns the binary of integer n, using count number of digits"""
    return "".join([str((n >> y) & 1) for y in range(count-1, -1, -1)])

def hex2bin(s):
    try:
        i = int(s.split('x')[1], 16)
    except IndexError:
        raise ParseError, "invalid hex string"
    return int2bin(i)

class Vector:
    groups = OrderedDict([('state', None),
                          ('opcode', None),
                          ('internal', None)])

    def __init__(self):
        self.groups['state'] = OrderedDict([
                           ('th_in_en', '0'),
                           ('th_out_en', '0'),
                           ('tl_in_en', '0'),
                           ('tl_out_en', '0'),
                           ('p_sel', '0'),
                           ('p_out_en', '0'),
                           ('pch_in_en', '0'),
                           ('pch_out_en', '0'),
                           ('pcl_in_en', '0'),
                           ('pcl_out_en', '0'),
                           ('pc_inc_en', '1'),
                           ('pc_sel', '0'),
                           ('d_out_sel', '0'),
                           ('ah_sel', '000'),
                           ('al_sel', '00'),
                           ('c_temp_en', '0'),
                           ('carry_sel', '00'),
                           ('flag_en', '0'),
                           ('read_en', '1'),
                           ('constant_en', '0'),
                           ('constant', '00000000')])
        self.groups['opcode'] = OrderedDict([
                           ('alu_op', '0000'),
                           ('d_in_en', '0'),
                           ('reg_write_en', '0'),
                           ('reg_read_addr_a', '00'),
                           ('reg_read_addr_b', '00'),
                           ('reg_write_addr', '00'),
                           ('reg_a_en', '0'),
                           ('reg_b_en', '0')])
        self.groups['internal'] = OrderedDict([
                           ('last_cycle', '0'),
                           ('func_mode', '0'),
                           ('next_source', '00'),
                           ('next_state', '00000000')])

       def bin_rep(self):
        parts = ["".join(group.values()) for group in self.groups.values()]
        return "_".join(parts)

       def total_len(self):
        return len(self.bin_rep().replace('_',''))

       def __setattr__(self, name, value):
        for group in self.groups.values():
            if name in group.keys():
                group[name] = value

class State:
       def __init__(self):
        self.state_num = None
        self.next_state = None
        self.last_state = False
        self.vals = ""
        self.fields = ""

        # input state table
        self.in_states = { 'a_sel' : None,
                           'b_sel' : None,
                           'alu_op' : "pass",
                           'wrt_en' : "none",
                           'pc_w_en' : "0",
                           'pc_sel' : "pc_n",
                           'a_h_sel' : "pc_n",
                           'a_l_sel' : "pc_n",
                           'th_lat' : "0",
                           'tl_lat' : "0",
                           'memwri' : "0",
                           'flag' : "0",
                           'c_sel' : "-",
                           'next_s' : "-",
                           'pcinc' : '1',
                           'sta_src' : '00',
                           'last_cy' : '1',
                           'nxt_src': '0'}
        self.out = Vector()

       def parse_line(self):
        fields_parsed = self.fields.split('\t')
        vals_parsed = self.vals.split('\t')
        if not len(fields_parsed) == len(vals_parsed):
            raise ParseError, "field mismatch"
        for (attr, value) in zip(fields_parsed, vals_parsed):
            if not attr in self.in_states.keys():
                raise ParseError, "invalid field: %s" % attr
            self.in_states[attr] = value
```

```
def make_line(self):
    # parse all the fieldsself.out.
    if '1' in self.in_states['th_lat']:
        self.out.th_in_en = '1'
    if '1' in self.in_states['tl_lat']:
        self.out.tl_in_en = '1'
    if 'db' in self.in_states['a_sel']:
        self.out.d_in_en = '1'
    if 'tl' == self.in_states['a_sel']:
        self.out.tl_out_en = '1'
    if 'th' == self.in_states['a_sel']:
        self.out.th_out_en = '1'
    if 'sp' == self.in_states['a_sel']:
        self.out.reg_read_addr_a = '11'
        self.out.reg_a_en = '1'
    if '0x' in self.in_states['a_sel']:
        self.out.constant_en = '1'
        self.out.constant = hex2bin(self.in_states['a_sel'])

   if 'a' == self.in_states['b_sel']:
        self.out.reg_read_addr_b = '00'
        self.out.reg_b_en = '1'
    if 'x' == self.in_states['b_sel']:
        self.out.reg_read_addr_b = '01'
        self.out.reg_b_en = '1'
    if 'y' == self.in_states['b_sel']:
        self.out.reg_read_addr_b = '10'
        self.out.reg_b_en = '1'
    if 'pc_h' == self.in_states['b_sel']:
        self.out.pch_out_en = '1'
    if 'pc_l' == self.in_states['b_sel']:
        self.out.pcl_out_en = '1'
    if 'p' == self.in_states['b_sel']:
        self.out.p_out_en = '1'
    if 'sp' == self.in_states['wrt_en']:
        self.out.reg_write_addr = '11'
        self.out.reg_write_en = '1'
    if 'p' == self.in_states['wrt_en']:
        self.out.p_sel = '1'
        self.out.flag_en = '1'
    if 'a'== self.in_states['wrt_en']:
        self.out.reg_write_addr = '00'
        self.out.reg_write_en = '1'
    if 'x' == self.in_states['wrt_en']:
        self.out.reg_write_addr = '01'
        self.out.reg_write_en = '1'
    if 'y' == self.in_states['wrt_en']:
        self.out.reg_write_addr = '10'
        self.out.reg_write_en = '1'
    if '1' == self.in_states['pc_w_en']:
        self.out.pch_in_en = '1'
        self.out.pcl_in_en = '1'
    if '10' == self.in_states['pc_w_en']:
        self.out.pch_in_en = '1'
    if '01' == self.in_states['pc_w_en']:
        self.out.pcl_in_en = '1'
    if 'r' == self.in_states['pc_sel']:
        self.out.pc_sel = '1'

    if 'b' == self.in_states['nxt_src']:
        self.out.next_source = '10'

    # address selection
    if 'r' == self.in_states['a_h_sel']:
        self.out.ah_sel = '001' #a1
    if 'temp' == self.in_states['a_h_sel']:
        self.out.ah_sel = '010' #a2
    if '0' == self.in_states['a_h_sel']:
        self.out.ah_sel = '100' #a3
    if '1' == self.in_states['a_h_sel']:
        self.out.ah_sel = '101' #a4
```

```
if 'r' == self.in_states['a_l_sel']:
    self.out.al_sel = '01'
if 'temp' == self.in_states['a_l_sel']:
    self.out.al_sel = '10'
if '1' == self.in_states['memwri']:
    self.out.read_en = '0' # high on READ.

# other latches
if '1' == self.in_states['tl_lat']:
    self.out.tl_in_en = '1'
if '1' == self.in_states['th_lat']:
    self.out.th_in_en = '1'

if 'b' == self.in_states['memwri']:
    self.out.read_en = '0'
    self.out.d_out_sel = '1'
if 'r' == self.in_states['memwri']:
    self.out.read_en = '0'
    self.out.d_out_sel = '0'

if '1' == self.in_states['pcinc']:
    self.out.pc_inc_en = '1'
if '0' == self.in_states['pcinc']:
    self.out.pc_inc_en = '0'
if '1' == self.in_states['flag']:
    self.out.flag_en = '1'
if 't' == self.in_states['flag']:
    self.out.flag_en = '0'
    self.out.c_temp_en = '1'

# todo: flag selection, flag enable, branch enable
if 'pass' == self.in_states['alu_op']:
    self.out.alu_op = int2bin(0x0, 4)
    self.out.carry_sel = '10'
if 'inc' == self.in_states['alu_op']:
    self.out.alu_op = int2bin(0x0, 4)
    self.out.carry_sel = '11'
if 'pass+t' == self.in_states['alu_op']:
    self.out.alu_op = int2bin(0x0, 4)
    self.out.carry_sel = '01'
if 'add' == self.in_states['alu_op']:
    self.out.alu_op = int2bin(0x2, 4)
    self.out.carry_sel = '10'
if 'add+1' == self.in_states['alu_op']:
    self.out.alu_op = int2bin(0x2, 4)
    self.out.carry_sel = '11'
if 'add+t' == self.in_states['alu_op']:
    self.out.alu_op = int2bin(0x2, 4)
    self.out.carry_sel = '01'
if 'dec' == self.in_states['alu_op']:
    self.out.alu_op = int2bin(0x1, 4)
    self.out.carry_sel = '11'
if 'rol' == self.in_states['alu_op']:
    self.out.alu_op = int2bin(0x6, 4)

# carry source selection -- set by alu_op unless c_sel is present.
# If c_sel is present and not '-', then the carry selection made by
# alu_op gets overridden.
if '-' == self.in_states['c_sel']:
    pass
if 'p' == self.in_states['c_sel']:
    self.out.carry_sel = '00'
if 't' == self.in_states['c_sel']:
    self.out.carry_sel = '01'
if '0' == self.in_states['c_sel']:
    self.out.carry_sel = '10'
if '1' == self.in_states['c_sel']:
    self.out.carry_sel = '11'

if 'opcode' == self.in_states['sta_src']:
    self.out.next_source = '01'
```

```
        if 'func' in self.in_states.values():
            self.out.func_mode = '1'
        if '0' == self.in_states['last_cy']:
            self.out.last_cycle = '0'

    def __repr__(self):
        return "state %s: %s" % (self.state_num, self.in_states)

def process_block(block, next_state_num, labeled_states):
    # assign everybody a state num
    for state in block[2]:
        state.state_num = next_state_num
        state.next_state = next_state_num + 1
        next_state_num += 1

    # go back to base state on last
    block[2][-1].next_state = BASE_STATE
    block[2][-1].last_state = True

    toprint = "// %s:%s\n" % (block[0], block[2][0].state_num)

    # now parse and generate the blocks
    for state in block[2]:
        state.parse_line()
        state.out.__init__()
        if state.last_state:
            state.out.last_cycle = '1'
            if state.in_states['next_s'] != "-":
                state.out.last_cycle = '0'
                block[2][-1].next_state = labeled_states.get(state.in_states['next_s'])
        state.make_line()
        state.out.next_state = int2bin(state.next_state)
        vector = state.out.bin_rep()

        # for use if I'm feeling cruel.
        hex_vector = hex(int(vector.replace('_',''), 2))[2:].replace('L','')

        toprint += "8'd%03d : out_controls <= %d'b%s;\n" % (state.state_num,
                                               state.out.total_len(),
                                               vector)

    # print block
    print "\n" + toprint.strip()
    return next_state_num

def do_file():
    try:
        f = open(sys.argv[1])
    except IndexError:
        f = open('6502.ucode')
    current_block = [None, "", []]
    next_state_num = 0

    labeled_states = []

    print "// generated by ucasm"
    sizevec = Vector()
    print "// c_state = %s" % len("".join(sizevec.groups['state'].values()))
    print "// c_op = %s" % len("".join(sizevec.groups['opcode'].values()))
    print "// c_internal = %s" % len("".join(sizevec.groups['internal'].values()))

    for line in f:
        if line[0] in ['#', '\n']:
            # ignore comments
            continue
        if line.strip() == '':
            continue
        elif line.strip()[-1] == ':':
            # finish old block
            if current_block[0]:
                next_state_num = process_block(current_block, next_state_num, dict(labeled_states))
            # set current block
            current_block[2] = []
            current_block[1] = ""
            current_block[0] = line.strip()[:-1]
            labeled_states.append((current_block[0], next_state_num))
            continue
        else:
            if not current_block[1]:
                current_block[1] = line.strip()
            else:
                new_state = State()
                new_state.fields = current_block[1]
                new_state.vals = line.strip()
                current_block[2].append(new_state)

    # one left...
    next_state_num = process_block(current_block, next_state_num, dict(labeled_states))

    print "\n// state signals"
    for signal in sizevec.groups['state'].keys():
        print "//  %s," % signal
    print "\n// op signals"
    for signal in sizevec.groups['opcode'].keys():
        print "//  %s," % signal

if __name__ == "__main__":
    do_file()
```

Microcode: instrtable2opcodes.py

```
# Author:   Kyle Marsh <KMarsh@cs.hmc.edu>, Harvey Mudd College
# Date:     11 March 2008
#
#   IN:
#       *argv[1]                # Table of instructions including opcode,
#                               # instruction name, addressing mode, srcA,
#                               # srcB, dest, and aluop.
#
#   OUT:
#       *opcodes.txt            # list of opcodes generated from the table
#                               # with next-state labels instead of numbers.
#
#   NOTES:
#       *The opcode output is system verilog in the following form:
#           8'hXX: out_data <= 31'b<c_sel>_<aluop>_<d_in_en>_<reg_write_en>_
#               <reg_read_addr_a>_<reg_read_addr_b>_<reg_write_addr>_
#               <reg_a_en>_<reg_b_en>__<branch_polarity>_<flags>_<state>;
#
#       *This code reads and writes to files instead of standard IO because it
#       may be run on Windows machines and I don't know how the Windows shell
#       handles IO redirection.

help = """Useage: python instrtable2opcodes.py infile.txt
Script to take in a table of instructions and generate lines of system verilog
to be included in the case statement of the opcode_pla module in control.sv.
This script leaves the next state label as a string, so
/src/ucode/opcode_translator/opcode_label2bin.py must be run on its output
to convert those into bitfields.

Output lines are in the following form:
    8'hXX: out_data <= 31'b<c_sel>_<aluop>_<d_in_en>_<reg_write_en>_
        <reg_read_addr_a>_<reg_read_addr_b>_<reg_write_addr>_
        <reg_a_en>_<reg_b_en>__<branch_polarity>_<flags>_<next_state_label>;
"""

import re
import sys
import os

# Global Constants (order of fields in the input file)
FIELDS = 10
OPCODE = 0
```

```
INSTR = 1
STATE = 2
SRCA = 3
SRCB = 4
DEST = 5
ALUOP = 6
C_SEL = 7
BP = 8
FLAGS = 9

def usage():
    sys.stderr.write(help)

def int2bin(n, count=8):
    """returns the binary of integer n, using count number of digits"""
    return "".join([str((n >> y) & 1) for y in range(count-1, -1, -1)])

def hex2int(n):
    """fairly hacky function to convert a hex number in a string to an int."""
    hex = {'a':10, 'b':11, 'c':12, 'd':13, 'e':14, 'f':15}
    sum = 0
    for i in range(len(n)):
        if hex.__contains__(n[i]):
            sum += 16**(len(n)-i-1)*hex[n[i]]
        else:
            sum += 16**(len(n)-i-1)*int(n[i])
    return sum

def main():

    # Open the input file in a semi-robust manner.
    try:
        infilename = sys.argv[1]
    except IndexError:
        usage()
        sys.stderr.write('Please provide the input filename as the first '
                + 'argument on the command line.\n')
        sys.exit(1)
    try:
        infile = open(infilename)
    except IOError:
        usage()
        sys.stderr.write('%s is not a valid file.\n' %infilename)
        sys.exit(1)

    # Read the input file into a list, split the list on tab characters,
    # and throw away lines that do not contain 7 fields. Finally, trim the
    # last field to only one character in case it contains comments.
    input_list = [line.split('\t') for line in infile.readlines()]
    input_list = [line for line in input_list if len(line) == FIELDS]

    for line in input_list:
        line[ALUOP] = line[ALUOP][0]
        line[FLAGS] = line[FLAGS][0:2]

    # Write the lines of output based on the input:
    outfile = open('opcodes.txt', 'w')
    for line in input_list:
        opcode = str(line[OPCODE])
        aluop = str(int2bin(hex2int(line[ALUOP].lower()), 4))
        # If data comes from memory, disable register in:
        if line[SRCA] == 'dp':
            d_in_en = '1'
            reg_a_en = '0'
            reg_read_addr_a = '00'
        # Otherwise, set SrcA to a register:
        elif line[SRCA] == 'A':
            d_in_en = '0'
            reg_a_en = '1'
            reg_read_addr_a = '00'
        elif line[SRCA] == 'X':
```

```
    d_in_en = '0'
    reg_a_en = '1'
    reg_read_addr_a = '01'
elif line[SRCA] == 'Y':
    d_in_en = '0'
    reg_a_en = '1'
    reg_read_addr_a = '10'
elif line[SRCA] == 'SP':
    d_in_en = '0'
    reg_a_en = '1'
    reg_read_addr_a = '11'
else:
    d_in_en = '0'
    reg_a_en = '0'
    reg_read_addr_a = '00'

# Set SrcB:
if line[SRCB] == '_':
    reg_b_en = '0'
    reg_read_addr_b = '00'
elif line[SRCB] == 'A':
    reg_b_en = '1'
    reg_read_addr_b = '00'
elif line[SRCB] == 'X':
    reg_b_en = '1'
    reg_read_addr_b = '01'
elif line[SRCB] == 'Y':
    reg_b_en = '1'
    reg_read_addr_b = '10'
elif line[SRCB] == 'SP':
    reg_b_en = '1'
    reg_read_addr_b = '11'

# Set Dest:
if line[DEST] == 'A':
    reg_write_en = '1'
    reg_write_addr = '00'
elif line[DEST] == 'X':
    reg_write_en = '1'
    reg_write_addr = '01'
elif line[DEST] == 'Y':
    reg_write_en = '1'
    reg_write_addr = '10'
elif line[DEST] == 'SP':
    reg_write_en = '1'
    reg_write_addr = '11'
else:
    reg_write_en = '0'
    reg_write_addr = '00'

# Set Carry Select:
if line[C_SEL] == '0':
    carry_select = '10'
elif line[C_SEL] == '1':
    carry_select = '11'
elif line[C_SEL] == 'p':
    carry_select = '00'
else:
    sys.stderr.write('Bad carry select: %s\n' %line[OPCODE])
    sys.exit(2)

branch_polarity = str(line[BP])
flags = str(int2bin(hex2int(line[FLAGS].lower()), 8))
label = line[STATE]
instruction = line[INSTR]

# Write the output to the file.
outfile.write("8'h%s: out_data <= 33'b%s_%s_%s_%s_%s_%s_%s_%s_%s__%s_%s_%s; //%s (%s)\n"
        %(opcode, carry_select, aluop, d_in_en, reg_write_en,
          reg_read_addr_a, reg_read_addr_b, reg_write_addr,
          reg_a_en, reg_b_en, branch_polarity, flags, label,
                  instruction, label))

    outfile.close()

if __name__ == "__main__":
    main()
```

Microcode: opcode_label2bin.py

```
# Author:   Kyle Marsh <kmarsh@cs.hmc.edu>, Harvey Mudd College
# Date:     04 March 2008
#
#   IN:
#       *argv[1]                # List of lines destined for control.sv with
#                               # next-state labels instead of ROM indices.
#       *6502.ucode.compiled    # File containing state labels and their
#                               # corresponding ROM indices in decimal.
#       *argv[2]                # Optional expected output file for testing.
#
#   OUT:
#       *tranlsated_opcodes.txt # List of lines for control.sv with
#                               # next-state indices in place
#       *out.diff               # If argv[2] supplied and valid, contains
#                               # `diff translated_opcodes.txt argv[2]'
#
#   NOTES:
#       This code reads and writes to files instead of standard IO because it
#       may be run on Windows machines and I don't know how the Windows shell
#       handles IO redirection.

help = """Useage: python opcode_label2bin.py infile [expected_output]
Translation script to replace next-state names in lines of the input file with
their corresponding ROM indices as defined in 6502.ucode.compiled.

If the optional expected_output file is included, this is compared against
the output of the script using diff and the result is sent to out.diff.  If
diff does not produce output the script prints the message "The actual
output matches the expected output." to stderr.
"""

import re
import sys
import os

# Number of bits in binary representation of the state; if we can extract
# this from the label file we shoud do so:
NUMBITS = 8

def usage():
    sys.stderr.write(help)

def int2bin(n, count=8):
    """returns the binary of integer n, using count number of digits"""
    return "".join([str((n >> y) & 1) for y in range(count-1, -1, -1)])

def main():

    # Open the input file in a semi-robust manner.
    try:
        infilename = sys.argv[1]
    except IndexError:
        usage()
        sys.stderr.write('Please provide the input filename as the first '
                + 'argument on the command line.\n')
        sys.exit(1)
    try:
        infile = open(infilename)
    except IOError:
        usage()
        sys.stderr.write('%s is not a valid file.\n' %infilename)
        sys.exit(1)
```

```
# Providing a second filename on the command line initiates debug mode
# which will diff the output against the expected output and save the
# result of that in out.diff.  This is *nix specific, I think.
try:
    expectedoutfilename = sys.argv[2]
    debug = True
    try:
        expectedoutfile = os.access(expectedoutfilename, os.R_OK)
    except IOError:
        usage()
        sys.stderr.write('%s is not a valid file.\n' %infilename)
        sys.exit(1)
except IndexError:
    debug = False

# Hard-coded relative path to compiled microcode.
namefile = open('../6502.ucode.compiled')
name_list = namefile.readlines()
namefile.close()

# Suck out the lines with the labels and indices.  First chop off the
# leading comments which get in the way.  Assumes we know a significant
# amount about the format of 6502.ucode.compiled **and that the format
# doesn't change -- this is non-robust code**.
name_list = name_list[4:]
name_list = [line.strip('// ').strip() for line in name_list if
        re.match('^// [a-z_]+:[0-9]+$', line)]

if debug: # Write the list of labels to a file.
    namelistoutfile = open('labels_list.txt', 'w')
    namelistoutfile.write('\n'.join(name_list))
    namelistoutfile.close()

# First part of each element is the key, second is the value; turn it
# into a dictionary.  All non alphanumeric characters in the key are
# escaped to make it regex-safe later.
name_dict = {}
for line in name_list:
    line = line.split(':')
    name_dict[re.escape(line[0])] = int2bin(int(line[1]), NUMBITS)

if debug: # Write the dictionary of labels to a file.
    namedictoutfile = open('labels_dict.txt', 'w')
    dictstr = ''
    for elem in name_dict:
        dictstr += str(elem) + ':' + str(name_dict[elem]) + '\n'
    namedictoutfile.write(dictstr)
    namedictoutfile.close()

# Read and parse the input list.  Uses an even more magic list
# comprehension that matches one of the values in name_dict against each
# line and replaces it with its corresponding value.
# Note: this assumes that each input line has the format
# '^.*[10]_pat; //comment$' in order to avoid something like 'abs' matching
# 'abs_x'.
input_list = infile.readlines()
infile.close()
output_list = [re.sub(pat, name_dict[pat], line, 1) for line in input_list
        for pat in name_dict if re.search('[10]_' +
            pat + '; //[a-z \)\(_]+$', line)]

# Write out the new file.
outfile = open('translated_opcodes.txt', 'w')
output_string = ''.join(output_list)
outfile.write(output_string)
outfile.close()

if debug: # Diff the output against the expected output.
    os.system('diff %s translated_opcodes.txt > out.diff'
            %expectedoutfilename)
        if not os.stat('out.diff').st_size:
            sys.stderr.write("Actual output matches expected output.\n")
            os.unlink('out.diff')
        else:
            sys.stderr.write("Actual output differs from expected output:\n")
            os.execlp('cat', 'cat', 'out.diff')


if __name__ == "__main__":
    main()
```

Microcode: 6502.ucode

```
# 6502.ucode
# 6502 microcode table for hmc-6502
# 28oct07
# tbarr at cs hmc edu
# hjustice at hmc edu
# kmarsh at cs hmc edu
#
# Microcode for the Mudd II hmc-6502.  This table is used as input by
# /src/ucode/ucasm.py to generate the contents of the system verilog case
# structure in control.sv for the state ROM (called state_pla in the verilog).
#
# Each labeled section, called a state, executes some number of cycles,
# referred to as cycles or lines.  Each line describes the control signals
# that are important to that cycle.  Each signal listed will be set by
# /src/ucode/ucasm.py when it assembles this table. If any signal is listed
# as "func", the opcode ROM is used to determine the control signals rather
# than the state ROM -- see func_mux, c_op_state, and c_op_opcode in
# control.sv to see which signals these are.
#
# Finally, two warnings: because a "func" listed anywhere in a line will
# trigger the opcode ROM, other signals may be set in that line that are not
# actually used because they will be selected from the opcode ROM instead of
# the state ROM.  This is confusing and the assembler should have been
# written to prevent it, but it is something we must deal with.  Second,
# when next_s is listed for the last cycle in a state, the state that gets
# jumped to must appear earlier in this file than the first reference to it.
# Basically the assembler takes a single pass and has to know about a state
# before it can reference it.

reset:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  pcinc
0x00    -       pass    p       p       -       -       -       -       0
0x00    -       pass    a       0       -       -       -       -       0
0x00    -       pass    x       0       -       -       -       -       0
0x00    -       pass    y       0       -       -       -       -       0
0xff    -       pass    sp      0       -       -       -       -       0
# load program counter from {0xfffd, 0xfffc}.
0xff    -       pass    -       0       -       -       -       1       0
0xfd    -       pass    -       0       -       temp    r       -       0
db      -       pass    pc_h    10      r       -       -       0       0
0xfc    -       pass    -       0       -       temp    r       -       0
db      -       pass    pc_l    01      r       -       -       0       0
0x00    -       pass    -       0       -       pc_n    pc_n    0       0

# ROM index of base needs to be hard coded into ucasm.py.  If you make any
# changes above this point, make sure to update ucasm.py.
base:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    sta_src last_cy
db      -       pass    none    1       pc_n    pc_n    pc_n    0       0       opcode  0

branch_taken:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel pcinc   flag    tl_lat  th_lat  c_sel
tl      -       rol     none    0       -       -       -       0       t       0       0       -
0x00    -       dec     none    0       -       -       -       0       0       0       1       t
th      pc_h    add     pc_h    10      r       -       -       0       0       0       0       0
tl      pc_l    add     pc_l    01      r       -       -       0       t       0       0       0
0x0     pc_h    add+t   pc_h    10      r       r       pc_n    0       0       0       1       -
```

```
branch_head:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel pcinc   flag    tl_lat  th_lat  nxt_src
db      -       pass    none    1       pc_n    pc_n    pc_n    1       t       1       0       b
# if we don't take the branch, go back to base.

none:
# just fetch next instruction
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag
-       -       pass    none    0       -       pc_n    pc_n    0       0

single_byte:
a_sel   b_sel   alu_op  wrt_en  pcinc   pc_sel  a_h_sel a_l_sel tl_lat  flag
func    func    func    func    0       -       pc_n    pc_n    0       1

set_flag:
a_sel   b_sel   alu_op  wrt_en  pcinc   pc_sel  a_h_sel a_l_sel tl_lat  flag
0xff    -       pass    p       0       -       pc_n    pc_n    0       1

clear_flag:
a_sel   b_sel   alu_op  wrt_en  pcinc   pc_sel  a_h_sel a_l_sel tl_lat  flag
0x00    -       pass    p       0       -       pc_n    pc_n    0       1

imm:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag
db      func    func    func    1       pc_n    pc_n    pc_n    0       1

mem_ex_zpa:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    next_s
db      -       pass    none    0       pc_n    0       r       0       0       imm

mem_wr_final:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    memwri
db      -       -       none    1       pc_n    pc_n    pc_n    0       1       0

mem_wr_zpa:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    memwri  next_s
db      func    pass    none    0       pc_n    0       r       0       0       b       mem_wr_final

mem_wr_zpxy2:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    memwri  next_s
-       func    -       none    0       -       0       temp    0       0       b       mem_wr_final

mem_wr_zpx:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    memwri  next_s
db      x       add     none    0       pc_n    0       -       1       0       0       mem_wr_zpxy2

mem_wr_zpy:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    memwri  next_s
db      y       add     none    0       pc_n    0       -       1       0       0       mem_wr_zpxy2

mem_wr_abs:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    memwri  next_s
db      -       pass    none    1       pc_n    pc_n    pc_n    1       0       0       -
db      func    pass    none    0       -       r       temp    0       0       b       mem_wr_final

mem_wr_abxy2:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat  flag    memwri  next_s
db      -       pass+t  none    0       -       -       -       1       0       0       0       -
db      func    pass    none    0       -       temp    temp    0       0       0       b
mem_wr_final

mem_wr_abx:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat  flag    memwri  next_s
db      x       add     none    1       pc_n    pc_n    pc_n    0       1       t       0
mem_wr_abxy2

mem_wr_aby:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat  flag    memwri  next_s
db      y       add     none    1       pc_n    pc_n    pc_n    0       1       t       0
mem_wr_abxy2
```

```
mem_wr_idy:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat  flag    memwri  next_s
# Address of low byte comes in over data line -- save it in temp_high and pass it to address
db      -       pass    none    0       -       0       r       1       0       0       0       -
# Low byte comes in on data -- pass it and store in temp_low
db      -       pass    none    0       -       pc_n    pc_n    0       1       0       0       -
# Increment address of low byte to get address of high byte -- pass to address
th      -       inc     none    0       -       0       r       0       0       0       0       -
# Data from high byte comes in -- save in temp_high
db      -       pass    none    0       -       -       -       1       0       0       0       -
# Add y to low byte and save back in low byte
tl      y       add     none    0       -       temp    r       0       1       t       0       -
# Add carry to high byte and save back in high byte
th      -       pass+t  none    0       -       r       temp    1       0       0       0       -
# Use temporary register as address and write whatever func says
-       func    pass    none    0       -       temp    temp    0       0       0       b
mem_wr_final

mem_wr_idx:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat  flag    memwri  next_s
# store instruction input in temp_low:
db      -       pass    none    0       -       -       -       0       1       0       0       -
# get memory location of high byte address:
tl      x       add+1   none    0       -       0       r       0       0       0       0       -
# store high byte address in temp_high:
db      -       pass    none    0       -       -       -       1       0       0       0       -
# get memory location of low byte address:
tl      x       add     none    0       -       0       r       0       0       0       0       -
# knowing result is low byte address, get data for writing:
db      func    pass    none    0       -       temp    r       0       0       0       b
mem_wr_final

abs:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    next_s
db      -       pass    none    1       pc_n    pc_n    pc_n    1       0       -
db      -       pass    none    0       -       r       temp    0       0       imm

indirect_x:
# Fetch opcode, then base offset+X, stored in a_lat. Repeat for offset+x+1,
# assert onto address bus, use as high part of next fetch
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat  flag    next_s
# store instruction input in temp_low:
db      -       pass    none    0       -       -       -       0       1       0       -
# get memory location of high byte address:
tl      x       add+1   none    0       -       0       r       0       0       0       -
# store high byte address in temp_high:
db      -       pass    none    0       -       -       -       1       0       0       -
# get memory location of low byte address:
tl      x       add     none    0       -       0       r       0       0       0       -
# knowing result is low byte address, get data from memory:
db      -       pass    none    0       -       temp    r       0       0       0       imm

abs_xy_final:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    next_s
db      -       pass+t  none    0       -       r       temp    0       0       imm

abs_x:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    next_s
db      x       add     none    1       pc_n    pc_n    pc_n    1       t       abs_xy_final

abs_y:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    next_s
db      y       add     none    1       pc_n    pc_n    pc_n    1       t       abs_xy_final

zp_x:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    next_s
db      x       add     none    0       -       0       r       0       0       imm

zp_y:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    next_s
db      y       add     none    0       -       0       r       0       0       imm
```

```
inc:
a_sel   b_sel   alu_op  wrt_en  pcinc   pc_sel  a_h_sel a_l_sel tl_lat  flag
func    func    inc     func    0       -       pc_n    pc_n    0       1

dec:
a_sel   b_sel   alu_op  wrt_en  pcinc   pc_sel  a_h_sel a_l_sel tl_lat  flag
func    func    dec     func    0       -       pc_n    pc_n    0       1

# acc: (takes value in accumulator, sends it through ALU, and puts result in accumulator)
acc:
a_sel   b_sel   alu_op  wrt_en  pcinc   pc_sel  a_h_sel a_l_sel tl_lat  flag
-       a       func    func    0       pc_n    pc_n    pc_n    0       1

mem_rw_final:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    memwri
db      -       -       none    1       pc_n    pc_n    pc_n    0       0       0

mem_rw_zp2:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    memwri  next_s
db      func    func    func    0       -       0       temp    0       1       r       mem_rw_final

mem_rw_zpa:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    memwri  next_s
db      -       pass    none    0       pc_n    0       r       1       0       0       mem_rw_zp2

mem_rw_zpx:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  flag    memwri  next_s
db      x       add     none    0       pc_n    0       r       1       0       0       mem_rw_zp2

mem_rw_ab2:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  th_lat  flag    memwri  next_s
db      func    func    func    0       -       temp    temp    0       0       1       r
mem_rw_final

mem_rw_abs:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  th_lat  flag    memwri  next_s
db      -       pass    none    1       pc_n    pc_n    pc_n    1       0       0       0       -
db      -       pass    none    0       pc_n    r       temp    0       1       0       0
mem_rw_ab2

mem_rw_abx:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel tl_lat  th_lat  flag    memwri  next_s
db      x       add     none    1       pc_n    pc_n    pc_n    1       0       t       0       -
db      -       pass+t  none    0       pc_n    r       temp    0       1       0       0
mem_rw_ab2

indirect_y:
# two cycles longer than on original 6502
# "indirect indexed" addressing: *((*arg)+y)
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat  flag    next_s
# Address of low byte comes in over data line -- save it in temp_high and pass it to address
db      -       pass    none    0       -       0       r       1       0       0       -
# Low byte comes in on data -- pass it and store in temp_low
db      -       pass    none    0       -       -       -       0       1       0       -
# Increment address of low byte to get address of high byte -- pass to address
th      -       inc     none    0       -       0       r       0       0       0       -
# Data from high byte comes in -- save in temp_high
db      -       pass    none    0       -       -       -       1       0       0       -
# Add y to low byte and save back in low byte
tl      y       add     none    0       -       -       -       0       1       t       -
# Add carry to high byte and save back in high byte
th      -       pass+t  none    0       -       -       -       1       0       0       -
# Get data from address stored in temps
db      -       pass    none    0       -       temp    temp    0       0       0       imm

# store and read-modify-write instructions look a lot like these.
# RMW ops must use non-default data read source R.

push_final:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel memwri  pcinc   c_sel
```

```
-       -       -       none    1       pc_n    pc_n    pc_n    0       0       -

# push/pull used only for accumulator; push_flags/pull_flags used for PHP/PLP
push:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel memwri  pcinc   c_sel   next_s
sp      func    dec     sp      0       -       1       r       b       0       1       push_final

pull_final:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel memwri  pcinc   c_sel
sp      -       inc     sp      1       pc_n    pc_n    pc_n    0       0       1

pull:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel memwri  pcinc   c_sel   next_s
sp      -       pass    none    0       -       1       r       0       0       -       -
db      -       pass    func    0       -       -       -       0       0       0       pull_final

push_flags:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel memwri  pcinc   c_sel   next_s
sp      p       dec     sp      0       -       1       r       b       0       1       push_final

pull_flags:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel memwri  pcinc   c_sel   next_s
sp      -       pass    none    0       -       1       r       0       0       -       -
db      -       pass    p       0       -       -       -       0       0       -       pull_final

jsr:
# push high order byte of pc+2, then low order byte, jump to {pcl, pch}.
# one cycle long.
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat  memwri  pcinc
db      -       pass    none    1       pc_n    pc_n    pc_n    0       1       0       1
db      -       pass    none    0       -       -       -       1       0       0       1
sp      pc_h    dec     sp      0       -       1       r       0       0       b       0
sp      pc_l    dec     sp      0       -       1       r       0       0       b       0
tl      -       pass    pc_l    01      r       -       -       0       0       0       0
th      -       pass    pc_h    10      r       temp    temp    0       0       0       0

jmp_abs:
# absolute jump
# can't munge PC until we fetch the high part, so we write the low part back
# on the third cycle
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat
db      -       pass    none    1       pc_n    pc_n    pc_n    0       1
db      -       pass    pc_h    10      r       r       temp    1       0
tl      -       pass    pc_l    01      r       temp    temp    0       0

jmp_ind_final:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat  pcinc
db      -       pass    pc_l    01      r       -       -       0       0       0
tl      -       inc     none    0       -       temp    r       0       0       0
# this is where we'd fix Mr. Peddle's bug.
db      -       pass    pc_h    10      r       r       pc_n    0       0       0

jmp_ind:
# recall bug with page crossing
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel th_lat  tl_lat  pcinc   next_s
db      -       pass    none    1       pc_n    pc_n    pc_n    0       1       1       -
db      -       pass    none    0       -       r       temp    1       0       0       jmp_ind_final

rts:
# pull low, then high of pc, jump to pc+1
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel pcinc
sp      -       pass    none    0       -       1       r       0
db      -       pass    pc_l    01      r       -       -       0
sp      -       inc     sp      0       -       1       r       0
db      -       pass    pc_h    10      r       -       -       0
sp      -       inc     sp      1       pc_n    pc_n    pc_n    1

rti:
# pull p, then pc, jump to pc. similar to above, but don't increment PC
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel pcinc   flag    next_s
sp      -       pass    none    0       -       1       r       0       0       -
```

```
db      -       pass    p       0       -       -       -       0       1       -
sp      -       inc     sp      0       -       1       r       0       0       -
db      -       pass    pc_l    01      r       -       -       0       0       -
sp      -       inc     sp      0       -       1       r       0       0       -
db      -       pass    pc_h    10      r       -       -       0       0       pull_final

break:
a_sel   b_sel   alu_op  wrt_en  pc_w_en pc_sel  a_h_sel a_l_sel memwri  pcinc   tl_lat  th_lat  c_sel
flag    next_s
# set break flag
0x10    -       pass    p       1       -       pc_n    pc_n    0       1       0       0       0
1       -
# once pc has been incremented again, push it onto the stack
sp      pc_h    dec     sp      0       -       1       r       b       0       0       0       1
0       -
sp      pc_l    dec     sp      0       -       1       r       b       0       0       0       1
0       -
# push processor onto the stack
sp      p       dec     sp      0       -       1       r       b       0       0       0       1
0       -
# set interrupt flag
0x14    -       pass    p       1       -       pc_n    pc_n    0       1       0       0       0
1       -
# load and set PC to address stored at $FFFE-$FFFF
0xfe    -       pass    none    0       -       -       -       0       0       1       0       0
0       -
0xff    -       pass    none    0       -       r       temp    0       0       0       1       0
0       jmp_ind_final
```

Microcode: instrtable.txt

```
This table lists all defined opcodes and the opcode-specific control signals
for each.  It gets compiled by
/src/ucode/opcode_translator/instrtable2opcodes.py and
/src/ucode/opcode_translator/opcode_label2bin.py into the system verilog
lines to populate the case structure in the opcode_pla module in control.sv.
the signals are provided in the following order:
        Opcode Instr AddrMd SrcA SrcB Dest ALUOp carry_sel Branch_polarity flags
This file should be viewed with a tab stop of 12 or larger to keep its
structure in a human-readable form.

Note: Bitfield for flags is: [N,V,_,B,D,I,Z,C]
N = Negative Result -- sometimes (S)ign Bit
V = Overflow
_ = Expansion Bit
B = Break Command
D = Decimal Mode
I = Interrupt Disable
Z = Zero Result
C = Carry
```

69	adc	imm	dp	A	A	2	p	0	C3
65	adc	mem_ex_zpa	dp	A	A	2	p	0	C3
75	adc	zp_x	dp	A	A	2	p	0	C3
6D	adc	abs	dp	A	A	2	p	0	C3
7D	adc	abs_x	dp	A	A	2	p	0	C3
79	adc	abs_y	dp	A	A	2	p	0	C3
61	adc	indirect_x	dp	A	A	2	p	0	C3
71	adc	indirect_y	dp	A	A	2	p	0	C3
29	and	imm	dp	A	A	8	p	0	82
25	and	mem_ex_zpa	dp	A	A	8	p	0	82
35	and	zp_x	dp	A	A	8	p	0	82
2D	and	abs	dp	A	A	8	p	0	82
3D	and	abs_x	dp	A	A	8	p	0	82
39	and	abs_y	dp	A	A	8	p	0	82
21	and	indirect_x	dp	A	A	8	p	0	82
31	and	indirect_y	dp	A	A	8	p	0	82
0A	asl	acc	A	_	A	5	p	0	83
06	asl	mem_rw_zpa	dp	_	mem	5	p	0	83
16	asl	mem_rw_zpx	dp	_	mem	5	p	0	83
0E	asl	mem_rw_abs	dp	_	mem	5	p	0	83
1E	asl	mem_rw_abx	dp	_	mem	5	p	0	83
24	bit	mem_ex_zpa	dp	A	_	9	p	0	C2
2C	bit	abs	dp	A	_	9	p	0	C2
C9	cmp	imm	dp	A	_	3	1	0	83
C5	cmp	mem_ex_zpa	dp	A	_	3	1	0	83
D5	cmp	zp_x	dp	A	_	3	1	0	83
CD	cmp	abs	dp	A	_	3	1	0	83
DD	cmp	abs_x	dp	A	_	3	1	0	83
D9	cmp	abs_y	dp	A	_	3	1	0	83
C1	cmp	indirect_x	dp	A	_	3	1	0	83
D1	cmp	indirect_y	dp	A	_	3	1	0	83
E0	cpx	imm	dp	X	_	3	1	0	83
E4	cpx	mem_ex_zpa	dp	X	_	3	1	0	83
EC	cpx	abs	dp	X	_	3	1	0	83
C0	cpy	imm	dp	Y	_	3	1	0	83
C4	cpy	mem_ex_zpa	dp	Y	_	3	1	0	83
CC	cpy	abs	dp	Y	_	3	1	0	83
C6	dec	mem_rw_zpa	dp	_	mem	1	1	0	82
D6	dec	mem_rw_zpx	dp	_	mem	1	1	0	82
CE	dec	mem_rw_abs	dp	_	mem	1	1	0	82
DE	dec	mem_rw_abx	dp	_	mem	1	1	0	82
49	eor	imm	dp	A	A	A	p	0	82
45	eor	mem_ex_zpa	dp	A	A	A	p	0	82
55	eor	zp_x	dp	A	A	A	p	0	82
4D	eor	abs	dp	A	A	A	p	0	82
5D	eor	abs_x	dp	A	A	A	p	0	82

59	eor	abs_y	dp	A	A	A	p	0	82
41	eor	indirect_x	dp	A	A	A	p	0	82
51	eor	indirect_y	dp	A	A	A	p	0	82
E6	inc	mem_rw_zpa	dp	_	mem	0	1	0	82
F6	inc	mem_rw_zpx	dp	_	mem	0	1	0	82
EE	inc	mem_rw_abs	dp	_	mem	0	1	0	82
FE	inc	mem_rw_abx	dp	_	mem	0	1	0	82
4C	jmp	jmp_abs	_	_	_	5	p	0	00
6C	jmp	jmp_ind	_	_	_	5	p	0	00
20	jsr	jsr	_	_	_	5	p	0	00
A9	lda	imm	dp	-	A	0	0	0	82
A5	lda	mem_ex_zpa	dp	-	A	0	0	0	82
B5	lda	zp_x	dp	-	A	0	0	0	82
AD	lda	abs	dp	-	A	0	0	0	82
BD	lda	abs_x	dp	-	A	0	0	0	82
B9	lda	abs_y	dp	-	A	0	0	0	82
A1	lda	indirect_x	dp	-	A	0	0	0	82
B1	lda	indirect_y	dp	-	A	0	0	0	82
A2	ldx	imm	dp	-	X	0	0	0	82
A6	ldx	mem_ex_zpa	dp	-	X	0	0	0	82
B6	ldx	zp_y	dp	-	X	0	0	0	82
AE	ldx	abs	dp	-	X	0	0	0	82
BE	ldx	abs_y	dp	-	X	0	0	0	82
A0	ldy	imm	dp	-	Y	0	0	0	82
A4	ldy	mem_ex_zpa	dp	-	Y	0	0	0	82
B4	ldy	zp_x	dp	-	Y	0	0	0	82
AC	ldy	abs	dp	-	Y	0	0	0	82
BC	ldy	abs_x	dp	-	Y	0	0	0	82
4A	lsr	acc	A	_	A	4	0	0	83
46	lsr	mem_rw_zpa	dp	_	mem	4	0	0	83
56	lsr	mem_rw_zpx	dp	_	mem	4	0	0	83
4E	lsr	mem_rw_abs	dp	_	mem	4	0	0	83
5E	lsr	mem_rw_abx	dp	_	mem	4	0	0	83
09	ora	imm	dp	A	A	7	p	0	82
05	ora	mem_ex_zpa	dp	A	A	7	p	0	82
15	ora	zp_x	dp	A	A	7	p	0	82
0D	ora	abs	dp	A	A	7	p	0	82
1D	ora	abs_x	dp	A	A	7	p	0	82
19	ora	abs_y	dp	A	A	7	p	0	82
01	ora	indirect_x	dp	A	A	7	p	0	82
11	ora	indirect_y	dp	A	A	7	p	0	82
2A	rol	acc	A	_	A	6	p	0	83
26	rol	mem_rw_zpa	dp	_	mem	6	p	0	83
36	rol	mem_rw_zpx	dp	_	mem	6	p	0	83
2E	rol	mem_rw_abs	dp	_	mem	6	p	0	83
3E	rol	mem_rw_abx	dp	_	mem	6	p	0	83

6a	ror	acc	A	_	A	4	p	0	83
66	ror	mem_rw_zpa	dp	_	mem	4	p	0	83
76	ror	mem_rw_zpx	dp	_	mem	4	p	0	83
6e	ror	mem_rw_abs	dp	_	mem	4	p	0	83
7e	ror	mem_rw_abx	dp	_	mem	4	p	0	83
E9	sbc	imm	dp	A	A	3	p	0	C3
E5	sbc	mem_ex_zpa	dp	A	A	3	p	0	C3
F5	sbc	zp_x	dp	A	A	3	p	0	C3
ED	sbc	abs	dp	A	A	3	p	0	C3
FD	sbc	abs_x	dp	A	A	3	p	0	C3
F9	sbc	abs_y	dp	A	A	3	p	0	C3
E1	sbc	indirect_x	dp	A	A	3	p	0	C3
F1	sbc	indirect_y	dp	A	A	3	p	0	C3
85	sta	mem_wr_zpa	dp	A	mem	0	0	0	00
95	sta	mem_wr_zpx	dp	A	mem	0	0	0	00
8D	sta	mem_wr_abs	dp	A	mem	0	0	0	00
9D	sta	mem_wr_abx	dp	A	mem	0	0	0	00
99	sta	mem_wr_aby	dp	A	mem	0	0	0	00
81	sta	mem_wr_idx	dp	A	mem	0	0	0	00
91	sta	mem_wr_idy	dp	A	mem	0	0	0	00
86	stx	mem_wr_zpa	dp	X	mem	0	0	0	00
96	stx	mem_wr_zpy	dp	X	mem	0	0	0	00
8E	stx	mem_wr_abs	dp	X	mem	0	0	0	00
84	sty	mem_wr_zpa	dp	Y	mem	0	0	0	00
94	sty	mem_wr_zpx	dp	Y	mem	0	0	0	00
8C	sty	mem_wr_abs	dp	Y	mem	0	0	0	00
00	brk	break	_	_	_	0	0	0	FF
EA	nop	single_byte	_	_	_	0	p	0	00
18	clc	clear_flag	A	A	_	a	p	0	01
38	sec	set_flag	-	-	_	b	p	0	01
58	cli	clear_flag	A	A	_	a	p	0	04
78	sei	set_flag	-	-	_	b	p	0	04
b8	clv	clear_flag	A	A	_	a	p	0	40
d8	cld	clear_flag	A	A	_	a	p	0	08
f8	sed	set_flag	-	-	_	b	p	0	08
aa	tax	single_byte	A	_	X	0	0	0	82
8a	txa	single_byte	X	_	A	0	0	0	82
ca	dex	dec	X	X	X	1	1	0	82
e8	inx	inc	X	X	X	0	1	0	82
98	tya	single_byte	Y	_	A	0	0	0	82
a8	tay	single_byte	A	_	Y	0	0	0	82
88	dey	dec	Y	Y	Y	1	1	0	82
c8	iny	inc	Y	Y	Y	0	1	0	82

Note: Bitfield for flags is: [N,V,_,B,D,I,Z,C]

10	bpl	branch_head	dp	_	PC	0	p	1	80
30	bmi	branch_head	dp	_	PC	0	p	0	80

50	bvc	branch_head	dp	_	PC	0	p	1	40
70	bvs	branch_head	dp	_	PC	0	p	0	40
90	bcc	branch_head	dp	_	PC	0	p	1	01
B0	bcs	branch_head	dp	_	PC	0	p	0	01
D0	bne	branch_head	dp	_	PC	0	p	1	02
f0	beq	branch_head	dp	_	PC	0	p	0	02
9a	txs	single_byte	X	_	SP	0	0	0	00
ba	tsx	single_byte	SP	_	X	0	0	0	82
48	pha	push	SP	A	SP	1	1	0	00
68	pla	pull	dp	_	A	0	0	0	82
08	php	push_flags	SP	PS	SP	1	p	0	00
60	rts	rts	_	_	_	0	p	0	00
40	rti	rti	_	_	_	0	p	0	FF
28	plp	pull_flags	dp	_	PS	0	p	0	FF

Microcode: generate_plas.bat

```
REM  In Windows, run this batch script from the command prompt
REM    in the directory hmc-6502\src to produce the PLA code.

cd ucode
python ucasm.py > 6502.ucode.compiled
cd opcode_translator
python instrtable2opcodes.py ..\instrtable.txt
python opcode_label2bin.py opcodes.txt
cd ..\..
```

Appendix: ROM Generator

The ROM generator produces a psudo-nMOS NOR ROM with an NOR decoder from a verilog file describing the desired output for each input. The NORs uses transistors of width 4 everywhere. The opcode ROM had 150 entries and thus used an 8-input NOR gate. The pseudo-nMOS NORs uses nMOS transistors of width 4 and weak pMOS transistors of width 3 and length 3. Simulation in hspice of the opcode ROM found a delay of 12ns at 5V and a delay of 150ns at 1.5V. Disturbance from V=0 at the out put of the psudo-nMOS at Vdd=1.5V was 0.03V. The state ROM and opcode ROM together consume an average of 5.2mW of power in the instructions in test suite P.

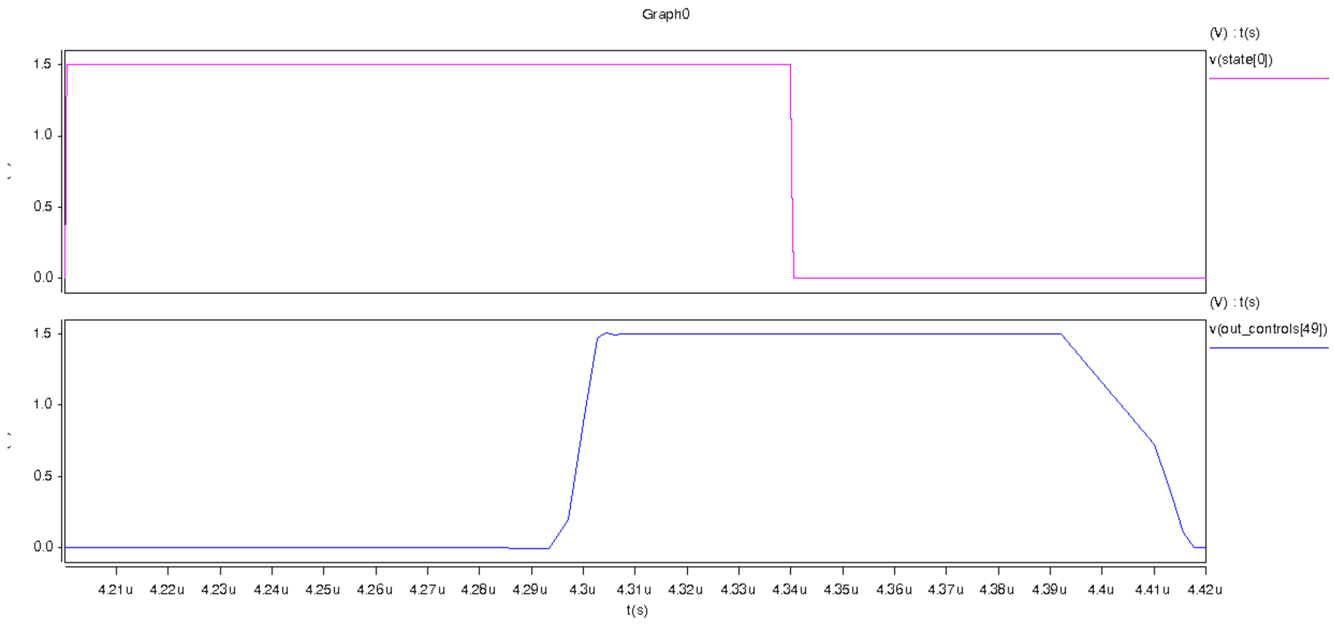


Figure 16 Waveform showing the operation of pseudo-nMOS gates

The verilog input file should contain one valid verilog module with one input bus and one output bus. The module should contain a valid case statement maping input bits to output bits; declarations of the input and output values may be in binary, hex or decimal. "default" statements will be ignored; all outputs for all inputs not explicitly defined in the case statement are 0.

The ROM generator is written in python and requires a working installation of python 2.4 or better to run. The command to run the generator is:

>python romgenerator.py *sourcefile.ext*

The generator will produce *sourcefile.jelib* containing a hierarchical ROM layout with the top level layout and its exports named after the module and inputs/outputs in the sourcefile. The generator will also generate *sourcefile_stim.vec* with a translation of the inputs and outputs of the ROM in to spice stimulus vectors for validation of the behavior.

The generator consists of two source files: layout_gen.py contains classes and helpers for easily generating layouts of any type, romgenerator.py extends this library to generate rom libraries.

ROM Generator: romgenerator.py

```
#!/usr/bin/python
#romgeneratior.py

import re, sys, math
from layout_gen import *

#####################################################################
#Inverters
class InvIN(Cell):
         printed = False
         def __init__(self,name,top,left,rot=''):
                  width = 25
                  height = 102
                  Cell.__init__(self,name,'InvIN','mocmos',top,left,width,height,rot='')

         def exp_loc(self, name):
                  if (name == 'vdd'):
                           return (self.x+8,self.y+76)
                  elif (name == 'gnd'):
                           return (self.x+8,self.y+17)
                  elif (name == 'a'):
                           return (self.x+8,self.y+5)
                  elif (name == 'y'):
                           return (self.x+8,self.y+93.5)
                  elif (name == "yn"):
                           return (self.x+16,self.y+100.5)

         def print_cell(self):
                  if not InvIN.printed:
                           InvIN.printed = True
                           return """
# Cell InvIN{lay}
CInvIN{lay}||mocmos|1163648192318|1205311632465||DRC_last_good_drc_bit()I19|DRC_last_good_drc_date()G120
4701935253
Ngeneric:Facet-Center|art@0||0|0||||AV
NMetal-1-N-Active-Con|contact@0||8|-27||23|Y|
NMetal-1-P-Active-Con|contact@1||8|-71||33|Y|
NMetal-1-Polysilicon-1-Con|contact@16||8|-5||||
NMetal-1-N-Active-Con|contact@17||16|-27||23|Y|
NMetal-1-P-Active-Con|contact@18||16|-71||33|Y|
NMetal-1-Metal-2-Con|contact@19||8|-17||||
NMetal-1-Metal-2-Con|contact@20||8|-76||||
NMetal-1-Polysilicon-1-Con|contact@21||16|-100.5||||
NN-Transistor|nmos@1||12|-27|24||YR|
NPolysilicon-1-Pin|pin@36||8|-11.5||||
NPolysilicon-1-Pin|pin@37||12|-93.5||||
NPolysilicon-1-Pin|pin@38||8|-93.5||||
NP-Well-Node|plnode@0||12|-23|40|46|Y|A
NN-Well-Node|plnode@1||12|-74.5|40|56|Y|A
NP-Transistor|pmos@1||12|-71|34||YR|
APolysilicon-1|net@82|||S2700|pmos@1|p-trans-poly-left|12|-50.5|nmos@1|n-trans-poly-right|12|-42.5
APolysilicon-1|net@93|||S0|nmos@1|n-trans-poly-left|12|-11.5|pin@36||8|-11.5
APolysilicon-1|net@94|||S2700|pin@36||8|-11.5|contact@16||8|-5.5
AMetal-1|net@98||1|S2700|contact@18||16|-55|contact@17||16|-38.5
AMetal-1|net@101||1|S2700|contact@19||8|-17.5|contact@0||8|-17
AP-Active|net@102|||S1800|pmos@1|p-trans-diff-bottom|15.75|-71.5|contact@18||16|-71.5
AP-Active|net@103|||S0|pmos@1|p-trans-diff-top|8.25|-70.5|contact@1||8|-70.5
AN-Active|net@104|||S1800|contact@0||7.5|-31|nmos@1|n-trans-diff-top|8.25|-31
AN-Active|net@105|||S0|contact@17||16|-31.5|nmos@1|n-trans-diff-bottom|15.75|-31.5
AMetal-1|net@106||1|S900|contact@20||8|-76|contact@1||8|-76
APolysilicon-1|net@107|||S900|pmos@1|p-trans-poly-right|12|-91.5|pin@37||12|-93.5
APolysilicon-1|net@108|||S0|pin@37||12|-93.5|pin@38||8|-93.5
AMetal-1|net@109||1|S2700|contact@21||16|-100|contact@18||16|-87.5
```

```
Ea||D5G2;|contact@16||I
Egnd||D5G2;|contact@19||G
Evdd||D5G2;|contact@20||P
Ey||D5G2;|pin@38||O
Eyn||D5G2;|contact@21||O
X
"""
                    return ""
class InvOD(Cell):
          printed = False
          def __init__(self,name,top,left,rot=''):
                    width = 33
                    height = 102
                    Cell.__init__(self,name,'InvOD','mocmos',top,left,width,height,rot='')

          def exp_loc(self, name):
                    if (name == 'vdd'):
                              return (self.x+8,self.y+66.5)
                    elif (name == 'gnd'):
                              return (self.x+8,self.y+2)
                    elif (name == 'a_0'):
                              return (self.x,self.y)
                    elif (name == 'a_1'):
                              return (self.x+16,self.y)
                    elif (name == 'y_0'):
                              return (self.x,self.y+81.5)
                    elif (name == 'y_1'):
                              return (self.x+16,self.y+81.5)

          def print_cell(self):
                    if not InvOD.printed:
                              InvOD.printed = True
                              return """
# Cell InvOD;1{lay}
CInvOD{lay}||mocmos|1163648192318|1205171564387||DRC_last_good_drc_bit()I19|DRC_last_good_drc_date()G120
4701935253
Ngeneric:Facet-Center|art@0||0|0|||AV
NMetal-1-N-Active-Con|contact@0||8|-22.5||23|Y|
NMetal-1-P-Active-Con|contact@1||8|-66.5||33|Y|
NMetal-1-N-Active-Con|contact@3||16|-22.5||23|Y|
NMetal-1-P-Active-Con|contact@4||16|-66.5||33|Y|
NMetal-1-Metal-2-Con|contact@11||8|-66.5||||
NMetal-1-Metal-2-Con|contact@12||16|-81.5||||
NMetal-1-Polysilicon-1-Con|contact@14||16|0||||
NMetal-1-Metal-2-Con|contact@15||0|-81.5||||
NMetal-1-Polysilicon-1-Con|contact@16||0|0||||
NMetal-1-N-Active-Con|contact@17||0|-22.5||23|Y|
NMetal-1-P-Active-Con|contact@18||0|-66.5||33|Y|
NN-Transistor|nmos@0||12|-22.5|24||YR|
NN-Transistor|nmos@1||4|-22.5|24||YR|
NPolysilicon-1-Pin|pin@28||16|0.5||||
NPolysilicon-1-Pin|pin@29||16|-7||||
NMetal-1-P-Well-Con|pin@30||8|-2||||
NPolysilicon-1-Pin|pin@36||0|-7||||
NP-Well-Node|plnode@0||8|-19|42|46|Y|A
NN-Well-Node|plnode@1||8|-70|42|56|Y|A
NP-Transistor|pmos@0||12|-66.5|34||YR|
NP-Transistor|pmos@1||4|-66.5|34||YR|
AN-Active|net@9|||S1800|contact@0||8|-22.5|nmos@0|n-trans-diff-top|8|-22.5
AN-Active|net@10|||S0|contact@3||16|-22.5|nmos@0|n-trans-diff-bottom|16|-22.5
AP-Active|net@14|||S1800|contact@1||8|-79|pmos@0|p-trans-diff-top|8|-79
AP-Active|net@15|||S0|contact@4||16.5|-66.5|pmos@0|p-trans-diff-bottom|16|-66.5
APolysilicon-1|net@65|||S2700|pmos@0|p-trans-poly-left|12|-46|nmos@0|n-trans-poly-right|12|-38
AMetal-1|net@66||1|S0|contact@11||8|-66.5|contact@1||8|-66.5
APolysilicon-1|net@69|||S900|pin@28||16|0.5|pin@29||16|-7
APolysilicon-1|net@70|||S0|pin@29||16|-7|nmos@0|n-trans-poly-left|12|-7
APolysilicon-1|net@71|||S0|contact@14||16|0.5|pin@28||16|0.5
AMetal-1|net@74||1|S900|pin@30||8|-2|contact@0||8|-11
APolysilicon-1|net@82|||S2700|pmos@1|p-trans-poly-left|4|-46|nmos@1|n-trans-poly-right|4|-38
AMetal-1|net@88||1|S2700|contact@4||16|-50|contact@3||16|-34
AMetal-1|net@90||1|S900|contact@12||16|-81.5|contact@4||16|-81.5
```

```
AN-Active|net@91|||S1800|nmos@1|n-trans-diff-bottom|7.5|-22.5|nmos@0|n-trans-diff-top|8|-22.5
AP-Active|net@92|||S1800|pmos@1|p-trans-diff-bottom|7.5|-60|pmos@0|p-trans-diff-top|8|-60
APolysilicon-1|net@93|||S0|nmos@1|n-trans-poly-left|4|-7|pin@36||0|-7
APolysilicon-1|net@94|||S2700|pin@36||0|-7|contact@16||0|0.5
AN-Active|net@95|||S1800|contact@17||-0.5|-22.5|nmos@1|n-trans-diff-top|0|-22.5
AP-Active|net@96|||S1800|contact@18||-0.5|-66.5|pmos@1|p-trans-diff-top|0|-66.5
AMetal-1|net@98||1|S2700|contact@18||0|-50|contact@17||0|-34
AMetal-1|net@99||1|S900|contact@15||0|-81.5|contact@18||0|-81.5
Ea_0||D5G2;|contact@16||I
Ea_1||D5G2;|contact@14||I
Egnd||D5G2;|pin@30||I
Evdd||D5G2;|contact@11||P
Ey_0||D5G2;|contact@15||O
Ey_1||D5G2;|contact@12||O
X"""
                   return ""

class InvOS(Cell):
         printed = False
         def __init__(self,name,top,left,rot=''):
                   width = 25
                   height = 102
                   Cell.__init__(self,name,'InvOS','mocmos',top,left,width,height,rot='')

         def exp_loc(self, name):
                   if (name == 'vdd'):
                             return (self.x+8,self.y+66.5)
                   elif (name == 'gnd'):
                             return (self.x+8,self.y+2)
                   elif (name == 'a_0'):
                             return (self.x,self.y)
                   elif (name == 'y_0'):
                             return (self.x,self.y+81.5)

         def print_cell(self):
                   if not InvOS.printed:
                             InvOS.printed = True
                             return """
# Cell InvOS;1{lay}
CInvOS{lay}||mocmos|1163648192318|1205171750944||DRC_last_good_drc_bit()I19|DRC_last_good_drc_date()G120
4701935253
Ngeneric:Facet-Center|art@0||0|0||||AV
NMetal-1-N-Active-Con|contact@0||8|-22.5||23|Y|
NMetal-1-P-Active-Con|contact@1||8|-66.5||33|Y|
NMetal-1-Metal-2-Con|contact@11||8|-66.5||||
NMetal-1-Metal-2-Con|contact@15||0|-81.5||||
NMetal-1-Polysilicon-1-Con|contact@16||0|0||||
NMetal-1-N-Active-Con|contact@17||0|-22.5||23|Y|
NMetal-1-P-Active-Con|contact@18||0|-66.5||33|Y|
NN-Transistor|nmos@1||4|-22.5|24||YR|
NMetal-1-P-Well-Con|pin@30||8|-2||||
NPolysilicon-1-Pin|pin@36||0|-7||||
NP-Well-Node|plnode@0||4|-19|24|46|Y|A
NN-Well-Node|plnode@1||4|-70|24|56|Y|A
NP-Transistor|pmos@1||4|-66.5|34||YR|
APolysilicon-1|net@82|||S2700|pmos@1|p-trans-poly-left|4|-46|nmos@1|n-trans-poly-right|4|-38
APolysilicon-1|net@93|||S0|nmos@1|n-trans-poly-left|4|-7|pin@36||0|-7
APolysilicon-1|net@94|||S2700|pin@36||0|-7|contact@16||0|0.5
AN-Active|net@95|||S1800|contact@17||0.25|-22.5|nmos@1|n-trans-diff-top|0.25|-22.5
AP-Active|net@96|||S1800|contact@18||0.25|-66.5|pmos@1|p-trans-diff-top|0.25|-66.5
AMetal-1|net@98||1|S2700|contact@18||0|-66.5|contact@17||0|-22.5
AMetal-1|net@99||1|S2700|contact@15||0|-81.5|contact@18||0|-66.5
AMetal-1|net@100||1|S900|pin@30||8|-2|contact@0||8|-22.5
AN-Active|net@102|||S1800|nmos@1|n-trans-diff-bottom|7.75|-22.5|contact@0||7.75|-22.5
AP-Active|net@104|||S1800|pmos@1|p-trans-diff-bottom|7.75|-66.5|contact@1||7.75|-66.5
AMetal-1|net@105||1|S0|contact@11||8|-66.5|contact@1||8|-66.5
Ea_0||D5G2;|contact@16||I
Egnd||D5G2;|pin@30||I
Evdd||D5G2;|contact@11||P
Ey_0||D5G2;|contact@15||O
X
```

```
"""
                    return ""
#################################################################################
#Pulllups
class PullUpD(Cell):
          printed = False
          def __init__(self,name,top,left,rot=''):
                    width = 32
                    height = 40
                    Cell.__init__(self,name,'PullUpD','mocmos',top,left,width,height,rot='')

          def exp_loc(self, name):
                    if (name == 'gnd_l'):
                              return (self.x+24,self.y+30)
                    elif (name == 'gnd_r'):
                              return (self.x+8,self.y+30)
                    elif (name == 'gnd_out'):
                              return (self.x+16,self.y+38)
                    elif (name == 'vdd_l'):
                              return (self.x+24,self.y+13)
                    elif (name == 'vdd_r'):
                              return (self.x+8,self.y+13)
                    elif (name == 'poly_l'):
                              return (self.x+27.5,self.y+17.5)
                    elif (name == 'poly_r'):
                              return (self.x+4.5,self.y+17.5)
                    elif (name == 'y0'):
                              return (self.x+8,self.y+22)
                    elif (name == 'y1'):
                              return (self.x+24,self.y+22)

          def print_cell(self):
                    if not PullUpD.printed:
                              PullUpD.printed = True
                              return """
# Cell PullUpD{lay}
CPullUpD{lay}||mocmos|1205174373733|1205176323729|
Ngeneric:Facet-Center|art@0||0|0||||AV
NMetal-1-P-Active-Con|contact@0||24|-13||||
NMetal-1-P-Active-Con|contact@1||24|-22||||
NMetal-1-P-Active-Con|contact@2||8|-13||||
NMetal-1-P-Active-Con|contact@3||8|-22||||
NMetal-1-Metal-2-Con|contact@4||8|-13||||
NMetal-1-Metal-2-Con|contact@5||24|-13||||
NMetal-1-Metal-2-Con|contact@6||24|-22||||
NMetal-1-Metal-2-Con|contact@7||8|-22||||
NMetal-1-Pin|pin@6||8|-30||||
NMetal-1-Pin|pin@7||16|-30||||
NMetal-1-Pin|pin@8||24|-30||||
NMetal-1-Pin|pin@9||16|-38||||
NN-Well-Node|plnode@0||16|-16|32|32||A
NP-Select-Node|plnode@1||24|-18|8|18||
NP-Select-Node|plnode@2||8|-17.5|8|18||
NP-Transistor|pmos@0||24|-17.5||1|||SIM_weak_node(D5G1;)SWeak
NP-Transistor|pmos@1||8|-17.5||1|||SIM_weak_node(D5G1;)SWeak
NMetal-1-N-Well-Con|substr@0||24|-5||||
AP-Active|net@0|||S900|contact@0||24|-13|pmos@0|p-trans-diff-top|24|-13.25
AP-Active|net@1|||S900|pmos@0|p-trans-diff-bottom|24|-21.75|contact@1||24|-22
AMetal-1|net@2||1|S900|substr@0||24|-5|contact@0||24|-13
AP-Active|net@3|||S900|contact@2||8|-13|pmos@1|p-trans-diff-top|8|-13.25
AP-Active|net@4|||S900|pmos@1|p-trans-diff-bottom|8|-21.75|contact@3||8|-22
APolysilicon-1|net@6||1|S0|pmos@0|p-trans-poly-left|20.5|-17.5|pmos@1|p-trans-poly-right|11.5|-17.5
AMetal-1|net@7||1|S900|contact@5||24|-13|contact@0||24|-13
AMetal-1|net@8||1|S2700|contact@2||8|-13|contact@4||8|-13
AMetal-2|net@9||1|S1800|contact@4||8|-13|contact@5||24|-13
AMetal-1|net@14||1|S1800|pin@6||8|-30|pin@7||16|-30
AMetal-1|net@15||1|S1800|pin@7||16|-30|pin@8||24|-30
AMetal-1|net@16||1|S900|pin@7||16|-30|pin@9||16|-38
AMetal-1|net@17||1|S2700|contact@1||24|-22|contact@6||24|-22
AMetal-1|net@18||1|S900|contact@3||8|-22|contact@7||8|-22
Egnd_l||D5G2;|pin@8||G
```

```
Egnd_out||D5G2;|pin@9||G
Egnd_r||D5G2;|pin@6||G
Epoly_l||D5G2;|pmos@0|p-trans-poly-right|O
Epoly_r||D5G2;|pmos@1|p-trans-poly-left|I
Evdd_l||D5G2;|contact@5||P
Evdd_r||D5G2;|contact@4||P
Ey0||D5G2;|contact@7||O
Ey1||D5G2;|contact@6||O
X

# Cell PullUpD;1{sch}
CPullUpD;1{sch}||schematic|1207025538585|1207025538587|
Ngeneric:Facet-Center|art@0||0|0||||AV
NWire_Pin|pin@0||24|-13||||
NWire_Pin|pin@1||24|-22||||
NWire_Pin|pin@2||8|-13||||
NWire_Pin|pin@3||8|-22||||
NWire_Pin|pin@4||8|-13||||
NWire_Pin|pin@5||24|-13||||
NWire_Pin|pin@6||24|-22||||
NWire_Pin|pin@7||8|-22||||
NWire_Pin|pin@8||8|-30||||
NWire_Pin|pin@9||16|-30||||
NWire_Pin|pin@10||24|-30||||
NWire_Pin|pin@11||16|-38||||
NWire_Pin|pin@12||16|-16||||
NWire_Pin|pin@13||24|-18||||
NWire_Pin|pin@14||8|-17.5||||
NWire_Pin|pin@15||24|-5||||
NTransistor|pmos@0||24|-17.5|||RRR|2|ATTR_length(D5G0.5;X-0.5;Y-1;)D3.0|ATTR_width(D5G1;X0.5;Y-
1;)D3.0|SIM_weak_node(D5G1;)SWeak
NTransistor|pmos@1||8|-17.5|||RRR|2|ATTR_length(D5G0.5;X-0.5;Y-1;)D3.0|ATTR_width(D5G1;X0.5;Y-
1;)D3.0|SIM_weak_node(D5G1;)SWeak
Awire|net@0|||F513|pin@0||24|-13|pmos@0|s|22|-15.5
Awire|net@1|||F1287|pmos@0|d|22|-19.5|pin@1||24|-22
Awire|net@2|||900|pin@15||24|-5|pin@0||24|-13
Awire|net@3|||F513|pin@2||8|-13|pmos@1|s|6|-15.5
Awire|net@4|||F1287|pmos@1|d|6|-19.5|pin@3||8|-22
Awire|net@6|||0|pmos@0|g|25|-17.5|pmos@1|g|9|-17.5
Awire|net@7|||F0|pin@5||24|-13|pin@0||24|-13
Awire|net@8|||F0|pin@2||8|-13|pin@4||8|-13
Awire|net@9|||1800|pin@4||8|-13|pin@5||24|-13
Awire|net@14|||1800|pin@8||8|-30|pin@9||16|-30
Awire|net@15|||1800|pin@9||16|-30|pin@10||24|-30
Awire|net@16|||900|pin@9||16|-30|pin@11||16|-38
Awire|net@17|||F0|pin@1||24|-22|pin@6||24|-22
Awire|net@18|||F0|pin@3||8|-22|pin@7||8|-22
Egnd_l||D5G2;|pin@10||G
Egnd_out||D5G2;|pin@11||G
Egnd_r||D5G2;|pin@8||G
Epoly_l||D5G2;|pmos@0|g|O
Epoly_r||D5G2;|pmos@1|g|I
Evdd_l||D5G2;|pin@5||P
Evdd_r||D5G2;|pin@4||P
Ey0||D5G2;|pin@7||O
Ey1||D5G2;|pin@6||O
X"""
                   return ""

class PullUpS(Cell):
          printed = False
          def __init__(self,name,top,left,rot=''):
                   width = 18
                   height = 40
                   Cell.__init__(self,name,'PullUpS','mocmos',top,left,width,height,rot='')

          def exp_loc(self, name):
                   if (name == 'gnd_r'):
                            return (self.x+8,self.y+30)
                   elif (name == 'gnd_out'):
                            return (self.x+16,self.y+38)
```

```
                elif (name == 'vdd_r'):
                        return (self.x+8,self.y+13)
                elif (name == 'poly_r'):
                        return (self.x+4.5,self.y+17.5)
                elif (name == 'y0'):
                        return (self.x+8,self.y+22)

        def print_cell(self):
                if not PullUpS.printed:
                        PullUpS.printed = True
                        return """
# Cell PullUpS{lay}
CPullUpS{lay}||mocmos|1205174373733|1205176382543|
NMetal-1-P-Active-Con|contact@2||8|-13||||
NMetal-1-P-Active-Con|contact@3||8|-22||||
NMetal-1-Metal-2-Con|contact@4||8|-13||||
NMetal-1-Metal-2-Con|contact@7||8|-22||||
NMetal-1-Pin|pin@6||8|-30||||
NMetal-1-Pin|pin@7||16|-30||||
NMetal-1-Pin|pin@9||16|-38||||
NP-Select-Node|plnode@2||8|-17.5|8|18||
NN-Well-Node|plnode@0||8|-16|16|32||A
NP-Transistor|pmos@1||8|-17.5||1|||SIM_weak_node(D5G1;)SWeak
AP-Active|net@3|||S900|contact@2||8|-13|pmos@1|p-trans-diff-top|8|-13
AP-Active|net@4|||S900|pmos@1|p-trans-diff-bottom|8|-22|contact@3||8|-22
AMetal-1|net@8||1|S2700|contact@2||8|-13|contact@4||8|-13
AMetal-1|net@14||1|S1800|pin@6||8|-30|pin@7||16|-30
AMetal-1|net@16||1|S900|pin@7||16|-30|pin@9||16|-38
AMetal-1|net@18||1|S900|contact@3||8|-22|contact@7||8|-22
Egnd_out||D5G2;|pin@9||G
Egnd_r||D5G2;|pin@6||G
Epoly_r||D5G2;|pmos@1|p-trans-poly-left|I
Evdd_r||D5G2;|contact@4||P
Ey0||D5G2;|contact@7||O
X

# Cell PullUpS{sch}
CPullUpS{sch}||schematic|1207188612233|1207188620883|
Ngeneric:Facet-Center|art@0||0|0||||AV
NWire_Pin|pin@0||8|-13||||
NWire_Pin|pin@1||8|-22||||
NWire_Pin|pin@2||8|-13||||
NWire_Pin|pin@3||8|-22||||
NWire_Pin|pin@4||8|-30||||
NWire_Pin|pin@5||16|-30||||
NWire_Pin|pin@6||16|-38||||
NTransistor|pmos@0||8|-17.5|||RRR|2|ATTR_length(D5G0.5;X-0.5;Y-1;)D3.0|ATTR_width(D5G1;X0.5;Y-
1;)D3.0|SIM_weak_node(D5G1;)SWeak
Awire|net@3|||F513|pin@0||8|-13|pmos@0|s|6|-15.5
Awire|net@4|||F1287|pmos@0|d|6|-19.5|pin@1||8|-22
Awire|net@8|||F0|pin@0||8|-13|pin@2||8|-13
Awire|net@14|||1800|pin@4||8|-30|pin@5||16|-30
Awire|net@16|||900|pin@5||16|-30|pin@6||16|-38
Awire|net@18|||F0|pin@1||8|-22|pin@3||8|-22
Egnd_out||D5G2;|pin@6||G
Egnd_r||D5G2;|pin@4||G
Epoly_r||D5G2;|pmos@0|g|I
Evdd_r||D5G2;|pin@2||P
Ey0||D5G2;|pin@3||O
X"""
                return ""


######################################################################
#ROM printer
def r_p(row): #helper for placing data/gnd pins (row)
        return row*8

def d_t(row): #helper for placing data transistors (row)
        return row*8+4
```

```
def d_c(col): #helper for placing data pins/trans (col)
         return (col+(col+1)/2)*8

def g_c(col): #helper for placing gnd pins (col)
         return (col*3+1)*8

class Encoder(Cell):
         printed = False
         def __init__(self,name,top,left,rows,cols,data,rot=''):
                  self.width = cols*12
                  self.height = rows*8
                  Cell.__init__(self,name,'Encoder','mocmos',top,left,self.width,self.height,rot='')
                  self.add_nodes(rows, cols, data)
                  self.add_arcs(rows, cols, data)
                  self.add_exports(rows,cols)

         def add_nodes(self,rows,cols,data):
                  for j in range(0,cols):
                           for i in range(0,rows+1):
                                    #metal layer
                                    s = "m1_aN" if (i%2==0) else "m1"
                                    self.addpin('dcon_'+und(i,j),[s,r_p(i),d_c(j)])
                                    #transitors
                                    if (i<rows):
                                             if(data[i][1][j]=="1"):
                                                      self.addmos('tran_'+und(i,j),["n-
trans",d_t(i),d_c(j),1,0,""])
                                             else:

         self.addpin('tran_'+und(i,j),["poly",d_t(i),d_c(j)])
                  #gnd rows nodes
                  for j in range(0,cols%2+cols/2):
                           for i in range(0,rows+1):
                                    #metal layer
                                    s = "m1_aN" if (i%2 != 0) else "m1"
                                    self.addpin('gcon_'+und(i,j),[s,r_p(i),g_c(j)])
                  #row input pins
                  for i in range(0,rows):
                           self.addpin('tran_'+und(i,-1),['m1_p',d_t(i),d_c(-1)])
                  #col input pins
                  for j in range(0,cols):
                           self.addpin('din_'+str(j),['m2_m1',r_p(0),d_c(j)])
                  #n-select
                  self.addpin('n-select',['N-Select-Node',\
                                                                  self.height/2.0-
8,self.width/2.0,self.width+24,self.height+24])
                  self.addpin('p-well',['P-Well-Node',\

         self.height/2.0,self.width/2.0,self.width+24,self.height+24])

         def add_arcs(self,rows,cols,data):
                  #col input arcs
                  for j in range(0,cols):
                           self.addarc("din_"+str(j), \
                                                      ["Metal-1",
self.nref("din_"+str(j)),self.nref("dcon_"+und(0,j))])
                  #data row arcs
                  for j in range(0,cols):
                           for i in range(0,rows):
                                    #metal layer
                                    self.addarc("netm1"+und(i,j), \
                                                      ["Metal-1",
self.nref("dcon_"+und(i,j)),self.nref("dcon_"+und(i+1,j))])
                                    #transitors
                                    if (i<rows and data[i][1][j]=="1"):
                                             if(i%2==0):
                                                      self.addarc("tran_"+und(i,j)+"_unet", \
                                                      ["N-Active",
self.nref("tran_"+und(i,j)),self.nref("dcon_"+und(i,j)),'top'])
                                                      self.addarc("tran_"+und(i,j)+"_bnet", \
                                                        ["N-Active",
self.nref("tran_"+und(i,j)),self.nref("gcon_"+und(i+1,j/2)),'bottom'])
                                                else:
                                                        self.addarc("tran_"+und(i,j)+"_unet", \
                                                        ["N-Active",
self.nref("tran_"+und(i,j)),self.nref("gcon_"+und(i,j/2)),'top'])
                                                        self.addarc("tran_"+und(i,j)+"_bnet", \
                                                        ["N-Active",
self.nref("tran_"+und(i,j)),self.nref("dcon_"+und(i+1,j)),'bottom'])
                                        if(i<rows):
                                                self.addarc("tran_"+und(i,j)+"_lnet", \
                                                ["Polysilicon-1",
self.nref("tran_"+und(i,j)),self.nref("tran_"+und(i,j-1)),'left','right'])
                #gnd metal arcs
                for j in range(0,cols%2+cols/2):
                        for i in range(0,rows):
                                #metal layer
                                self.addarc("netgnd1"+und(i,j), \
                                        ["Metal-1",
self.nref("gcon_"+und(i,j)),self.nref("gcon_"+und(i+1,j))])

        def add_exports(self,rows,cols):
                for j in range(0, cols):
                        if (j%2 == 0):
                                self.addexport("gin_"+str(j/2), ['G',
self.nref("gcon_"+und(0,j/2))])
                                self.addexport("gout_"+str(j/2), ['G',
self.nref("gcon_"+und(rows,j/2))])
                        self.addexport("din_"+str(j), ['I', self.nref("din_"+str(j))])
                        self.addexport("dout_"+str(j), ['O', self.nref("dcon_"+und(rows,j))])
                for i in range(0, rows):
                        self.addexport("word_"+str(i), ['I', self.nref("tran_"+und(i,-1)),'left'])

##############################################################################
# DecoderN cell
class DecoderN(Cell):
        "A basic nor decoder, n half"
        def __init__(self,name,top,left,cols,rows,strs,rot=''):
                width = cols*16
                height = rows*8
                Cell.__init__(self,name,'DecoderN','mocmos',top,left,width,height,rot='')
                self.add_nodes(rows, cols,strs)
                self.add_arcs(rows, cols,strs)
                self.add_exports(rows,cols)

        def add_nodes(self,rows, cols,strs):
                for j in range(0,rows):
                        for i in range(0,cols):
                                #transistors
                                if (strs[j][0][i] == '0'):
                                        self.addmos('tran1_'+und(i,j),["n-
trans",j*8,i*16+4,1,0,"R"])
                                        self.addpin('m1_a_'+und(i,j),["m1_aN",j*8,i*16,0,0])
                                        self.addpin('m2_'+und(i,j),["m2_m1",j*8,i*16,0,0])
                                        self.addpin('m1_ga_'+und(i,j),["m1_aN",j*8,i*16+8,0,0])
                                elif(strs[j][0][i] == '1'):
                                        self.addmos('tran0_'+und(i,j),["n-
trans",j*8,i*16+12,1,0,"R"])

        self.addpin('m1_a_'+und(i+1,j),["m1_aN",j*8,i*16+16,0,0])
                                        self.addpin('m2_'+und(i+1,j),["m2_m1",j*8,i*16+16,0,0])
                                        self.addpin('m1_ga_'+und(i,j),["m1_aN",j*8,i*16+8,0,0])
                for j in range(0,rows):
                        for i in range(0,cols):
                                self.addpin('m2_'+und(i,j),["m2",j*8,i*16,0,0])
                                self.addpin('m1_ga_'+und(i,j),["m1",j*8,i*16+8,0,0])
                                self.addpin('tran1_'+und(i,j),["poly",j*8,i*16+4,0,0])
                                self.addpin('tran0_'+und(i,j),["poly",j*8,i*16+12,0,0])
                        self.addpin('m2_'+und(cols,j),["m2_m1",j*8,cols*16,0,0])

        def add_arcs(self,rows, cols,strs):
```

```
			for j in range(0,rows):
				for i in range(0,cols):
					#pin transistors
					if (strs[j][0][i] == '0'):
						self.addarc("t1_g_"+und(i,j),["N-Active",
self.nref('tran1_'+und(i,j)), \

						self.addarc("t1_m1_"+und(i,j),["N-Active",
self.nref('tran1_'+und(i,j)), \

						self.addarc("t1_m2_"+und(i,j),["Metal-1",
self.nref('m2_'+und(i,j)), \

					if(strs[j][0][i] == '1'):
						self.addarc("t0_g_"+und(i,j),["N-Active",
self.nref('tran0_'+und(i,j)), \

						self.addarc("t0_m1_"+und(i,j),["N-Active",
self.nref('tran0_'+und(i,j)), \

						self.addarc("t0_m2_"+und(i,j),["Metal-1",
self.nref('m2_'+und(i+1,j)), \

					#wordlines
					self.addarc("arc_m2_"+und(i,j), \
						["Metal-2",
self.nref('m2_'+und(i,j)),self.nref('m2_'+und(i+1,j))])
					if (j<(rows-1)):
						#input lines
						self.addarc("arc_p1_"+und(i,j), \
						["Polysilicon-1", self.nref('tran1_'+und(i,j)), \

	self.nref('tran1_'+und(i,j+1)),'left','right'])
						self.addarc("arc_p0_"+und(i,j), \
						["Polysilicon-1", self.nref('tran0_'+und(i,j)), \

	self.nref('tran0_'+und(i,j+1)),'left','right'])
						#ground lines
						self.addarc("m1_ga_"+und(i,j), \
							["Metal-1", self.nref('m1_ga_'+und(i,j)), \
								self.nref('m1_ga_'+und(i,j+1))])

	def add_exports(self, rows, cols):
		for j in range(0,rows):
			#wordlines in and out
			self.addexport("wordin_"+str(j), ["I", self.nref('m2_'+und(0,j))])
			self.addexport("wordout_"+str(j), ["O", self.nref('m2_'+und(cols,j))])
		for i in range(0,cols):
			#inputs
			self.addexport("an_"+str(i), ["I", self.nref('tran0_'+und(i,0)),'right'])
			self.addexport("a_"+str(i), ["I", self.nref('tran1_'+und(i,0)),'right'])
			#gnd
			self.addpin('gnd_'+str(i),["m1",-8,i*16+8])
			self.addarc("gnd_"+str(i), \
					["Metal-1", self.nref('gnd_'+str(i)), \
						self.nref('m1_ga_'+und(i,0))])
			self.addexport("gnd_"+str(i), ["G", self.nref('gnd_'+str(i))])

##############################################################################
# DecoderP cell
class DecoderP(Cell):
	"A basic nor decoder, p half"
	def __init__(self,name,top,left,cols,rows,strs,rot=''):
		width = cols*16
		height = rows*8
		Cell.__init__(self,name,'DecoderP','mocmos',top,left,width,height,rot='')
		self.add_nodes(rows, cols,strs)
		self.add_arcs(rows, cols, strs)
		self.add_exports(rows,cols)

	def add_nodes(self,rows, cols, strs):
		for j in range(0,rows):
			for i in range(0,cols):
				#transistors
				if (strs[j][0][i] == '0'):
					self.addmos('tran1_'+und(i,j),["p-
trans",j*8,i*16+4,1,0,"R"])
					self.addpin('m1_'+und(i,j),["m1_aP",j*8,i*16,0,0])
					self.addpin('m1_a_'+und(i,j),["m1_aP",j*8,i*16+8,0,0])
				elif(strs[j][0][i] == '1'):
					self.addmos('tran0_'+und(i,j),["p-
trans",j*8,i*16+12,1,0,"R"])
					self.addpin('m1_'+und(i+1,j),["m1_aP",j*8,i*16+16,0,0])
					self.addpin('m1_a_'+und(i,j),["m1_aP",j*8,i*16+8,0,0])
		for j in range(0,rows):
			self.addpin('m1_'+und(0,j),["m1_Nw",j*8,0,0,0])
			for i in range(0,cols):
				self.addpin('m1_'+und(i,j),["m1",j*8,i*16,0,0])
				self.addpin('m1_a_'+und(i,j),["m1",j*8,i*16+8,0,0])
				self.addpin('tran1_'+und(i,j),["poly",j*8,i*16+4,0,0])
				self.addpin('tran0_'+und(i,j),["poly",j*8,i*16+12,0,0])
			self.addpin('m1_'+und(cols,j),["m1",j*8,cols*16,0,0])

	def add_arcs(self,rows, cols, strs):
		for j in range(0,rows):
			for i in range(0,cols):
				#pin transistors
				if (strs[j][0][i] == '0'):
					self.addarc("t1_a_"+und(i,j),["P-Active",
self.nref('tran1_'+und(i,j)), \

					self.addarc("t1_m1_"+und(i,j),["P-Active",
self.nref('tran1_'+und(i,j)), \

				else:
					self.addarc("m1_1_"+und(i,j),["Metal-1",
self.nref('m1_'+und(i,j)), \

				if(strs[j][0][i] == '1'):
					self.addarc("t0_a_"+und(i,j),["P-Active",
self.nref('tran0_'+und(i,j)), \

					self.addarc("t0_m1_"+und(i,j),["P-Active",
self.nref('tran0_'+und(i,j)), \

				else:
					self.addarc("m1_0_"+und(i,j),["Metal-1",
self.nref('m1_'+und(i+1,j)), \

				#wordlines
				if (j<(rows-1)):
					#input lines
					self.addarc("arc_p1_"+und(i,j), \
					["Polysilicon-1", self.nref('tran1_'+und(i,j)), \

	self.nref('tran1_'+und(i,j+1)),'left','right'])
					self.addarc("arc_p0_"+und(i,j), \
					["Polysilicon-1", self.nref('tran0_'+und(i,j)), \

	self.nref('tran0_'+und(i,j+1)),'left','right'])

	def add_exports(self, rows, cols):
		for j in range(0,rows):
			#wordlines out
			self.addpin('wordout_'+str(j),["m2_m1",j*8,(cols)*16])
			self.addarc("wordout_"+str(j), \
					["Metal-1", self.nref('wordout_'+str(j)), \
						self.nref('m1_'+und(cols,j))]])
			self.addexport("wordout_"+str(j), ["O", self.nref('wordout_'+str(j))]])
			#vdd
			self.addpin('vdd_'+str(j),["m2_m1",j*8,0])
			self.addarc("vdd_"+str(j), \
                                                ["Metal-1", self.nref('vdd_'+str(j)), \
                                                        self.nref('m1_'+und(0,j))])
                                self.addexport("vdd_"+str(j), ["O", self.nref('vdd_'+str(j))])
                        for i in range(0,cols):
                                #a and a'
                                self.addexport("an_"+str(i), ["I", self.nref('tran0_'+und(i,0)),'right'])
                                self.addexport("a_"+str(i), ["I", self.nref('tran1_'+und(i,0)),'right'])


########################################################################################
#Output inverter bank
class InvOut(Cell):
        def __init__(self,name,top,left,cols,rot=''):
                width = 24*(cols/2)+16*(cols%2)+9
                height = 102
                Cell.__init__(self,name,'InvOut','mocmos',top,left,width,height,rot='')
                self.add_nodes(cols)
                self.add_arcs(cols)
                self.add_exports(cols)

        def add_nodes(self, cols):
                for i in range(0,cols/2):
                        self.addnode("inv_"+str(i),[0,24*i],InvOD)
                if (cols%2 == 1):
                        self.addnode("inv_"+str(cols/2),[0,24*(cols/2)],InvOS)
                self.addpin('vdd',["m2_m1",66.5,-8])
                self.addpin('vwell',["m1_Nw",66.5,0-8])

        def add_arcs(self, cols):
                for i in range(0,(cols/2)-1):
                        self.addarc("vdd_"+str(i), \
                                        ["Metal-2", self.nref('inv_'+str(i)), \
                                                self.nref('inv_'+str(i+1)),'vdd','vdd',1])
                if (cols%2 == 1):
                        self.addarc("vdd_"+str((cols/2)-1), \
                                        ["Metal-2", self.nref('inv_'+str((cols/2)-1)), \
                                                self.nref('inv_'+str(cols/2)),'vdd','vdd',1])
                self.addarc("vdd_-1",["Metal-2",
self.nref('inv_'+str(0)),self.nref('vdd'),'vdd','',1])
                self.addarc("vin",["Metal-1", self.nref('vwell'),self.nref('vdd'),'','',1])
        def add_exports(self, cols):
                self.addexport("vdd", ["P", self.nref('vdd')])
                for i in range(0,cols):
                        if (i%2==0):
                                self.addexport("gnd_"+str(i/2), ["G",
self.nref('inv_'+str(i/2)),'gnd'])
                        self.addexport("a_"+str(i), ["I",
self.nref('inv_'+str(i/2)),'a_'+str(i%2)])
                        self.addexport("y_"+str(i), ["O",
self.nref('inv_'+str(i/2)),'y_'+str(i%2)])


########################################################################################
#Pullup bank
class Pulls(Cell):
        def __init__(self,name,top,left,cols,rot=''):
                width = 24*(cols/2)+16*(cols%2)+9
                height = 40
                Cell.__init__(self,name,'Pulls','mocmos',top,left,width,height,rot='')
                self.add_nodes(cols)
                self.add_arcs(cols)
                self.add_exports(cols)

        def add_nodes(self, cols):
                for i in range(0,cols/2):
                        self.addnode("pullup_"+str(i),[0,24*i],PullUpD)
                if (cols%2 == 1):
                        self.addnode("pullup_"+str(cols/2),[0,24*(cols/2)],PullUpS)
        def add_arcs(self, cols):
                for i in range(0,(cols/2)-1):
                        self.addarc("vdd_"+str(i), \
                                      ["Metal-2", self.nref('pullup_'+str(i)), \

         self.nref('pullup_'+str(i+1)),'vdd_l','vdd_r'])
                            self.addarc("gnd_"+str(i), \
                                      ["Metal-1", self.nref('pullup_'+str(i)), \

         self.nref('pullup_'+str(i+1)),'gnd_l','gnd_r'])
                            self.addarc("poly_"+str(i), \
                                      ["Polysilicon-1", self.nref('pullup_'+str(i)), \

         self.nref('pullup_'+str(i+1)),'poly_l','poly_r',1])
                   if (cols%2 == 1):
                            self.addarc("vdd_"+str((cols/2)-1), \
                                      ["Metal-2", self.nref('pullup_'+str((cols/2)-1)), \

         self.nref('pullup_'+str(cols/2)),'vdd_l','vdd_r'])
                            self.addarc("gnd_"+str((cols/2)-1), \
                                      ["Metal-1", self.nref('pullup_'+str((cols/2)-1)), \

         self.nref('pullup_'+str(cols/2)),'gnd_l','gnd_r'])
                            self.addarc("poly_"+str((cols/2)-1), \
                                      ["Polysilicon-1", self.nref('pullup_'+str((cols/2)-1)),
\

         self.nref('pullup_'+str(cols/2)),'poly_l','poly_r',1])

         def add_exports(self, cols):
                   self.addexport("vdd", ["P", self.nref('pullup_'+str(0)),'vdd_r'])
                   self.addexport("gnd", ["G", self.nref('pullup_'+str(0)),'gnd_r'])
                   self.addexport("poly", ["I", self.nref('pullup_'+str(0)),'poly_r'])
                   for i in range(0,cols):
                            if (i%2==0):
                                     self.addexport("gnd_"+str(i/2), ["G",
self.nref('pullup_'+str(i/2)),'gnd_out'])
                            self.addexport("y_"+str(i), ["O",
self.nref('pullup_'+str(i/2)),'y'+str(i%2)])

########################################################################################
#Input inverter bank
class InInv(Cell):
         printed = False
         def __init__(self,name,top,left,cols,rot=''):
                   width = cols*16+9
                   height = 102
                   Cell.__init__(self,name,'InInv','mocmos',top,left,width,height,rot='')
                   self.add_nodes(cols)
                   self.add_arcs(cols)
                   self.add_exports(cols)

         def add_nodes(self, cols):
                   for i in range(0,cols):
                            self.addnode("inv_"+str(i),[0,16*i],InvIN)
                   self.addpin("gwell",["m1_Pw",17,cols*16+8,0,0])
                   self.addpin("gm",["m2_m1",17,cols*16+8,0,0])

         def add_arcs(self, cols):
                   for i in range(0,cols-1):
                            self.addarc("vdd_"+str(i), \
                                      ["Metal-2", self.nref('inv_'+str(i)), \
                                               self.nref('inv_'+str(i+1)),'vdd','vdd'])
                            self.addarc("gnd_"+str(i), \
                                      ["Metal-2", self.nref('inv_'+str(i)), \
                                               self.nref('inv_'+str(i+1)),'gnd','gnd'])
                   self.addarc("gnd_"+str(cols-1), \
                                      ["Metal-2", self.nref('inv_'+str(cols-
1)),self.nref("gm"),'gnd'])
                   self.addarc("gw",["Metal-1", self.nref("gm"),self.nref("gwell")])

         def add_exports(self, cols):
                   self.addexport("vdd", ["P", self.nref('inv_'+str(0)),'vdd'])
                   self.addexport("gnd", ["G", self.nref("gm")])
```

```
                        for i in range(0,cols):
                                self.addexport("a_"+str(i), ["I", self.nref('inv_'+str(i)),'a'])
                                self.addexport("y_"+str(i), ["O", self.nref('inv_'+str(i)),"y"])
                                self.addexport("yn_"+str(i), ["O", self.nref('inv_'+str(i)),"yn"])

                def print_cell(self):
                        if not InInv.printed:
                                InInv.printed = True
                                return Cell.print_cell(self)
                        return ""

########################################################################################
#Input setup bank
class InSetup(Cell):
        def __init__(self,name,top,left,cols,rot=''):
                width = (cols*16)*2+25
                height = (cols*2+1)*4
                Cell.__init__(self,name,'InSetup','mocmos',top,left,width,height,rot='')
                self.add_nodes(cols)
                self.add_arcs(cols)
                self.add_exports(cols)

        def add_nodes(self, cols):
                for i in range(0,cols):
                        self.addpin("outP_"+str(i),["m1",0,i*16,0,0])
                        self.addpin("outN_"+str(i),["m1",0,i*16+cols*16+17,0,0])
                        self.addpin("inP_"+str(i),["m2_m1",(i+1)*-8,i*16,0,0])
                        self.addpin("inN_"+str(i),["m2_m1",(i+1)*-8,i*16+cols*16+17,0,0])

        def add_arcs(self, cols):
                for i in range(0,cols):
                        self.addarc("m1_0_"+str(i), \
                                        ["Metal-1", self.nref('outP_'+str(i)), \
                                                self.nref('inP_'+str(i))])
                        self.addarc("m1_1_"+str(i), \
                                        ["Metal-1", self.nref('outN_'+str(i)), \
                                                self.nref('inN_'+str(i))])
                        self.addarc("m2_"+str(i), \
                                        ["Metal-2", self.nref('inP_'+str(i)), \
                                                self.nref('inN_'+str(i))])

        def add_exports(self, cols):
                for i in range(0,cols):
                        self.addexport("a_"+str(i), ["I", self.nref('inP_'+str(i))])
                        self.addexport("yP_"+str(i), ["O", self.nref('outP_'+str(i))])
                        self.addexport("yN_"+str(i), ["O", self.nref('outN_'+str(i))])

########################################################################################
#Whole ROM
class ROM(Cell):
        def __init__(self,name,top,left,in_w,out_w,h,data,inname='a',outname='y',rot=''):
                width = 400
                height = 300
                Cell.__init__(self,name,name,'mocmos',top,left,width,height,rot='')
                self.add_nodes(in_w,out_w,h,data)
                self.add_arcs(in_w,out_w,h)
                self.add_exports(in_w,out_w,inname,outname)

        def add_nodes(self, in_w,out_w,h,data):
                #subunits
                self.addnode("insetup",[0,0,in_w],InSetup)
                self.addnode("invP",[-5,-8,in_w],InInv)
                self.addnode("invN",[-5,in_w*16+9,in_w],InInv)
                self.addnode("decP",[114,-4,in_w,h,data],DecoderP)
                self.addnode("decN",[114,in_w*16+13,in_w,h,data],DecoderN)
                self.addnode("enc",[110,in_w*32+33,h,out_w,data],Encoder)
                self.addnode("pulls",[58,in_w*32+25,out_w],Pulls)
                self.addnode("invO",[h*8+118,in_w*32+33,out_w],InvOut)
                #power and ground nwks
                self.addpin('g1',["m1_p",75.5,in_w*32+17,0,0])
                self.addpin('g2',["m1",88,in_w*32+17,0,0])
                self.addpin('g3',["m1",106,in_w*32+17,0,0])
                self.addpin('vdd',["m2_m1",h*8+184.5,-4,0,0])
                self.addpin('v1',["m2",71,-4,0,0])

        def add_arcs(self, in_w, out_w, h):
                #wordlines
                for i in range(0,h):
                        self.addarc("word0_"+str(i), \
                                        ["Metal-2", self.nref('decP'), \

        self.nref('decN'),'wordout_'+str(i),'wordin_'+str(i)])
                        self.addarc("word1_"+str(i), \
                                        ["Metal-1", self.nref('decN'), \

        self.nref('enc'),'wordout_'+str(i),'word_'+str(i)])
                        if i>0:
                                self.addarc("vdddec_"+str(i), \
                                        ["Metal-2", self.nref('decP'), \
                                                self.nref('decP'),"vdd_"+str(i),"vdd_"+str(i-
1)])
                #input columns
                for i in range(0, in_w):
                        self.addarc("inP"+str(i), \
                                        ["Metal-1", self.nref('insetup'), \
                                                self.nref('invP'),'yP_'+str(i),'a_'+str(i)])
                        self.addarc("aP"+str(i), \
                                        ["Polysilicon-1", self.nref('invP'), \
                                                self.nref('decP'),'y_'+str(i),'a_'+str(i)])
                        self.addarc("anP"+str(i), \
                                        ["Polysilicon-1", self.nref('invP'), \
                                                self.nref('decP'),"yn_"+str(i),"an_"+str(i)])
                        self.addarc("inN"+str(i), \
                                        ["Metal-1", self.nref('insetup'), \
                                                self.nref('invN'),'yN_'+str(i),'a_'+str(i)])
                        self.addarc("aN"+str(i), \
                                        ["Polysilicon-1", self.nref('invN'), \
                                                self.nref('decN'),'y_'+str(i),'a_'+str(i)])
                        self.addarc("anN"+str(i), \
                                        ["Polysilicon-1", self.nref('invN'), \
                                                self.nref('decN'),"yn_"+str(i),"an_"+str(i)])
                        if i >0:
                                self.addarc("gnddec_"+str(i), \
                                        ["Metal-1", self.nref('decN'), \
                                                self.nref('decN'),"gnd_"+str(i),"gnd_"+str(i-
1)])
                #output columns
                for i in range(0, out_w):
                        self.addarc("outT_"+str(i), \
                                        ["Metal-2", self.nref('pulls'), \
                                                self.nref('enc'),'y_'+str(i),'din_'+str(i)])
                        self.addarc("outB_"+str(i), \
                                        ["Metal-1", self.nref('enc'), \
                                                self.nref('invO'),'dout_'+str(i),'a_'+str(i)])
                        if(i%2==0):
                                self.addarc("gcolT_"+str(i/2), \
                                        ["Metal-1", self.nref('pulls'), \

        self.nref('enc'),'gnd_'+str(i/2),'gin_'+str(i/2)])
                                self.addarc("gcolB_"+str(i/2), \
                                        ["Metal-1", self.nref('enc'), \

        self.nref('invO'),'gout_'+str(i/2),'gnd_'+str(i/2)])
                #gnd nwk
                self.addarc("gnd0", ["Metal-1", self.nref('g1'),self.nref('invN'),'','gnd',1])
                self.addarc("gnd1", ["Metal-2", self.nref('invP'), self.nref('invN'),'gnd','gnd',1])
                self.addarc("gnd2", ["Polysilicon-1", self.nref('g1'),
self.nref('pulls'),'','poly',1])
                self.addarc("gnd3", ["Metal-1", self.nref('g1'), self.nref('g2'),'','',1])
                self.addarc("gnd4", ["Metal-1", self.nref('g2'), self.nref('pulls'),'','gnd',1])
                self.addarc("gnd5", ["Metal-1", self.nref('g2'), self.nref('g3'),'','',1])
                self.addarc("gnd6", ["Metal-1", self.nref('g3'),
self.nref('decN'),'','gnd_'+str(in_w-1),1])
                #pwr nwk
                self.addarc("vdd0", ["Metal-2", self.nref('vdd'),self.nref('invO'),'','vdd',1])
                self.addarc("vdd1", ["Metal-2", self.nref('vdd'),self.nref('decP'),'','vdd_'+str(h-
1),1])
                self.addarc("vdd2", ["Metal-2", self.nref('v1'), self.nref('decP'),'','vdd_0',1])
                self.addarc("vdd3", ["Metal-2", self.nref('v1'), self.nref('invP'),'','vdd',1])
                self.addarc("vdd4", ["Metal-2", self.nref('invP'), self.nref('invN'),'vdd','vdd',1])
                self.addarc("vdd5", ["Metal-2", self.nref('pulls'),
self.nref('invN'),'vdd','vdd',1])
        def add_exports(self,in_w,out_w,inname,outname):
                for i in range(0,in_w):
                        self.addexport(inname+"["+str(in_w-1-i)+"]", ["I",
self.nref('insetup'),'a_'+str(i)])
                for i in range(0,out_w):
                        self.addexport(outname+"["+str(out_w-1-i)+"]", ["O", self.nref('invO'),
'y_'+str(i)])
                self.addexport("vdd", ["P", self.nref("vdd")])
                self.addexport("gnd", ["G", self.nref('invN'),'gnd'])

######################################################################
# Verilog Parser

def parseline(line):
        bstr = lambda n: n>0 and bstr(n>>1).lstrip('0')+str(n&1) or '0';
        format = lambda l,r,v: bstr(int(v.replace('_',''),{'b':2,'h':16,'d':10}[r])).zfill(int(l));
        return format(*line[0:3]),format(*line[4:7])

def parsemod(vars):
        format = lambda n1,n2,name: (name,1+int(n1)-int(n2));
        if ((vars[1] == "input") and (vars[6] == "output")):
                return vars[0], format(*vars[3:6]), format(*vars[8:11])
        elif ((vars[6] == "input") and (vars[1] == "output")):
                return vars[0],format(*vars[8:11]),format(*vars[3:6])
        else:
                raise RuntimeError, "Wrong number and/or type of variables in module declaration"

def parseblock(text,start,end):
        text = text.strip();
        (pre,s,text) = text.partition(start);
        (text,e,post) = text.partition(end);
        if not (s and e):
                raise RuntimeError, 'No '+start+' statement in input file '
        return pre,text,post

def parse(filename):
        vfile = open(filename);
        text = "".join(vfile.readlines());
        vfile.close();

        #process module declaration
        premod,text,postmod = parseblock(text,'module','endmodule');
        text = text.strip();
        vardec = "(output|input)\s*(logic)\s*\[(\d+):(\d+)\]\s*(\w+)";
        mreg = re.compile("(\w+)\s*\(\s*"+vardec+"\s*,\s*"+vardec+"\);?");
        match = mreg.match(text);
        if not match:
                raise RuntimeError, 'Cannot parse module declaration.'
        romname,invar,outvar = parsemod(match.groups());
        text = text[match.end():]

        #process case declaration
        precase,text,postcase = parseblock(text,'case','endcase');
        text = text.strip();
        (invarn,nl,text) = text.partition('\n');
        invarn = invarn.strip('( );')

        #process case body
        text = text.strip();
        reg = re.compile("([0-9]+)'([bdh])(\w+)\s*:\s*(\w+)\s*<=\s*([0-9]+)'([bdh])(\w+)")
```

```
          lines = reg.findall(text)
          strs = map(parseline, lines)

          return romname,{'name':invar[0],'w':invar[1]},{'name':outvar[0],'w':outvar[1]},strs

######################################################################
# Output writers
def writeROMlib(filename,romname, ins, outs, strs):
          print strs
          print len(strs)
          file = open(filename,'w')
          file.write(printheader())
          rom = ROM(romname,0,0,ins['w'],outs['w'],len(strs),strs,ins['name'], outs['name'])
          file.write(rom.print_cell())

def writespicehex(filename, ins, outs, strs):
          file = open(filename,'w');
          rstr = "RADIX " + "".ljust(ins['w'],"1") + " " + "".ljust(outs['w'],"1") + "\n"
          rstr += "IO " + "".ljust(ins['w'],"i") + " " + "".ljust(outs['w'],"o") + "\n"
          rstr += "VNAME "
          for i in range(0,ins['w']):
                    rstr += ins['name']+"["+str(ins['w']-1-i)+"] "
          for i in range(0,outs['w']):
                    rstr += outs['name']+"["+str(outs['w']-1-i)+"] "
          rstr = """
;*********************************************************************
; stim.vec
; Generated with romgenerator.py
; Tests each instruction of generated ROM
;*********************************************************************
%s\n
; define the units and switching points
; these parameters are defined in the test bench
TUNIT ns\nPERIOD UNITLESSTC\nSLOPE UNITLESSEDGE\nVIH SUPPLY\nVIL 0\nVOH VOH\nVOL VOL\nVTH VSWITCH\n
; specify the vectors
""" % rstr
          file.write(rstr)
          for i in range(0,4):
                    file.write('%s %s\n' % (strs[0][0],"".ljust(outs['w'],"X")))
          for i in range(1,len(strs)):
                    file.write('%s %s\n' % (strs[i][0],strs[i-1][1]))
          file.write('%s %s\n' % ("".ljust(ins['w'],"X"),strs[len(strs)-1][1]))

######################################################################
#Begin Main

infile = sys.argv[1];
romname, ins, outs, strs = parse(infile);

writeROMlib(infile.split('.')[0]+".jelib",romname,ins,outs,strs);
writespicehex(infile.split('.')[0]+'_stim.vec',ins,outs,strs)
```

ROM Generator: layout_gen.py

```
#!/usr/bin/python
#layout_gen.py

"""
Example cells:
#######################################################################################
#Simple cell test
class Simple(Cell):
          def __init__(self,name,top,left,rot=''):
                    width = 40
                    height = 40
                    Cell.__init__(self,name,'Simple','mocmos',top,left,width,height,rot='')
                    self.add_nodes()
                    self.add_arcs()
                    self.add_exports()

          def add_nodes(self):
                              self.addpin("pin1",["poly",0,0])
```

```
                                self.addmos("pin2",["n-trans",0,10,1,0,'RR'])
                                self.addpin("pin3",["m1_aN",10,10])
                def add_arcs(self):
                                self.addarc("arc1",["Polysilicon-1",
self.nref("pin1"),self.nref("pin2"),'','right'])
                                self.addarc("arc2",["N-Active",
self.nref("pin3"),self.nref("pin2"),'','top'])

                def add_exports(self):
                        self.addexport("pin1", ["O", self.nref("pin1")])

#######################################################################################
#Simple cell test
class Complex(Cell):
                def __init__(self,name,top,left,rot=''):
                        width = 40
                        height = 40
                        Cell.__init__(self,name,'Complex','mocmos',top,left,width,height,rot='')
                        self.add_nodes()
                        self.add_arcs()
                        self.add_exports()

                def add_nodes(self):
                                self.addpin("pin1",["poly",0,0])
                                self.addnode("simp",[10,0],Simple)
                def add_arcs(self):
                                self.addarc("arc1",["Polysilicon-1",
self.nref("pin1"),self.nref("simp"),'','pin1'])

                def add_exports(self):
                        "bbjb"

######################################################################################
"""



import re, sys

###########################################################
#General helpers

nodenames = { #short pin type names, width, height
  'aN': ["N-Active-Pin",4,4],
  'aP': ["P-Active-Pin",4,4],
  'Nw': ["N-Well-Node",4,4],
  'Pw': ["P-Well-Node",4,4],
  'poly':["Polysilicon-1-Pin",2,2],
  'm1': ["Metal-1-Pin",4,4],
  'm2': ["Metal-2-Pin",4,4],
  'm1_Nw': ["Metal-1-N-Well-con",4,4],
  'm1_Pw': ["Metal-1-P-Well-Con",4,4],
  'm1_aN': ["Metal-1-N-Active-Con",4,4],
  'm1_aP': ["Metal-1-P-Active-Con",4,4],
  'm1_p':  ["Metal-1-Polysilicon-1-Con",4,4],
  'm2_m1': ["Metal-1-Metal-2-Con",4,4],
  'n-trans': ["N-Transistor",3,2],
  'p-trans': ["P-Transistor",3,2]
}

def und(a,b): #helper for names
                return str(a)+"_"+str(b)

###########################################################
# Basic Class Definitions
class Node:
                "Represents a node object in the layout"
                def __init__(self,name,type,y,x,width=0,height=0,rot=''):
                        try:
                                self.defaults = nodenames[type]
                        except:
                                self.defaults = [type,0,0]
                        (self.name, self.type, self.x, self.y, self.rotation) = (name, type, x, y, rot)
                        (self.w, self.h) = ((self.defaults[1] + width), (self.defaults[2] + height))

                def __str__(self):
                        return 'N'+'|'.join(map(str,[self.defaults[0],self.name,'',self.x,-self.y, \
                                                               self.w-self.defaults[1],self.h-
self.defaults[2],self.rotation,'']))

                def exp_loc(self, exp=''):
                        return (self.x,self.y)

                def map_exp(self,str):
                        return str

                def print_cell(self):
                        return ""

class Arc():
                "Represents an arc object in the layout"
                def __init__(self,name,type,head,tail,h_exp='',t_exp='',width=''):
                        (self.name, self.type, self.width) = (name, type, width)
                        self.h_n = head.name
                        self.h_exp = head.map_exp(h_exp)
                        (self.h_x, self.h_y) = head.exp_loc(h_exp)
                        self.t_n = tail.name
                        self.t_exp = tail.map_exp(t_exp)
                        (self.t_x, self.t_y)  = tail.exp_loc(t_exp)

                def __str__(self):
                        return "A"+'|'.join(map(str,[self.type,self.name,'',self.width,'', \
                                                self.h_n,self.h_exp,self.h_x,-
self.h_y,self.t_n,self.t_exp,self.t_x,-self.t_y]))
class Mos(Node):
                "Represents a transistor object in the layout"
                def map_exp(self,str):
                        return self.type+{'right': "-poly-right", 'left': "-poly-left",
                                                                        'top':"-diff-top", 'bottom': "-diff-
bottom"}[str]

                def exp_loc(self, name):
                        side = {'top':0,'left':1,'bottom':2,'right':3}[name]
                        delta = {1:self.w/2.0+2,0:self.h/2.0+3}[side%2]
                        side = (side + len(self.rotation)) % 4
                        return {0:(self.x,self.y-delta), 1:(self.x-delta,self.y), \
                                                2:(self.x,self.y+delta), 3:(self.x+delta,self.y)}[side]

class Export():
                def __init__(self,name,type,node,port=''):
                        (self.name,self.type,self.node,self.port) = (name,type,node.name,node.map_exp(port))
                        exploc = node.exp_loc(port)
                        (self.x, self.y) = (exploc[0], exploc[1])

                def __str__(self):
                        return "E%s||D5G2;|%s|%s|%s" %(self.name,self.node,self.port,self.type)

class Cell(Node):
                "Represents a generic cell"
                printed = False
                def __init__(self,name,type,tech,top,left,width,height,rot=''):
                        self.tech = tech
                        self.nodes = []
                        self.arcs = []
                        self.exports = []
                        self.alookup = {}
                        self.nlookup = {}
                        self.elookup = {}
                        Node.__init__(self,name,type,top,left,width,height,rot)

                def nref(self,name):
                        return self.nodes[self.nlookup[name]]
```

```
	def addx(self,name,args,lst,dict,cls):
		try:
			return dict[name]
		except:
			dict[name] = len(lst)
			lst.append(cls(name,*args))

	def addnode(self,name,args,cls):
		self.addx(name,args,self.nodes,self.nlookup,cls)

	def addpin(self,name,args):
		self.addx(name,args,self.nodes,self.nlookup,Node)

	def addmos(self,name,args):
		self.addx(name,args,self.nodes,self.nlookup,Mos)

	def addarc(self,name,args):
		self.addx(name,args,self.arcs,self.alookup,Arc)

	def addexport(self,name,args):
		self.addx(name,args,self.exports,self.elookup,Export)

	def exp_loc(self, name):
		exp = self.exports[self.elookup[name]]
		return (self.x+exp.x,self.y+exp.y)

	def __str__(self):
		return "I%s{lay}|%s||%s|%s|||D5G4;" % (self.type,self.name,self.x,-self.y)

	def print_cell(self):
		"writes cell definition"
		print self.name
		out = "\n"
		for n in self.nodes: #print subcell definitions
			out += n.print_cell()
		out += "\n# Cell %s{%s}" % (self.type,'lay')
		out +="\nC%s{%s}||%s|1204615247283|1204615267901|E" % (self.type,'lay',self.tech)
		out += "\n"+"\n".join(map(str,self.nodes)) #this cells nodes
		out += "\n"+"\n".join(map(str,self.arcs)) #this cells arcs
		out += "\n"+"\n".join(map(str,self.exports)) #this cells exports
		out += "\nX"
		return out


########################################################################
def printheader():
	return """
# header information:
Htest_rom|8.06

# Views:
Vlayout|lay
Vschematic|sch

# Technologies:
Tmocmos|MoCMOSAlternateActivePolyRules()I1|MoCMOSNumberOfMetalLayers()I3|MoCMOSSecondPolysilicon()I0|Sca
leFORmocmos()D300.0
"""
```

Appendix: vcd2sp and vcd2cmd

The vcd2sp/vcd2cmd package contains three perl scripts and a configuration script. Vcd2sp.pl and vcd2cmd.pl are frontend programs to convert a Verilog Change Dump (.vcd) to a SPICE digital vector file (.vec) and an IRSIM command file (.cmd), respectively. They both take as an argument the .vcd filename.

To generate a Verilog Change Dump, it is necessary to add three commands to Verilog testbenches: `$dumpfile`, `$dumpvars`, and `$dumpflush`. `$dumpfile("myVCD.vcd");` specifies the path of the VCD file generate (in this example, myVCD.vcd is the file generated). `$dumpvars(1, var1,  …, varn);` selects signals to include in the dump. The first argument is levels of recursion. The vcd2sp/vcd2cmd scripts to not support more than one Verilog module in the VCD, and thus the first argument should always be 1. `$dumpflush` flushes the write buffer, writing the VCD file to disk.

Vcd2sp.pl and vcd2cmd.pl both use vcd2.pl as a backend script to parse the Verilog Change Dump file. Each then generated a stimulus file from the parsed data. VCD files contain textual data equivalent to waveforms in Modelsim, and thus are not synchronous with a clock (i.e. signals can change at any time, not just at clock edges). The SPICE digital vector file and IRSIM command file are synchronous with clock. To configure clock edges to apply inputs and assert ouputs, a vcd2sp/vcd2cmd configuration file is used. The file is called myVCD.conf, where myVCD is the basename of the VCD file given in script arguments. myVCD.conf is a perl read by vcd2sp/vcd2cmd a run-time, and thus follows perl's syntax. The config file also lists input, output, and bidirectional ports, as the VCD file does not specify signal direction. SPICE digital vector file signal delays can be specificied in the vcd2sp, though vcd2cmd does not use these when generating .cmd files. Timesteps are assumed to be in nanoseconds. A summary of variables supported in the configuration file is given below.

Vcd2sp/vcd2cmd configuration file variables

Variable	**Example Value**	**Description**
$clkname	"ph0"	Clock signal name to synchronize assertion of outputs and application of inputs, for .vec and .cmd generation
$clkdir	"fall" or "rise"	Direction of clock edge
$clkdelay	1 or 0	To record inputs/outputs on clock edge, or one timestep before
@input	('ph0', 'resetb')	Array of input bus names
@output	('address', read_en')	Array of output bus names
@inout		Array of bidirectional bus names
%enable	"~read_en"	Hash of bidirectional buses and corresponding enable signals (ENABLE in .vec file)
%intiming	900	Hash of bidirectional buses and corresponding input delays (IDELAY in .vec file)
%outtiming	800	Hash of bidirectional buses and corresponding output delays (ODELAY in .vec file)
%timing	225	Hash of input/output buses and corresponding delays (DELAY in .vec file)
$irsimspeed	1000	Clock speed in IRSIM simulation. A value of 1000 (ns) is then 1 Mhz.

$clkname and $clkdir specifies the name and clock edge direction to synchronize to. With the example values given above, vcd2sp.pl will generate vectors that change on the rising or falling edge of ph0. The state of inputs/outputs on this edge in VCD file will be used for the generated vectors. If $clkdelay is 0, the state is recorded on the clock edge. If $clkdelay is 1, the state is recorded immediately before the clock edge, and is useful if there are no delays in the RTL used to generate the VCD file. Each input/output pair corresponds to one line in the .vec stimulus. Depending on delays in the configuration file, the output corresponds to the previous inputs or to the current inputs for the SPICE digital vector file. For the .cmd, inputs are applied. After one cycle, outputs are checked and new inputs applied. Thus the configuration file should be configured so that every time inputs/outputs are recorded (every clock edge), the output vector is the corresponding output of the device-under-test after one cycle of the input.

The enable signals given in the %enable must be inputs to the chip. For the 6502, read_en is an output from the chip as discussed in the Verificiation section of this report. The .vec generated from vcd2sp was manually edited to fix contention issues, as discussed in the Verification section.

Vcd2sp also generates a duration_myVCD.sp, which contains values for the reset and runtime cycle counts, which are used in SPICE testbench file. A sample testbench file (testbench.sp) is given below, along with the perl scripts and configuration file.

outSuiteP.conf

```
################################################################################
# outSuiteP.conf: vcd2sp and vcd2cmd configuration file for Suite P.
################################################################################

# Define when to check inputs and outputs.
$clkname = "ph0";
$clkdir = "fall";
$clkdelay = 1;

# Define inputs and outputs.
@input = ('ph0', 'resetb', 'data_in');
@output = ('ph1', 'ph2', 'address', 'data_out', 'read_en', 'razor_error');
@inout = ('data');

$enable{"data"} = "~read_en";
$intiming{"data"} = "900";
$outtiming{"data"} = "800";
$timing{"address"} = "225";
$timing{"read_en"} = "225";
$timing{"resetb"} = "900";

$irsimspeed = 1000;

return true;
```

vcd2.pl

```
#!/usr/bin/perl
#
# vcd2sp.pl: Converts a Verilog Value Change Dump
# to a spice digital vector stimulus.
#
# Date: May 1, 2008
# Author: Nathaniel Pinckney
#

use File::Basename;

# Parse a VCD file and return its data.
sub parseVCD() {
      %VCD;

      open(INFILE, $vcdFilename) or die "Can't open VCD file: $!\n";
      while(<INFILE>) {
            $_ = trim($_);
            if(/\$date/) {
                  $VCD{date} = readDeclaration();
                  next;
            } elsif(/\$version/) {
                  $VCD{version} = readDeclaration();
                  next;
            } elsif(/\$timescale/) {
                  my $tmptime = readDeclaration();
                  # For right now we just extract the units
                  # in the future we'd want to keep track of the scale.
                  $tmptime =~ /(\ws)/g;
                  $VCD{timescale} = $1;
                  next;
            } elsif(/\$var/) {
                  # A variable definition
                  $_ = split(/ /);
                  if($_[1] =~ /(reg|wire)/) {
                        my $size = $#_;
                        # $VCD{'ports'}{$_[3]}{'type'} = $1;
                        $VCD{'ports'}{$_[3]}{'name'} = $_[4];
                        $VCD{'ports'}{$_[3]}{'size'} = $_[2];
                        if($size == 6) {
                              $VCD{'ports'}{$_[3]}{'portnum'} = $_[5];
                        }
                        if(grep $_ eq $VCD{'ports'}{$_[3]}{'name'}, @input) {
                              $VCD{'ports'}{$_[3]}{'type'} = 'input';
                        } elsif(grep $_ eq $VCD{'ports'}{$_[3]}{'name'}, @output) {
                              $VCD{'ports'}{$_[3]}{'type'} = 'output';
                        } elsif(grep $_ eq $VCD{'ports'}{$_[3]}{'name'}, @inout) {
                              $VCD{'ports'}{$_[3]}{'type'} = 'inout';
                        } else {
                              print STDERR @foo . "\n";
                              die("Port " . $VCD{'ports'}{$_[3]}{'name'} . " not
found in configuration file");
                        }
                  }
            } elsif(/^#\d+/) {
                  # Time marker
                  newTime();
            } elsif(/\$dumpvars/) {
                  # begin a variable dump
                  readDumpVars();
            } elsif(isVar) {
                  processVar();
            }
      }
      close(INFILE) or die "Can't close VCD file: $!\n";
}

sub isVar() {
      if(/^(x|0|1)(\S+)/) {
            return 1;
      } elsif(/^b(\S+)\s+(\S+)/) {
            return 1;
      } else {
            return 0;
      }
}

sub processVar() {
      if(/^(x|0|1)(\S+)/) {
            # This is a single assignment

      $VCD{'dump'}{$curtime}{"$VCD{'ports'}{$2}{'name'}$VCD{'ports'}{$2}{'portnum'}
"} = $1;
            if(1 != portSize($VCD{'ports'}{$2}{'portnum'})) {
                  die("Port $VCD{'ports'}{$2}{'name'}$VCD{'ports'}{$2}{'portnum'}
was incorrect size.")
            }
      }
      if(/^b(\S+)\s+(\S+)/) {
            # This is a bus assignment
            my $tmpstr = extendVector($1,portSize($VCD{'ports'}{$2}{'portnum'}));

      $VCD{'dump'}{$curtime}{"$VCD{'ports'}{$2}{'name'}$VCD{'ports'}{$2}{'portnum'}
"} = $tmpstr;
      }

}

sub extendVector(my $str, my $size) {
      my $str = $_[0];
      my $size = $_[1];
      if(length($str) > $size) {
            die("Port $VCD{'ports'}{$2}{'name'}$VCD{'ports'}{$2}{'portnum'} was too
big.");
      } else {
            my $chr = substr($str,-1);
            $chr =~ tr/10zxZX/00zxZX/;
            $str = ($chr x ($size - length($str))) . $str;
      }
      return $str;
}

sub newTime() {
      /^#(\d+)/;
      my $oldtime = $curtime;
      $curtime = $1;
      addTime($curtime, $oldtime);
      #if($curtime != 0) {
      #     for(my $time = $oldtime + 1; $time <= $curtime; $time++) {
      #           addTime($time, $oldtime);
      #     }
      #} else {
      #     # for first time, hope it is a $dumpvar
      #     # TODO: check for this
      #}
}

sub addTime() {
      my $time = $_[0];
      my $oldtime = $_[1];
      foreach my $k (sort keys %{$VCD{'ports'}}) {
            my %port = %{$VCD{'ports'}{$k}};
            #if($port{'type'} eq 'wire') {
            #     # Output, don't care if not specified.
            #     $VCD{'dump'}{$time}{"$port{'name'}$port{'portnum'}"} = 'x' x
portSize($port{'portnum'});
            #} else {
                  # Input, hold inputs.
                  $VCD{'dump'}{$time}{"$port{'name'}$port{'portnum'}"} =
$VCD{'dump'}{$oldtime}{"$port{'name'}$port{'portnum'}"};
            #}
      }
}

# Read in a dump var.  Initial list of variables.
sub readDumpVars() {
      while(<INFILE>) {
            $_ = trim($_);
            if(/\$end/) { last; }
            processVar();
      }
}

# Can handle both a bus and single variable.
sub readValue() {
      # [Radix][Values] [Identifier]
      # [Value][Identifier]
}

sub readDeclaration() {
      my $string;

      while(<INFILE>) {
            $_ = trim($_);
            if(/\$end/) { return $string };
            $string = $_ if not $string;
      }
}

# Returns port size of a register/wire
sub portSize() {
      $_= $_[0];
      if(/\[(\d+):(\d+)\]/) {
            return ($1 - $2 + 1);
      } elsif(/\[(\d+)\]/) {
            return 1;
      } else {
            return 1;
      }

}

### Utilities
sub trim($) {
      my $string = shift;
      $string =~ s/^\s+//;
      $string =~ s/\s+$//;
      return $string;
```

```
}

return true;
```

vcd2sp.pl

```
#!/usr/bin/perl
#
# vcd2sp.pl: Converts a Verilog Value Change Dump
# to a spice digital vector stimulus.
#
# Date: May 1, 2008
# Author: Nathaniel Pinckney
#

use File::Basename;

package main;
# require "vcd2.conf";
require "vcd2.pl";

main();

sub print_header() {
      print
";**********************************************************************\n";
      print "; $basename.vec\n";
      print "; Digital Vector file for stimulus\n";
      print ";\n";
      print "; Automatically generated by vcd2sp from '$vcdFilename'\n";
      print "; Original .vcd file from $VCD{version} ($VCD{date})\n";
      print
";**********************************************************************\n";
      print "\n";
}

sub print_wire_list() {
      print "RADIX ";
      foreach my $k (sort keys %{$VCD{'ports'}}) {
            my %port = %{$VCD{'ports'}{$k}};
            if(isSkip($port{'name'} =~ /(clk|reset|ph0|ph1|ph2)/i)) {
                  next;
            }
            my $char;
            print '1' x portSize($port{'portnum'});
            print " ";
      }
      print "\n\n";

      print "IO ";
      foreach my $k (sort keys %{$VCD{'ports'}}) {
            my %port = %{$VCD{'ports'}{$k}};
            if(isSkip($port{'name'})) {
                  next;
            }
```

```
		my $char;
		if($port{'type'} eq 'output') { $char = 'o'; }
		elsif($port{'type'} eq 'input') { $char = 'i'; }
		elsif($port{'type'} eq 'inout') { $char = 'b'; }
		print $char x portSize($port{'portnum'});
		print " ";
	}
	print "\n\n";

	print "VNAME ";
	# TODO: abstract into block...
	foreach my $k (sort keys %{$VCD{'ports'}}) {
		my %port = %{$VCD{'ports'}{$k}};

		if(isSkip($port{'name'})) {
			next;
		}
		if($port{'portnum'} =~ /\[(\d+):(\d+)\]/) {
			($start, $end) = ($2, $1);
			 for(my $i = $end; $i >= $start; $i--) {
				print uc($port{'name'}) . '[' . $i . ']' . ' ';
			}
		} elsif($port{'portnum'} =~ /\[(\d+)\]/) {
			print uc($port{'name'}) . '[' . $1 . ']' . ' ';
		} else {
			print uc($port{'name'}) . ' ';
		}
	}
	print "\n\n";

	foreach my $enablename (sort keys %enable) {
		print "ENABLE " . $enable{$enablename} . " ";
		# TODO: abstract into block...
		foreach my $k (sort keys %{$VCD{'ports'}}) {
				my %port = %{$VCD{'ports'}{$k}};
				if(isSkip($port{'name'})) {
					next;
				}
				my $char;
				if($port{'name'} eq $enablename) { $char = '1'; }
				else { $char = '0'; }
				print $char x portSize($port{'portnum'});
				print " ";
		}
		print "\n";
	}

	foreach my $timingname (sort keys %timing) {
		print "TDELAY " . $timing{$timingname} . " ";
		# TODO: abstract into block...
		foreach my $k (sort keys %{$VCD{'ports'}}) {
				my %port = %{$VCD{'ports'}{$k}};
				if(isSkip($port{'name'})) {
					next;
				}
				my $char;
				if($port{'name'} eq $timingname) { $char = '1'; }
```

```
                        else { $char = '0'; }
                        print $char x portSize($port{'portnum'});
                        print " ";
                }
                print "\n";
        }

        foreach my $timingname (sort keys %intiming) {
                print "IDELAY " . $intiming{$timingname} . " ";
                # TODO: abstract into block...
                foreach my $k (sort keys %{$VCD{'ports'}}) {
                        my %port = %{$VCD{'ports'}{$k}};
                        if(isSkip($port{'name'})) {
                                next;
                        }
                        my $char;
                        if($port{'name'} eq $timingname) { $char = '1'; }
                        else { $char = '0'; }
                        print $char x portSize($port{'portnum'});
                        print " ";
                }
                print "\n";
        }

        foreach my $timingname (sort keys %outtiming) {
                print "ODELAY " . $outtiming{$timingname} . " ";
                # TODO: abstract into block...
                foreach my $k (sort keys %{$VCD{'ports'}}) {
                        my %port = %{$VCD{'ports'}{$k}};
                        if(isSkip($port{'name'})) {
                                next;
                        }
                        my $char;
                        if($port{'name'} eq $timingname) { $char = '1'; }
                        else { $char = '0'; }
                        print $char x portSize($port{'portnum'});
                        print " ";
                }
                print "\n";
        }
        print "\n";
}

sub printVectors() {
        my $oldclk;
        my $clk;
        my $oldtime;

        $resetcount=0;
        $runcount=0;

        foreach my $time (sort {$a <=> $b} keys %{$VCD{'dump'}}) {
                my %vector = %{$VCD{'dump'}{$time}};

                $oldclk = $clk;
                $clk = $vector{$clkname};
                if(($oldclk == 1 && $clk == 0 && $clkdir eq "fall") or
               ($oldclk == 0 && $clk == 1 && $clkdir eq "rise")) {
                  # Count run vs reset
                  if($vector{'reset'} eq '1' || $vector{'resetb'} eq '0') {
$resetcount++; }
                  else { $runcount++; }
                  # Print vector
                  if($clkdelay == 1) { print1Vector($oldtime); }
                  else { print1Vector($time);   }
            }
            $oldtime = $time;
      }
      # Print the last vector, inputs are XXX's
      %oldvector = %{$VCD{'dump'}{$oldtime}};
      print1Vector($oldtime, -1);

      print "\n";
}

sub print1Vector {
      my ($time) = @_;
      my %vector = %{$VCD{'dump'}{$time}};

      foreach my $k (sort keys %{$VCD{'ports'}}) {
            my %port = %{$VCD{'ports'}{$k}};
            # TODO: This should be a list
            if(isSkip($port{'name'})) {
                  next;
            }
            if($port{'type'} eq 'output') {
                  # output
                  #if(! exists $oldvector{"$port{'name'}$port{'portnum'}"} ||
$oldvector{'reset'} == 1) {
                  if(! exists $vector{"$port{'name'}$port{'portnum'}"} ||
$vector{'reset'} == 1) {
                  #     print "ERROR: Time $time COULDN'T FIND
$port{'name'}$port{'portnum'}\n"
                        print "X" x portSize($port{'portnum'});
                  } else {
                  #     print uc($oldvector{"$port{'name'}$port{'portnum'}"});
                        print uc($vector{"$port{'name'}$port{'portnum'}"});
                  }
            } elsif($port{'type'} eq 'input') {
                  # input
                  if(! exists $vector{"$port{'name'}$port{'portnum'}"}) {
                  #     print "ERROR: Time $time COULDN'T FIND
$port{'name'}$port{'portnum'}\n"
                        print "X" x portSize($port{'portnum'});
                  } else {
                        print uc($vector{"$port{'name'}$port{'portnum'}"});
                  }
            } elsif($port{'type'} eq 'inout') { # Bidirectional
                  # input
                  if(! exists $vector{"$port{'name'}$port{'portnum'}"}) {
                  #     print "ERROR: Time $time COULDN'T FIND
$port{'name'}$port{'portnum'}\n"
                        print "X" x portSize($port{'portnum'});
                  } else {
                        print uc($vector{"$port{'name'}$port{'portnum'}"});
                  }
            }
            print " ";
      }
      print "\n";
}

sub isSkip() {
      ($in) = @_;
      if($in =~ /(clk|ph0|ph1|ph2)/i) {
            return 1;
      } else {
            return 0;
      }
}

sub writeSP() {
      open(SAVEOUT, ">&STDOUT");
      open(STDOUT, ">$basename.vec") or die("Can't open $basename.vec for
writing");

      # Print header
      print_header();

      print_wire_list();

      #
      print "; define the units and switching points\n";
      print "; these parameters are defined in the test bench\n";
      #print "TUNIT $VCD{timescale}\n";
      print "TUNIT ns\n";
      print "PERIOD UNITLESSTC\n";
      print "SLOPE UNITLESSEDGE\n";
      print "VIH SUPPLY\n";
      print "VIL 0\n";
      print "VOH VOH\n";
      print "VOL VOL\n";
      print "VTH VSWITCH\n";
      print "\n";

      #
      print "; specify the vectors\n";
      print "; the first cycles are for reset\n";
      print "; check the output of the last cycle\n";
      print "; at the same time as applying inputs\n";
      print "; for the next cycle\n";

      # Print vectors here
      printVectors();

      close(STDOUT);
      open(STDOUT, ">&SAVEOUT") or die("Can't open original STDOUT");
}

sub main() {
      # Parse command line arguments
```

```
	$numArgs = $#ARGV + 1;
	$vcdFilename = $ARGV[$#ARGV];
	$basename = basename($vcdFilename, ".vcd");

	if($numArgs != 1) { usage() };

	eval('require "' . "$basename.conf" . '";') or die "Couldn\'t load config
file $basename.conf";

	parseVCD();
	writeSP();
	writeDuration();
}

sub writeDuration() {
	open(SAVEOUT, ">&STDOUT");
	open(STDOUT,">duration_$basename.sp") or die("Can't open
duration_$basename.sp for writing");

	print
"**********************************************************************\n";
	print "* duration_$basename.sp\n";
	print "* Duration of simulation\n";
	print "*\n";
	print "* Automatically generated by vcd2sp from '$vcdFilename'\n";
	print "* Original .vcd file from $VCD{version} ($VCD{date})\n";
	print
"**********************************************************************\n";
	print "\n";

	print
"**********************************************************************\n";
	print "* Simulation Durations\n";
	print
"**********************************************************************\n";
	# TODO: This timing needs some work.
	print ".param FIRSTCYCLE=$resetcount * first nonreset cycle\n";
	print ".param SIMCYCLES='FIRSTCYCLE+$runcount'\n";

	close(STDOUT);
	open(STDOUT, ">&SAVEOUT") or die("Can't open original STDOUT");
}

sub usage() {
	print STDERR "Usage: ./vcd2sp.pl input.vcd\n";
	exit;
}
```

vcd2cmd.pl

```
#!/usr/bin/perl
#
# vcd2cmd.pl: Converts a Verilog Value Change Dump
# to an IRSIM cmd file.
#
# Date: May 1, 2008
# Author: Nathaniel Pinckney
```

```
#

use File::Basename;

package main;
require "vcd2.pl";

main();

sub print_header() {
      print "|$basename.cmd\n";
#     print
"//////////////////////////////////////////////////////////////////////\n";
#     print "// $basename.cmd\n";
#     print "// IRSIM .cmd file\n";
#     print "//\n";
#     print "// Automatically generated by vcd2cmd from '$vcdFilename'\n";
#     print "// Original .vcd file from $VCD{version} ($VCD{date})\n";
#     print
"//////////////////////////////////////////////////////////////////////\n";
      print "\n";
      print "stepsize $irsimstepsize\n";
      print "\n";
}

sub printVectors() {
      my $oldclk;
      my $clk;
      my $oldtime;
      my %oldvector;

      $resetcount=0;
      $runcount=0;
      my $count = 0;

      foreach my $time (sort {$a <=> $b} keys %{$VCD{'dump'}}) {
            my %vector = %{$VCD{'dump'}{$time}};

            # Todo: Case insentitivity
            $oldclk = $clk;
            $clk = $vector{$clkname};
            if(($oldclk == 1 && $clk == 0 && $clkdir eq "fall") or
               ($oldclk == 0 && $clk == 1 && $clkdir eq "rise")) {
                  if($clkdelay == 1) { print1Vector($oldtime, $count); }
                  else { print1Vector($time, $count); }
                  # Count run vs reset
                  if($vector{'reset'} eq '1' || $vector{'resetb'} eq '0') {
$resetcount++; }
                  else { $runcount++; }
                  $count++;
            }
            $oldtime = $time;
      }
      # Print the last vector, inputs are XXX's
      %oldvector = %{$VCD{'dump'}{$oldtime}};
      print1Vector($oldtime, $count);
      print "\n";
}

sub print1Vector {
      my ($time, $count) = @_;
      my %vector = %{$VCD{'dump'}{$time}};

      print "| $count\n";

      if($clkdir eq "rise") {
            print "h $clkname\n";
      } elsif($clkdir eq "fall") {
            print "l $clkname\n";
      }

      # Inout
      foreach my $k (sort keys %{$VCD{'ports'}}) {
            my %port = %{$VCD{'ports'}{$k}};

            if($port{'type'} eq 'inout') {
                  if($enable{$port{'name'}} =~ /^(~)?(\S+)$/) {
                        if(($1 eq '~' and $vector{$2} eq '0') or
                           ($1 eq '' and $vector{$2} eq '1')) {
                              # output
                              if($vector{'reset'} eq '1' or $vector{'resetb'} eq
'0') {

                                    next;
                              }

                              if($port{'portnum'} =~ /\[(\d+):(\d+)\]/) {
                                    ($start, $end) = ($2, $1);
                                     for(my $i = $end; $i >= $start; $i--) {
                                          print "x " . lc($port{'name'}) . '[' . $i
. ']' . "\n";

                                    }
                              } elsif($port{'portnum'} =~ /\[(\d+)\]/) {
                                    print "x " . lc($port{'name'}) . '[' . $1 . ']'
. "\n";
                              } else {
                                    print "x " . lc($port{'name'}) . "\n";
                              }
                        }
                  }
            }
      }

      print "s\n";
      print "s\n";

      # A step
      if($clkdir eq "rise") {
            print "l $clkname\n";
      } elsif($clkdir eq "fall") {
            print "h $clkname\n";
      }
      print "s\n";
      # Need two passes: inputs then outputs.

      # Inputs
      foreach my $k (sort keys %{$VCD{'ports'}}) {
            my %port = %{$VCD{'ports'}{$k}};
            # TODO: This should be a list
            #if($port{'name'} =~ /(clk|reset|ph1|ph2)/i) {
            if($port{'name'} =~ /(clk|ph[0123456789])/i) {
                  next;
            }
            if($port{'type'} eq 'input')  {
                  # input
                  if($port{'portnum'} =~ /\[(\d+):(\d+)\]/) {
                        ($start, $end) = ($2, $1);
                         for(my $i = $end; $i >= $start; $i--) {

      if(substr($vector{"$port{'name'}$port{'portnum'}"},$end - $i,1) eq "0") {
                                    print "l " . lc($port{'name'}) . '[' . $i . ']'
. "\n";
                              }
elsif(substr($vector{"$port{'name'}$port{'portnum'}"},$end - $i,1) eq "1") {
                                    print "h " . lc($port{'name'}) . '[' . $i . ']'
. "\n";
                              }
                        }
                  } elsif($port{'portnum'} =~ /\[(\d+)\]/) {
                        if($vector{"$port{'name'}$port{'portnum'}"}  eq "0") {
                              print "l " . lc($port{'name'}) . '[' . $1 . ']' .
"\n";
                        } elsif($vector{"$port{'name'}$port{'portnum'}"} eq "1") {
                              print "h " . lc($port{'name'}) . '[' . $1 . ']' .
"\n";
                        }
                  } else {
                        if($vector{"$port{'name'}$port{'portnum'}"} eq "0") {
                              print "l " . lc($port{'name'}) . "\n";
                        } elsif($vector{"$port{'name'}$port{'portnum'}"} eq "1") {
                              print "h " . lc($port{'name'}) . "\n";
                        }
                  }
            }
      }


      # Inout
      foreach my $k (sort keys %{$VCD{'ports'}}) {
            my %port = %{$VCD{'ports'}{$k}};

            if($port{'type'} eq 'inout') {
                  if($enable{$port{'name'}} =~ /^(~)?(\S+)$/) {
                        if(($1 eq '~' and $vector{$2} eq '0') or
                           ($1 eq '' and $vector{$2} eq '1')) {
                              # output
                              if($vector{'reset'} eq '1' or $vector{'resetb'} eq
'0') {
                                    next;
                              }
                              if($port{'portnum'} =~ /\[(\d+):(\d+)\]/) {
                                    ($start, $end) = ($2, $1);
                                     for(my $i = $end; $i >= $start; $i--) {

      if(substr($vector{"$port{'name'}$port{'portnum'}"},$end - $i,1) eq "0") {
                                                print "assert " . lc($port{'name'})
. '[' . $i . ']' . " 0\n";
                                          }
elsif(substr($vector{"$port{'name'}$port{'portnum'}"},$end - $i,1) eq "1") {
                                                print "assert " . lc($port{'name'})
. '[' . $i . ']' . " 1\n";
                                          }
                                    }
                              } elsif($port{'portnum'} =~ /\[(\d+)\]/) {
                                    if($vector{"$port{'name'}$port{'portnum'}"}  eq
"0") {
                                          print "assert " . lc($port{'name'}) . '['
. $1 . ']' . " 0\n";
                                    }
elsif($vector{"$port{'name'}$port{'portnum'}"} eq "1") {
                                          print "assert " . lc($port{'name'}) . '['
. $1 . ']' . " 1\n";
                                    }
                              } else {
                                    if($vector{"$port{'name'}$port{'portnum'}"} eq
"0") {
                                          print "assert " . lc($port{'name'}) . "
0\n";
                                    }
elsif($vector{"$port{'name'}$port{'portnum'}"} eq "1") {
                                          print "assert " . lc($port{'name'}) . "
1\n";
                                    }
                              }
                        } else {
                              # input
                              if($port{'portnum'} =~ /\[(\d+):(\d+)\]/) {
                                    ($start, $end) = ($2, $1);
                                     for(my $i = $end; $i >= $start; $i--) {

      if(substr($vector{"$port{'name'}$port{'portnum'}"},$end - $i,1) eq "0") {
                                                print "l " . lc($port{'name'}) .
'[' . $i . ']' . "\n";
                                          }
elsif(substr($vector{"$port{'name'}$port{'portnum'}"},$end - $i,1) eq "1") {
                                                print "h " . lc($port{'name'}) .
'[' . $i . ']' . "\n";
                                          }
                                    }
                              } elsif($port{'portnum'} =~ /\[(\d+)\]/) {
                                    if($vector{"$port{'name'}$port{'portnum'}"}  eq
"0") {
                                          print "l " . lc($port{'name'}) . '[' . $1
. ']' . "\n";
                                    }
elsif($vector{"$port{'name'}$port{'portnum'}"} eq "1") {
```

```
                                          print "h " . lc($port{'name'}) . '[' . $1
. ']' . "\n";
                                    }
                              } else {
                                    if($vector{"$port{'name'}$port{'portnum'}"} eq
"0") {
                                          print "l " . lc($port{'name'}) . "\n";
                                    }
elsif($vector{"$port{'name'}$port{'portnum'}"} eq "1") {
                                          print "h " . lc($port{'name'}) . "\n";
                                    }
                              }
                        }
                  }
            }
      }


      # Outputs
      foreach my $k (sort keys %{$VCD{'ports'}}) {
            my %port = %{$VCD{'ports'}{$k}};
            # TODO: This should be a list
            # if($port{'name'} =~ /(clk|reset)/i) {
            #     next;
            # }
            if($vector{'reset'} eq '1' or $vector{'resetb'} eq '0') {
                  next;
            }
            if($port{'type'} eq 'output') {
                  # output
                        if($port{'portnum'} =~ /\[(\d+):(\d+)\]/) {
                              ($start, $end) = ($2, $1);
                               for(my $i = $end; $i >= $start; $i--) {

      if(substr($vector{"$port{'name'}$port{'portnum'}"},$end - $i,1) eq "0") {
                                          print "assert " . lc($port{'name'}) . '['
. $i . ']' . " 0\n";
                                    }
elsif(substr($vector{"$port{'name'}$port{'portnum'}"},$end - $i,1) eq "1") {
                                          print "assert " . lc($port{'name'}) . '['
. $i . ']' . " 1\n";
                                    }
                              }
                        } elsif($port{'portnum'} =~ /\[(\d+)\]/) {
                              if($vector{"$port{'name'}$port{'portnum'}"}  eq "0")
{
                                    print "assert " . lc($port{'name'}) . '[' . $1
. ']' . " 0\n";
                              } elsif($vector{"$port{'name'}$port{'portnum'}"} eq
"1") {
                                    print "assert " . lc($port{'name'}) . '[' . $1
. ']' . " 1\n";
                              }
                        } else {
                              if($vector{"$port{'name'}$port{'portnum'}"} eq "0") {
                                    print "assert " . lc($port{'name'}) . " 0\n";
                              } elsif($vector{"$port{'name'}$port{'portnum'}"} eq
"1") {
                                    print "assert " . lc($port{'name'}) . " 1\n";
                              }
                        }

            }
      }

      print "s\n";

      print "\n";
}

sub writeCMD() {
      open(SAVEOUT, ">&STDOUT");
      open(STDOUT, ">$basename.cmd") or die("Can't open $basename.cmd for
writing");

      # Print header
      print_header();

      # Print vectors here
      printVectors();

      close(STDOUT);
      open(STDOUT, ">&SAVEOUT") or die("Can't open original STDOUT");
}

sub main() {
      # Parse command line arguments
      $numArgs = $#ARGV + 1;
      $vcdFilename = $ARGV[$#ARGV];
      $basename = basename($vcdFilename, ".vcd");

      if($numArgs != 1) { usage() };

      eval('require "' . "$basename.conf" . '";') or die "Couldn\'t load config
file $basename.conf";

      # Doubled because we do two steps per input/output pair.
      $irsimstepsize = $irsimspeed/4;

      parseVCD();
      writeCMD();
}

sub usage() {
      print STDERR "Usage: ./vcd2cmd.pl input.vcd\n";
      exit;
}
```

Appendix: 6502 Emulator

To help with the development of the processor, several emulators were used. The primary emulator used in the design was the appleiigo open source emulator (http://code.google.com/p/appleiigo/). This emulator currently runs on Mac OS X, though the authors intend to make it cross-platform in the future. The emulator runs any currently available ROM image, and served as the basis for our path testing efforts.

Once a ROM image was acquired for the Apple IIe using the ADTpro disk transfer utility and a pair of audio cables, the image was tested both on the Apple II emulator and the RTL in ModelSim. A Verilog testbench was created that compares instruction loads with the instruction loads as recorded from the modified Apple emulator. This allowed several bugs to be identified and fixed, though since the Apple II hardware was never fully implemented in RTL (IO systems were not modelled), the behavior has not been identical.

ADTpro has been used to transfer data to and from the Apple II using the cassette adapter, though this has proven challenging. In the future, a Super Serial card would be more appropriate to the task.