Compact Integrated Processor

EE498 Report

Senior Design II

Jared Hayes

Nick Repetti

Jason Silic

Faculty Advisor: Dr. R. Jacob Baker

12 May 2015

Table of Contents

1. Introduction

1.1 Introduction

MIPS (Microprocessor without Interlocked Pipeline Stages) is a popular implementation of a reduced instruction set computer (RISC). MIPS architectures are typically 32-bit, but 64-bit versions have been developed in more recent years. It became a widely used processor for embedded systems design and has been implemented in many consumer electronics products. These included routers, devices that ran Windows CE, and Sony video game consoles. Due to its popularity in modern electronics in the 1990's, it has often been studied in computer architecture courses. It is an effective, yet simple processor that fulfills the features required in a modern processor but without too much unnecessary complexity.

Our Senior Design project is a simple 8-bit CPU with onboard SRAM memory. Our instruction set is loosely based on the MIPS architecture with some significant simplifications. For example, our architecture only supports 21 simple instructions and instruction size has been reduced to two bytes. The 512 bytes of on-die memory is used to store both instructions and data.

The purpose of this project is twofold. Our first objective is to demonstrate the utility of our design. Most microprocessors of this size (the chip only has about 35,000 transistors) will use an external memory chip to store their program. The onboard memory used here results in a more compact design. The second goal is to document our design to provide an example for future students. Since our processor will be much easier to interpret and apply than most modern microprocessors, students can use it to create simple projects as an entry into embedded systems.

2. System Overview

2.1 Instruction Set

Our processor uses a modified instruction set with 16-bits per instruction, instead of the typical 32-bit instruction set. This system was chosen because our onboard SRAM is only 512 bytes, so conserving memory is very important. Also, the large instruction size is mostly to accommodate large register files (32 registers is typical) and large memory spaces. We only need nine bits to specify a memory location, so even our jump instructions can store the opcode (4 bits), register to compare (3 bits) and absolute address (9 bits) for a total of 16 bits. There was really no need for a larger instruction size which would have just increased the complexity of our design.

The first thing that should be understood is the various instruction formats. In the below diagrams, the number represent the bit at the end of the field. For example, the opcode is the four most significant bits. In a two byte instruction bit are numbered from 0 (LSB) to 15 (MSB). The opcode

therefore has bits numbered from 12 to 15. The text in **bold** indicates the name of the field, such as **opcode** or **L6** (a literal value).

Arithmetic Instruction

15 **opcode** 12	11 **r0** 9	8 **r1** 6	5 **r2** 3	2 **flags** 0

Arithmetic Instruction with Literal (L6)

15 **opcode** 12	11 **r0** 9	8 **L6** 3	2 **flags** 0

Control Instruction

15 **opcode** 12	11 **r0** 9	8 **L9** 0

Data Instruction (bit 8 is unused)

15 **opcode** 12	11 **r0** 9	8	7 **L8** 0

Shift Instruction

15 **opcode** 12	11 **r0** 9	8 **r1** 6	5 **unused** 3	2 **L3** 0

The instructions are fetched in two separate read. There are 21 instructions accessed through a 4-bit opcode and 3-bit flag field on arithmetic and IO instructions. For example, opcode 0000 is an arithmetic instruction, with the particular instruction (ADD, SUB, XOR, etc.) determined by the bits in the Flags field.
The encoding for the various instructions is as follows:

Table 1

opcode	instruction	opcode	instruction
0000	ARITH	1000	JNE
0001	LOAD	1001	JGE
0010	SHL	1010	[not used]
0011	SET	1011	NOP
0100	IO	1100	JE
0101	STORE	1101	JG
0110	SHR	1110	ROR
0111	STOP	1111	STOP*

*Note that the STOP instruction has two opcodes used for it.

The ARITH instruction breaks down as follows, depending on the Flags field

flags	instruction	flags	instruction
000	ADDI	100	AND
001	ADD	101	NOT
010	SUB	110	XOR
011	OR	111	[not used]

The IO instructions depend on the least significant bit of the Flags field.

If it is 0, the instruction is IN, otherwise the instruction is OUT.

2.1.1. Instruction Description

The basic operation of the instructions is described here. The following section will describe how the various fields in the 16-bit instruction are used.

The instructions are presented below with this format:
r0, **r1**, **r2** are 3-bit fields that specify one of eight registers.

L9 is a nine bit literal used to specify a jump address.
L6 is a six bit literal used for the ADDI instruction.
L8 is an eight bit literal used for setting a register to a value
L3 is a three bit literal to control the amount of shifting or rotation for SHL, SHR, and ROR instructions.

Finally, note that for instructions with unused fields (e.g., the STOP instruction only uses the opcode and has twelve bits unused) these bits are don't care bits. They can be filled with any sequence of ones and zeros and the CPU will not care. They are usually padded with zeros, but this is not required.

Table 2: Instruction Description

Instruction	Encoding Format	Notes
ADD r0, r1, r2	Arithmetic Instruction	r0 gets the result of r1 + r2
ADDI r0, L6	Arithmetic Instruction with Literal	r0 gets the result of r0 + L6 (sign extended)
SUB r0, r1, r2	Arithmetic Instruction	r0 = r1 - r2
OR r0, r1, r2	Arithmetic Instruction	r0 = r1 | r2 (bitwise OR operation)
AND r0, r1, r2	Arithmetic Instruction	r0 = r1 & r2 (bitwise AND operation)
NOT r0, r1	Arithmetic Instruction	r0 = ~r1 (bitwise inversion)
XOR r0, r1, r2	Arithmetic Instruction	r0 = r1 ^ r2 (bitwise XOR operation)
IN r0	Arithmetic Instruction	r0 get the value placed on the IO input pins
OUT r0	Arithmetic Instruction	The value in r0 is placed into the output register, and appears on the output IO pins
JE r0, L9	Control Instruction	Jump to address L9 if r0 is equal to zero.
JNE r0, L9	Control Instruction	Jump to address L9 if r0 is not equal to zero.
JG r0, L9	Control Instruction	Jump to address L9 if r0 is greater than zero (signed comparison)
JGE r0, L9	Control Instruction	Jump to address L9 if r0 is greater than or equal to zero (also signed)
LOAD r0, L9	Control Instruction	Load data at address L9 into register r0
STORE r0, L9	Control Instruction	Store contents of register r0 at address L9
SET r0, L8	Data Instruction	Store the value L8 into register r0
STOP	Any	Stop the execution of the current program, until a reset signal. Only the opcode is used for STOP and NOP instructions. The other twelve bits are unused.
NOP	Any	No operation

SHL r0, r1, L3	Shift Instruction	Shift register r1 left by L3, store result in r0. [Shift in zeros]
SHR r0, r1, L3	Shift Instruction	Shift register r1 right by L3, store result in r0. [Shift in zeros]
ROR r0, r1, L3	Shift Instruction	Rotate the bits in r1 to the right by L3 positions, store in r0

2.1.2. Instruction Encoding Examples

The above tables and lists may be a bit confusing, so the following examples will hopefully clear up any confusion. The complete process of forming an instruction is considered and explained, step by step.

Example 1

SHR R4, R3, #6

Here R4 indicates register four (don't confuse with the register fields above such as r0, r1, and r2). R3 indicates register 3 and #6 is a literal value. The value in register 3 is shifted right six positions and the result is stored in R4.

From Table 1 above we discover that the opcode for SHR is 0x6. The first field is 0x4 and the first bit of field r1 is 0. The upper eight bits of this instruction is therefore 0x68, or 104 in decimal. The first two bits of the lower byte are '11', from register three. The next three bits we leave as zeros because they are unused. The last three bits are 011, which is our literal '6.' The lower byte is therefore 0xC6, which is 198 in decimal.

The complete instruction is therefore 0x68C6. If you were writing a program on an Arduino like we did to load the SRAM, the first byte loaded is the lower byte, so your data array would be like this, excluding the other instructions:

int data[16] = {.. , .. , 198, 104, .. , .. , ….};

Example 2

ADDI R7, #62

Note that this instruction subtracts two from R7 and stores the result back in R7. Remember that the literal for this instruction is sign-extended, so 62 is binary 111110, which is two's complement for negative two.

Table 1 shows the opcode is 0x0. Table 2 tells us to use the Arithmetic Instruction with Literal format. Field **r0** is 111 (R7) and the literal **L6** is 111110. Finally, the flags field should be 000 for the ADDI instruction. The upper byte is therefore 0x0F, or decimal 15. The lower byte is 0xF0, which is 240.

The complete instruction is 0x0FF0, which is loaded into memory as 240, 15 (lower byte first!).

Example 3

OUT R2

This is an IO instruction with opcode 0x4. The Arithmetic Instruction format is used, but only the first register field is used. The flags field has to be 001 for the OUT instruction and 000 for the IN instruction. The upper byte is therefore 0x44 = 68. The lower byte is 0x01 = decimal 1.

The complete instruction is 0x4401, which is stored into memory as 1, 68.

Example 4

JG R3, #420

The opcode for JG is 0xD. The register field is 011. The address #420 is 1 1010 0100 in binary. The upper byte is therefore 0xD7 which is 215. The lower byte is 0xA4 which is 164 in decimal.

The complete instruction is 0xD7A4, or the decimal numbers 164, 215.

Note that the jump and control instructions use a fixed 9-bit address as their target. However, it is very easy to use the STORE instruction to write another value to the lower eight bits of this address. This feature can be used to implement subroutines. Simply write the return address to the address of a jump instruction at the end of the subroutine. When the program execution reaches this point it will simply return to the location from which it was called (assuming you wrote the correct return address).

2.2 System Architecture

The main components of the MIPS processor include:

- Control Unit
- SRAM
- ALU
- Register File

Control Unit

This processor is essentially a massive state machine. Neglecting the SRAM for a moment, the CPU's state is completely defined by the contents of the register file, ALU registers, and control unit registers. The next state is a deterministic result of the current state, and is achieved by the use of only twenty-four control signals. These control signals are generated by combinational logic which has a delay less than one-half of a clock cycle.

Some of these control signals are for the internal use of the Control Unit (CU). For example, the StepReset signal will reset the step counter, which tracks the current instruction's stage of execution. Other signals are for the ALU arithmetic logic, such as the three ALU control signals.

The control unit does have some additional responsibilities for the shifting instructions. The barrel shifter in the ALU can only shift data one bit position per clock cycle. To implement longer shifts, a small three-bit counter is necessary.

SRAM

The SRAM is of a standard six transistor (6T) design. A simple precharge is applied to the bitlines when the clock signal is high to prevent a low voltage bitline from flipping the datum stored in a cell. Loading data into the SRAM is accomplished by a specialized circuit that aids in the sequential writing of data to the SRAM.

Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit is responsible for all of the Arithmetic and Logic functions that most of the operations rely on. The ALU can perform multiple operations which will be discussed in a later section. Data is clocked into Registers A and B from their respective buses and the appropriate action is performed based on the input signals from the Control Unit. The ALU also handles the calculations for the jump instructions.

Register File (RF)

The register file is responsible for storing data in registers specified by the instruction register for use by the ALU. Since this is an 8-bit processor it reads in 3-bit opcodes and decodes them into the clocks of each register. This register file can read two registers and write to a single register at once. Two 3-to-8 decoders help to drive multiplexing for each of the two buses leading out of it. The bus A decoder is also responsible for gating the clock of the registers to select which register to write to. Instead of using multiplexers to determine which registers to output to the buses, 8-bit transmission gates powered by the decoders are used for each register. There are two sets of the 8-bit transmission gates in total, one for each bus.

We present a top-level schematic with the various parts of the processor labeled on the next page. Note the precharge circuit used to always define the voltage on BusA. This is important because when all the tri-state buffers are in the Hi-Z state, the voltage on the bus could float, possibly leading to contention

current in inverters connected to the bus. This could lead to failure of the chip, or fluctuations in the power supply voltage.

2.3 Functional Analysis and Requirements

Our processor is designed and simulated using Cadence Virtuoso design software with Spectre and Ultrasim simulation plugins. The processor was designed from scratch on the transistor level and fabricated on a chip using the MOSIS On Semiconductor 500 nm rules and the C5 CMOS process, which contains 2 polysilicon layers and 3 metal layers. Therefore, all digital logic and components will be custom made in order to fit the specifications of our project and the MOSIS design rules. Cadence allows us to design the chip layout according to those rules and will make checks to ensure that our layout stays consistent with them. The fabrication process takes considerable time and resources; therefore, we must ensure that our design is properly simulated so that our fabricated chip works as intended in our design. We use Spectre and Ultrasim to simulate the schematics of all of our components to ensure proper operation. Then, Cadence checks that our layouts exactly match the schematics in order to adhere to our simulation results.

2.4 Trade-offs

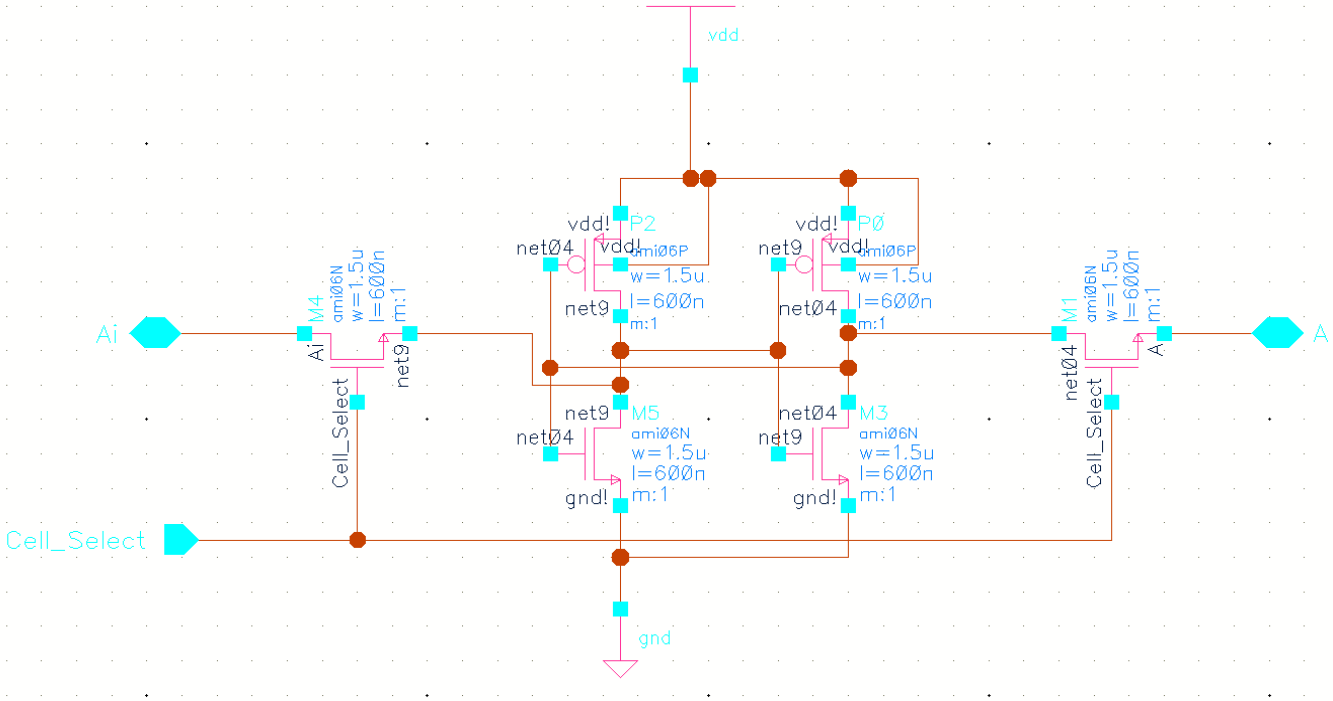
We had to consider many trade-offs in order to simplify our processor to fit our time constraints and resources. Designing an 8-bit processor instead of a 32-bit processor obviously reduces the chip's processing power and capabilities. Using a two byte instruction length allows the user to have a simpler and more streamlined set of instructions, but at the cost of flexibility. We also chose to forego implementing a pipelined microarchitecture. A pipelined architecture would have improved our processor's execution time enormously; however, the added design complexity would not have been feasible to implement in our project, considering our time limitations. So, we implemented a multi-cycle architecture instead.

We chose to include on-board memory in our processor. This eliminates many additional off-chip components, such as an external memory chips. Unfortunately, the SRAM memory is "volatile" and loses its data after losing power. The programs must therefore be loaded onto the chip again every time the power is turned on again. This is a severe problem for the commercialization of this design. Our process technology can not be used to fabricate flash memory cells or other non-volatile forms of storage.

3. SRAM

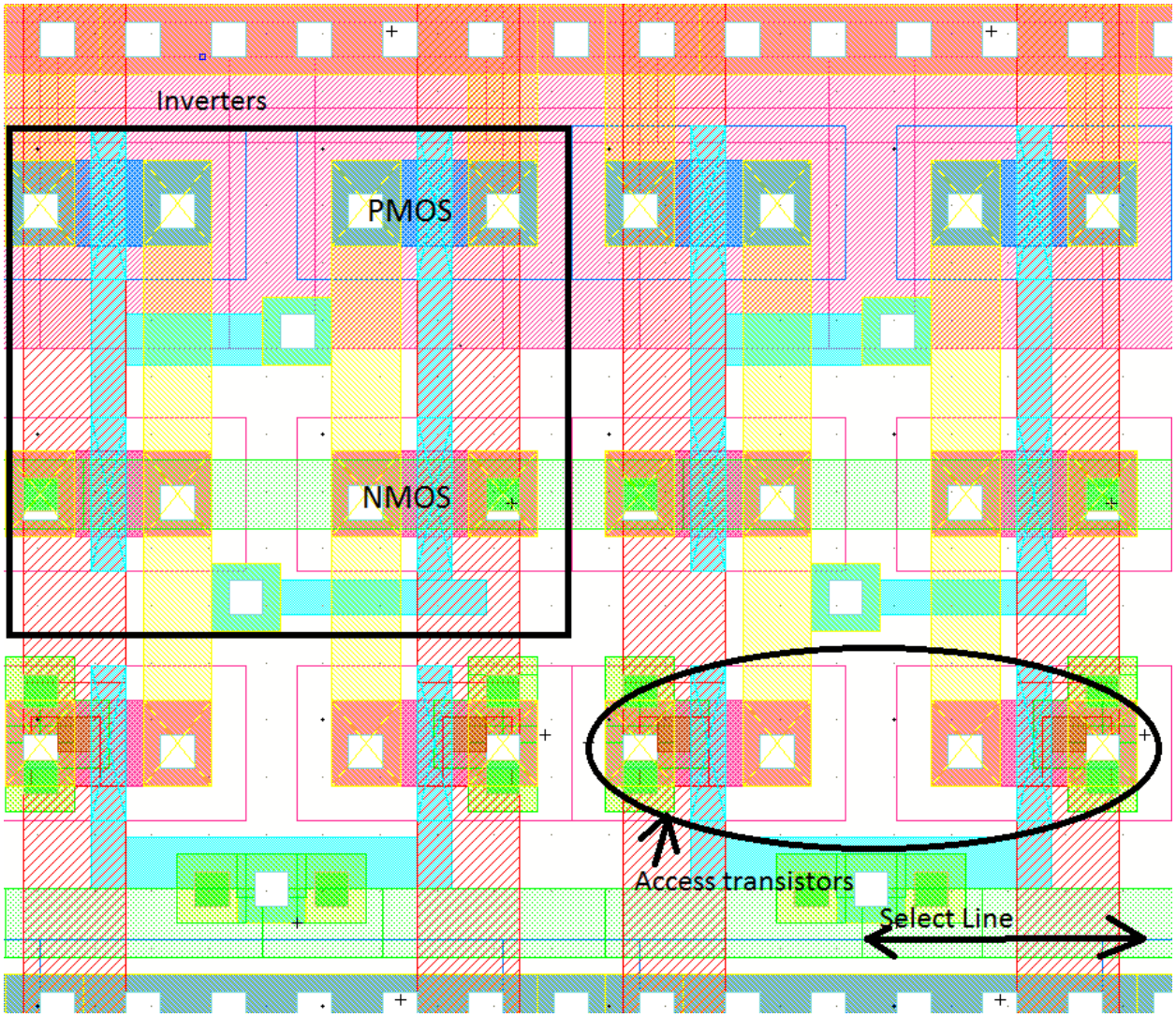
3.1 SRAM Design and implementation

The basic SRAM cell stores one bit of information. Our design is the quite common 6T (six transistors) cell. Although a four transistor cell exists, the resistors it used are difficult to fabricate in a small space and contribute to static power dissipation. The basic schematic is presented below, consisting of two cross-coupled inverters and two NMOS access transistors.



Writing data to the cell is accomplished by overpowering the internal transistors to “flip the bit.” Only writing a low voltage (actually 0V) will work, because the NMOS pass gate passes a logic ‘0’ better and the PMOS transistor is easier to overpower. Therefore, a low voltage on the “A” bitline writes a zero to the cell, and a low voltage on the “Ai” bitline writes a logic one into the cell.

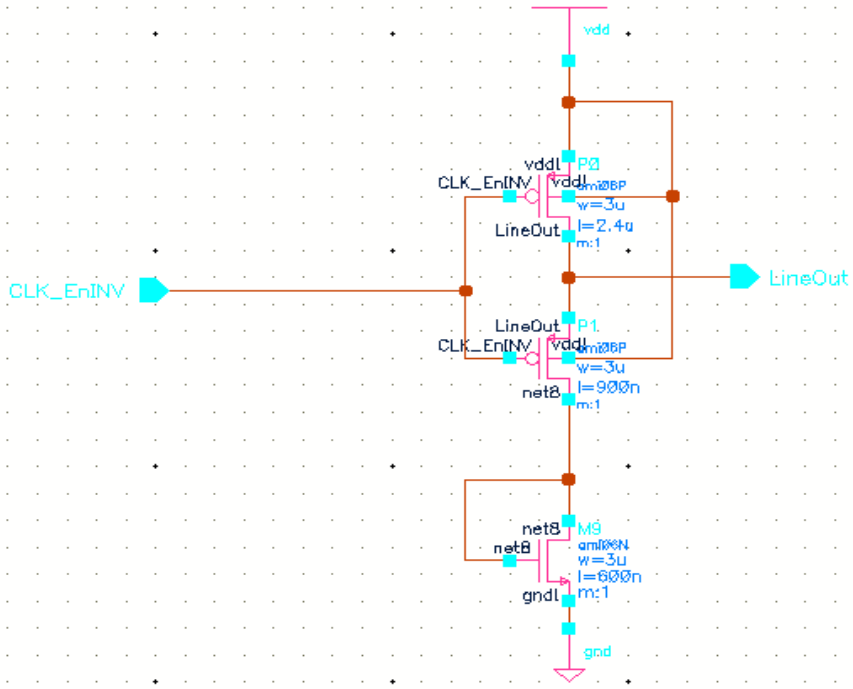
Below is the layout of the SRAM cell, with prominent features illustrated. The bitlines run vertically on Metal3, while the row select and byte select signals run horizontally on Metal2.



3.2 Precharging circuit

The original plan did not include provision for precharging the bitlines. While this would have simplified the design, simulations revealed that a bitline at ground potential could actually write data into another cell when the select line for that cell was asserted. The solution involves "precharging" the bitlines to a high enough voltage to avoid this problem. Although charging the bitlines to VDD- V_{thn} through an NMOS transistor could have worked, I decided to do a more complicated three-transistor design to get a more precise voltage. The stable output voltage of the below circuit is about 3.24V, with a 10MEG load to ground. The long length transistors limit the current that can flow, making sure the maximum sustained current is less than 100uA/via.

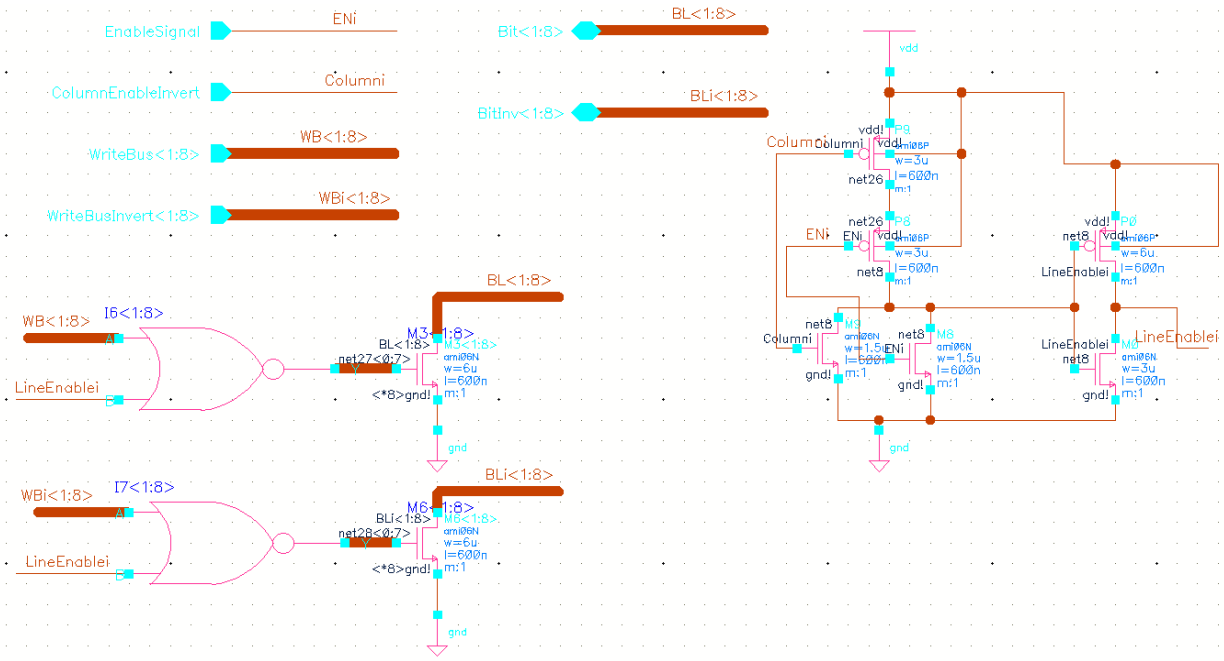
When the input is low, the PMOS transistors turn on and move the voltage on the output line toward 3.2V.



3.3 Write Driver circuit

When data is loaded into the SRAM, the data comes from an external source. In addition, data is loaded into sequential memory locations, starting at 0x000. Therefore, the address bus is connected to a nine-bit upcounter, and the write bus is connected to the external input. The external procedure for writing data to the SRAM is simple. Assert the Load Write Enable signal, and send a reset signal (Reset is active low) to reset the address counter. The values on the 8 input lines will be written to the SRAM on active clock edges. Additional schematics can be found in Appendix A.

The write circuit may also be of interest in this design. In the below schematic, powerful 6 micron wide NMOS transistors are strong enough to write data into the SRAM cells. The logic for writing is simple enough. A high signal at the gate of the NMOS will write a zero into the 6T cell. Therefore, the write signals need to be inverted first, a task accomplished by the NOR gate. When the enablei signal is low, the NOR gate is an inverter, and correct data is written into the SRAM. When the enablei signal is high, the output of the NOR gate is low, and the transistors are off. This allows read operations to proceed.



4. Control Unit

4.1 Overview

The control unit is probably the most complex part of the processor. It includes the instruction register (IR), a 3-bit downcounter for the shift instructions, a step counter to track the current stage of the instruction, and a large block of combinational logic to generate control signals for the entire CPU. The Control Unit (CU) controls not only the ALU but also the SRAM (read and write signals) and the internal state of the CU (when to reset the step counter, for example).

The first stage of the control unit determines the current instruction by decoding the 4-bit opcode and any relevant flags. The main stage of combinational logic determines appropriate control signals to generate based on the current instruction opcode, the value of the step counter, and whether the shift counter has reached zero for the shifting instructions.

4.2 Control Signal List

The following list describes the twenty-five control signals, and their purpose:

Number		Name		Purpose
0	-	PC_Update	-	Load a new value into the program counter
1	-	PC_LoadImmediate		Load data from IR into PC (for jump instructions)
2	-	IR_ReadLow		Load lower byte of IR from memory

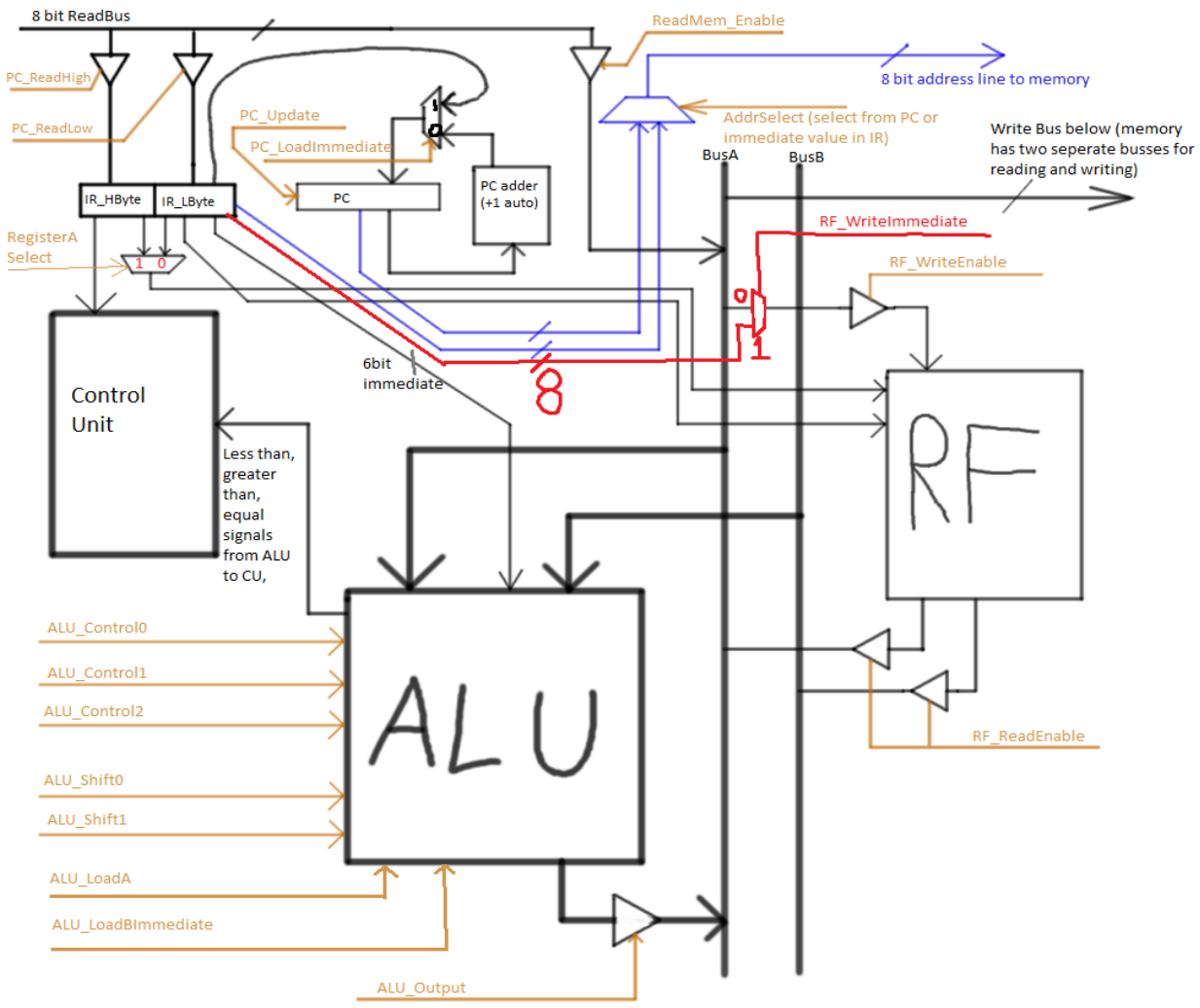
3 - IR_ReadHigh Load upper byte of IR from memory
4 - ALU_LoadA Load ALU registers
5 - ALU_LoadBImmediate Load value from IR (used for ADDI instruction)
6 - RF_ReadEnable Read data from RF to BusA and BusB
7 - ALU_Control[0] one of three ALU arithmetic control signals
8 - ALU_Control[1] one of three ALU arithmetic control signals
9 - ALU_Control[2] one of three ALU arithmetic control signals

10 - ALU_Output Place ALU register A on BusA for write to RF
11 - RF_WriteEnable Write data on BusA to RF (specified by address A to the RF)
12 - RegisterA_Select Determine which field in IR is used for RF adr. A
13 - StepReset Reset the step counter to begin the next instruction cycle
14 - Input_BusA Read data to BusA from external input pins
15 - Output_BusA Place data on BusA to output register for external IO
16 - Addr_Select Get address for SRAM from IR instead of PC (for load and store instructions)
17 - ReadMem_Enable Read memory contents onto BusA (for LOAD instruction)
18 - WriteMem_Enable Write memory from BusA (for STORE instruction)
19 - ALU_Shift[0] Control signal for ALU barrel shifter

20 - ALU_Shift[1] Control signal for ALU barrel shifter
21 - Pause_Step Pause the step counter for the shifting instruction (these instruction take a variable amount of time to complete, depending on the number of times the shift is performed)
22 - GateClock Stop the chip for the STOP instruction (prevent further action)
23 - ALU_LoadCounter Load the Shift counter from the Flags field of the IR
24 - RF_WriteImmediate Write to the RF from the immediate value in IR, not BusA (used for SET instruction)

4.3. Control Signal Block Diagram

The following diagram contains a visual block diagram that describes many of the control signals. Some parts, such as the IO circuitry, are missing.

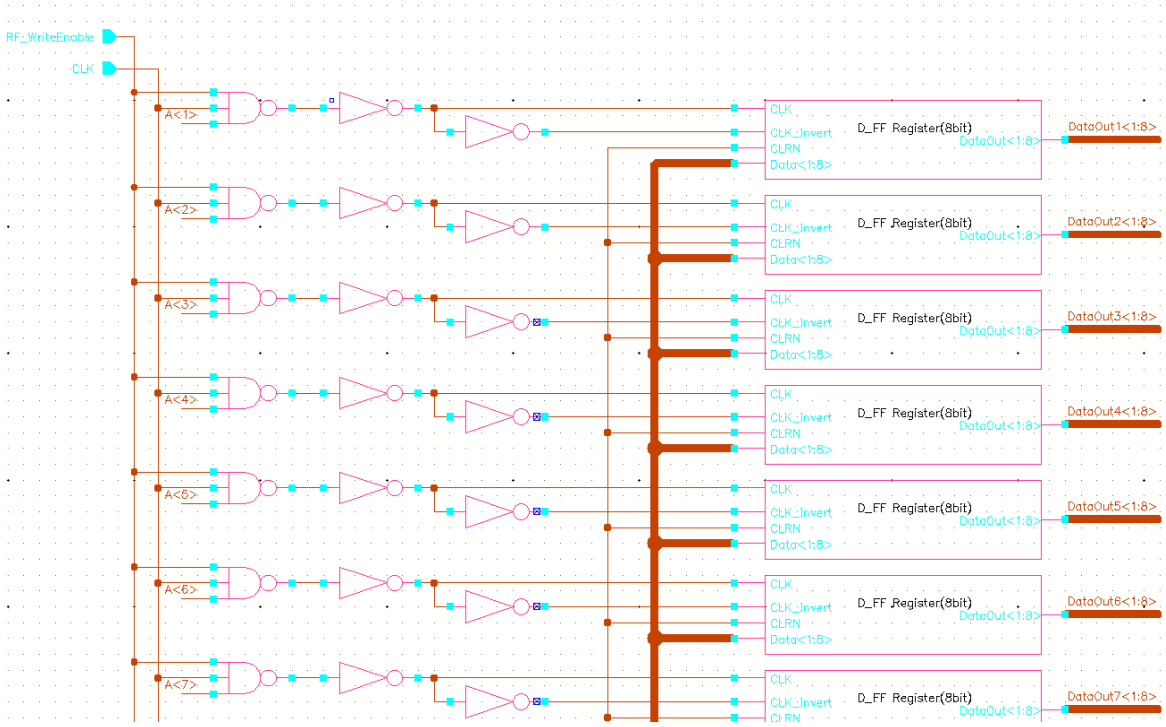


5. Register File

5.1 Design and Implementation

Registers

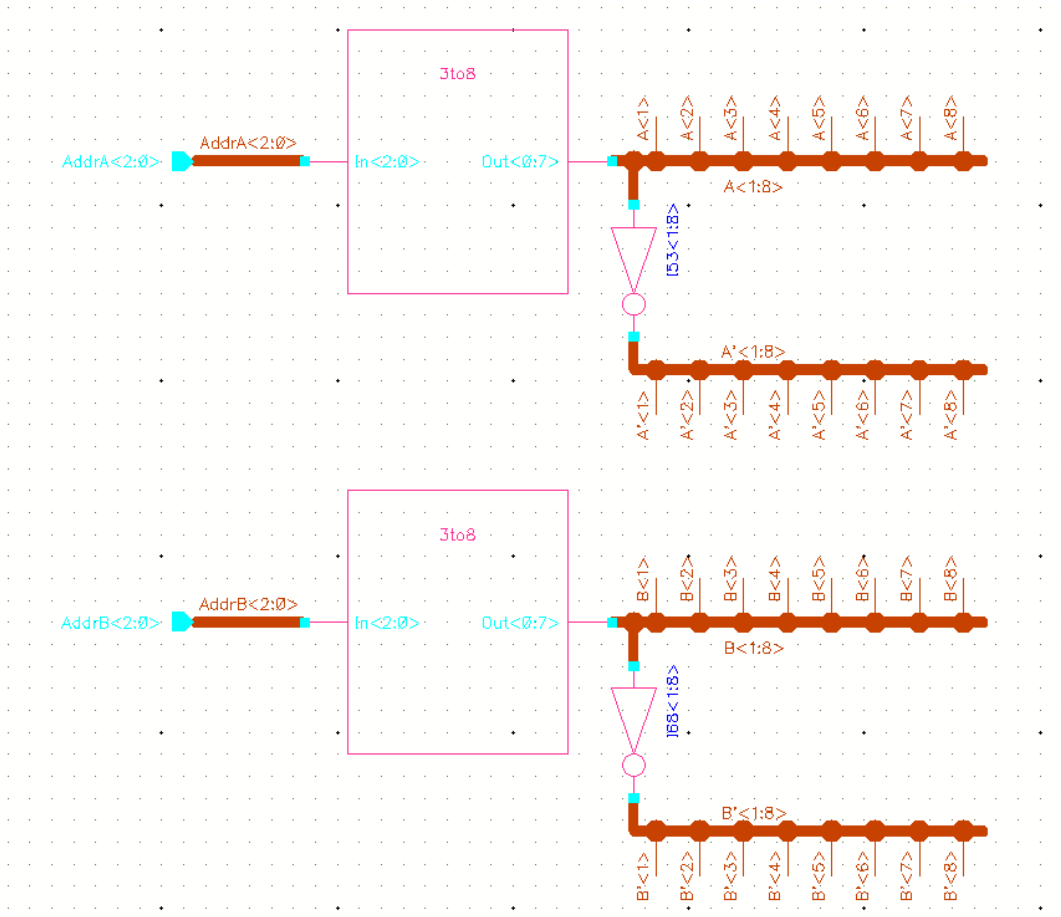
Eight registers composed of 8 D flip flops each, which are in turn composed of 18 transistors each. The majority of the transistors in the register file are of size 6 microns for each PMOS and 3 microns for each NMOS.

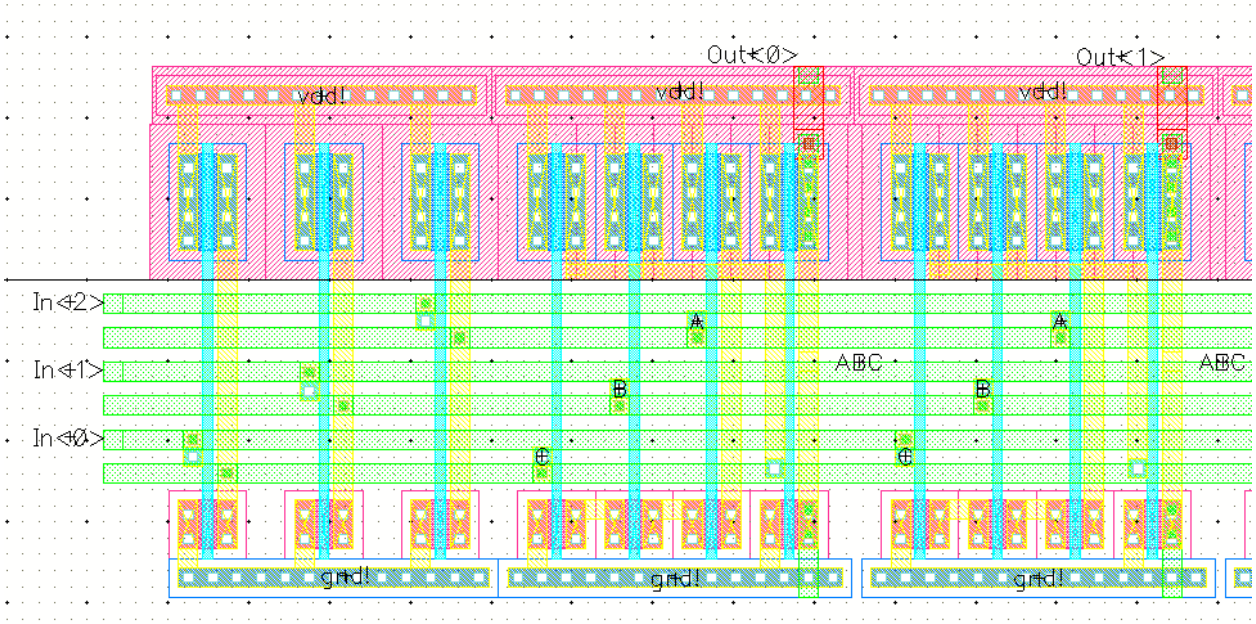


3-to-8 Decoders

There are two decoders, one for address A and another for address B. The outputs of each get sent through an inverter in order to send complimentary signals to the inverted transmission gate enables. The address A decoder is also responsible for gating the clocks of each register to enable writing to that register, as only the registers specified by address A will be written to.

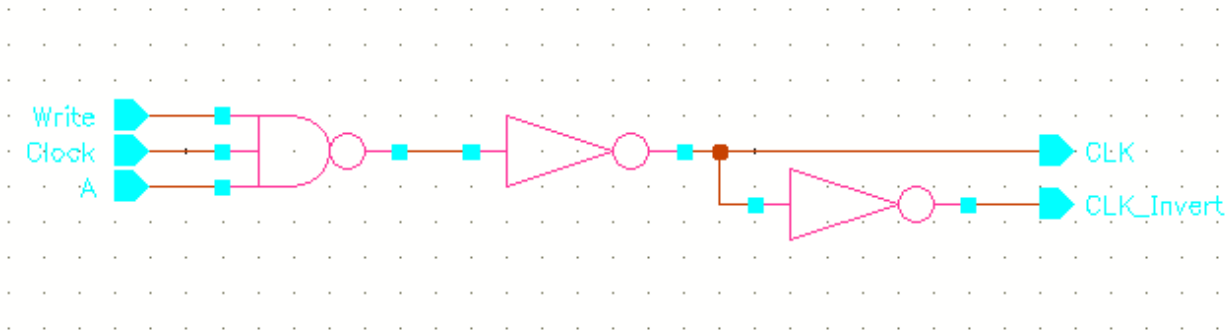
Below is the layout of the first two outputs of the decoder. The bitlines run vertically on metal 3 and the input lines run horizontally on metal 2.





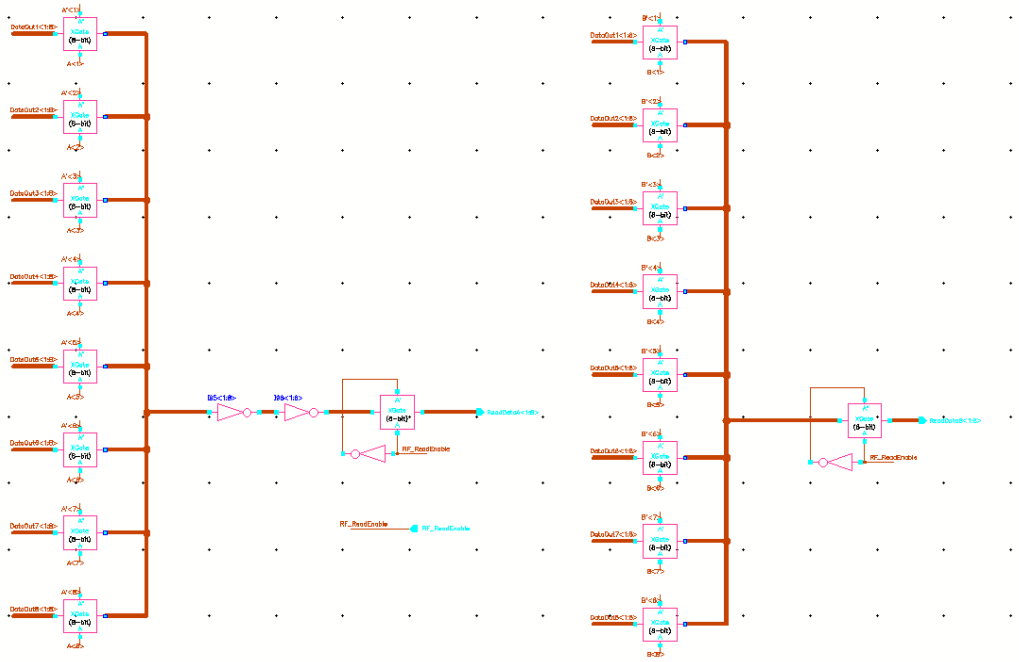
Clock Gating Logic

Gets sent into the CLK and CLK_Invert pins of a register to enable writing.



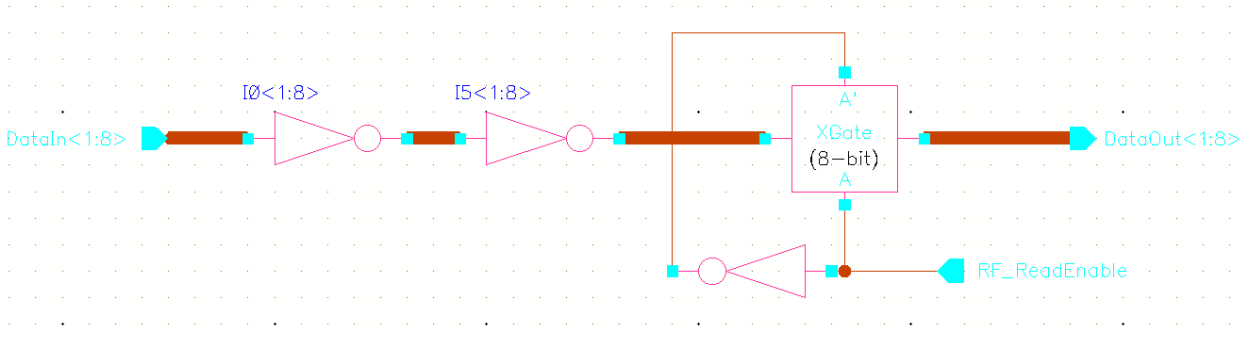
Transmission Gates & Output Buffers

The data from each register is sent into an 8-bit transmission gate (one 1-bit transmission gate for each D Flip-Flop in a register) and then out through an output buffer onto one of the address buses. The decoder outputs are sent into the transmission gate enables to decide which register's data gets sent to the address bus.



Output Buffer

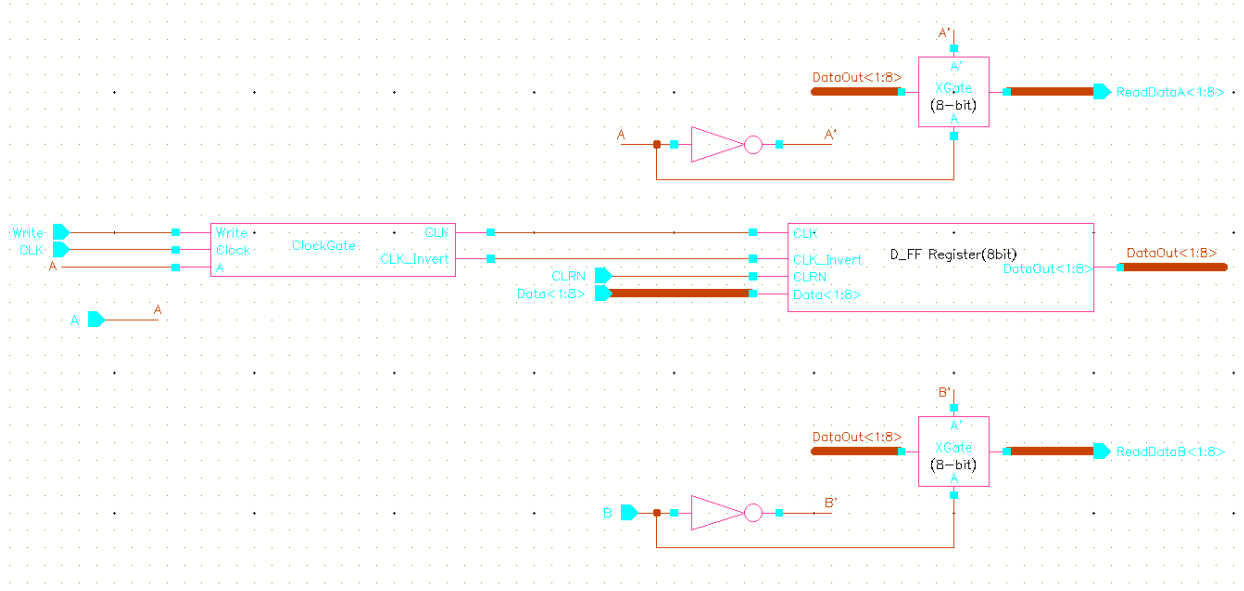
6-by-3 micron inverters followed by 12-by-6 micron inverters are used to drive the output bus. The transmission gate is turned on to read from the bus by RF_ReadEnable sent by the control unit.

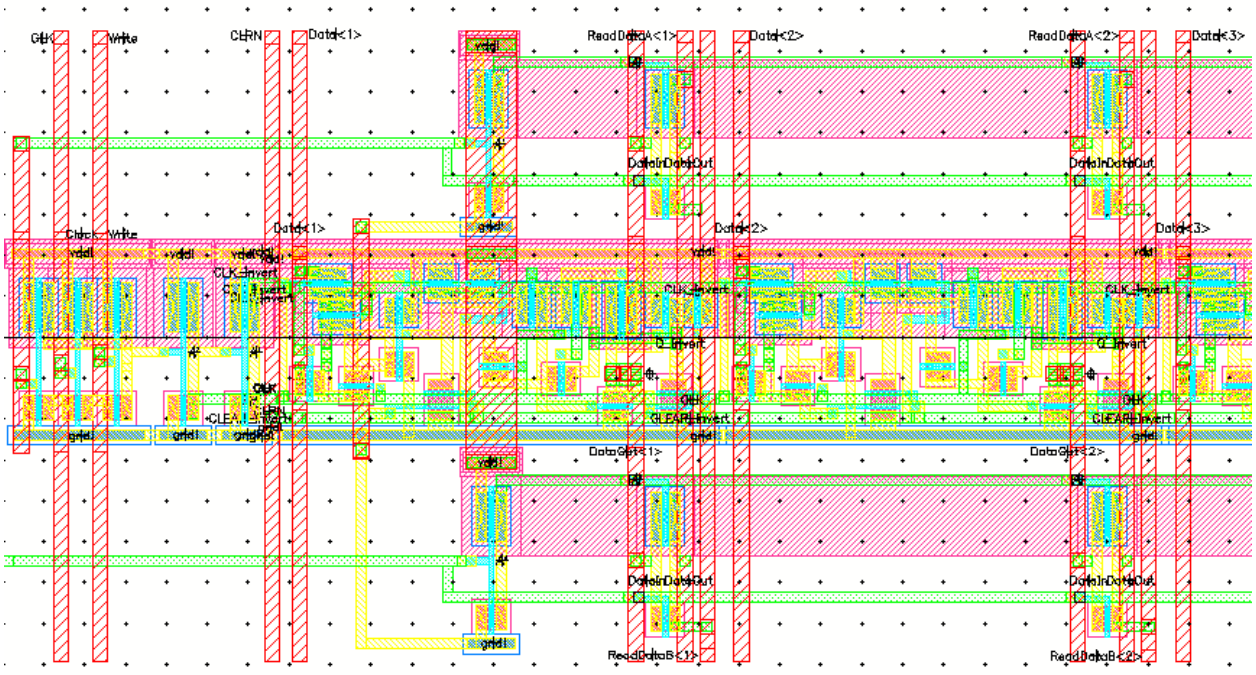


Single Register File Cell

The signals Write_Enable, CLK, and individual outputs of decoder A gate the clock into the register. DataOut<1:8> gets sent to the 8-bit transmission gates, which are enabled by inputs A and B from the decoders.

The layout of the first two bits of the register cell is below. The bitlines and control signals run vertically through the cell on metal 3 and transmission gate enables run horizontally on metal 2.





5.2 Alternatives and Trade-Offs

A register file is usually implemented with multiplexers. In this case, two 8 to 1 multiplexers would have been used to choose which register's data would be output onto the buses. However, by replacing the multiplexers with eight 8-bit transmission gates for each bus, we were able to save 256 transistors and the data only has to go through one level of logic instead of three.

6. ALU

6.1 Design and Implementation

The major design choice for the ALU was to decide which operations we wanted it to perform. We decided that we wanted to include a Barrel Shifter in the design, which called for two types of control signals. There are five control signals which are comprised of three bits for ALU_Control and two bits for ALU_Shift. The operations that the ALU can perform include the following:

ALU_Control:

000 - AND	100 - Pass Register A
001 - OR	101 - NOT
010 - ADD	110 - XOR
011 - SUBTRACT	

ALU_Shift:

00 - Pass Data

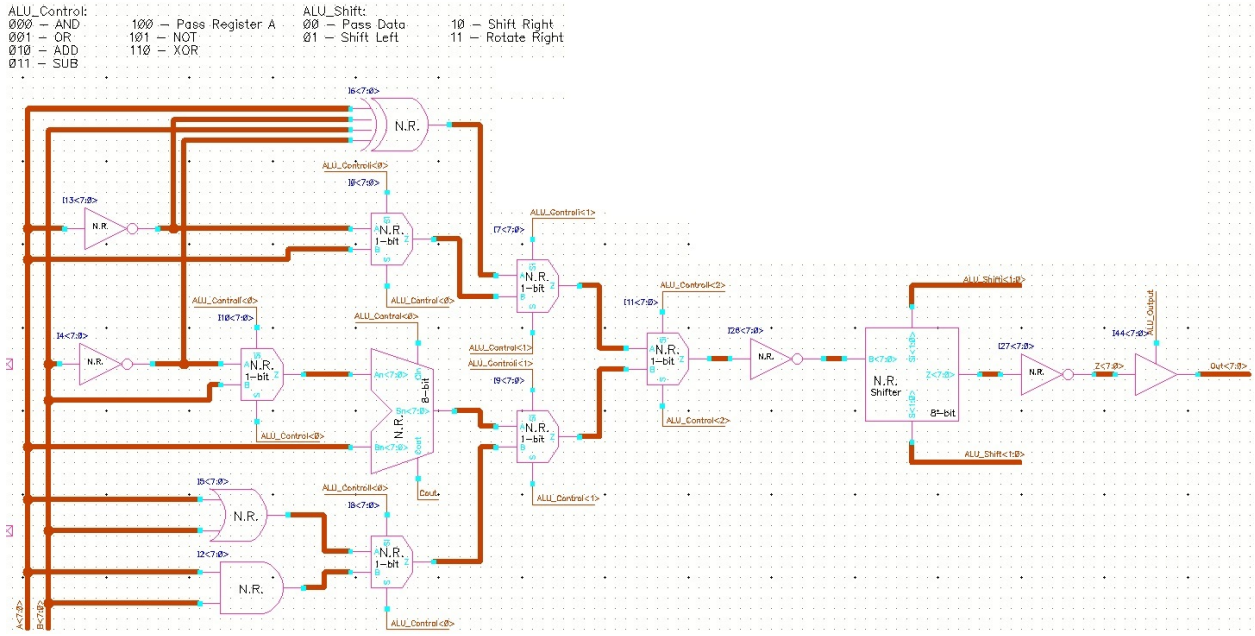
01 - Shift Left
10 - Shift Right
11 - Rotate Right

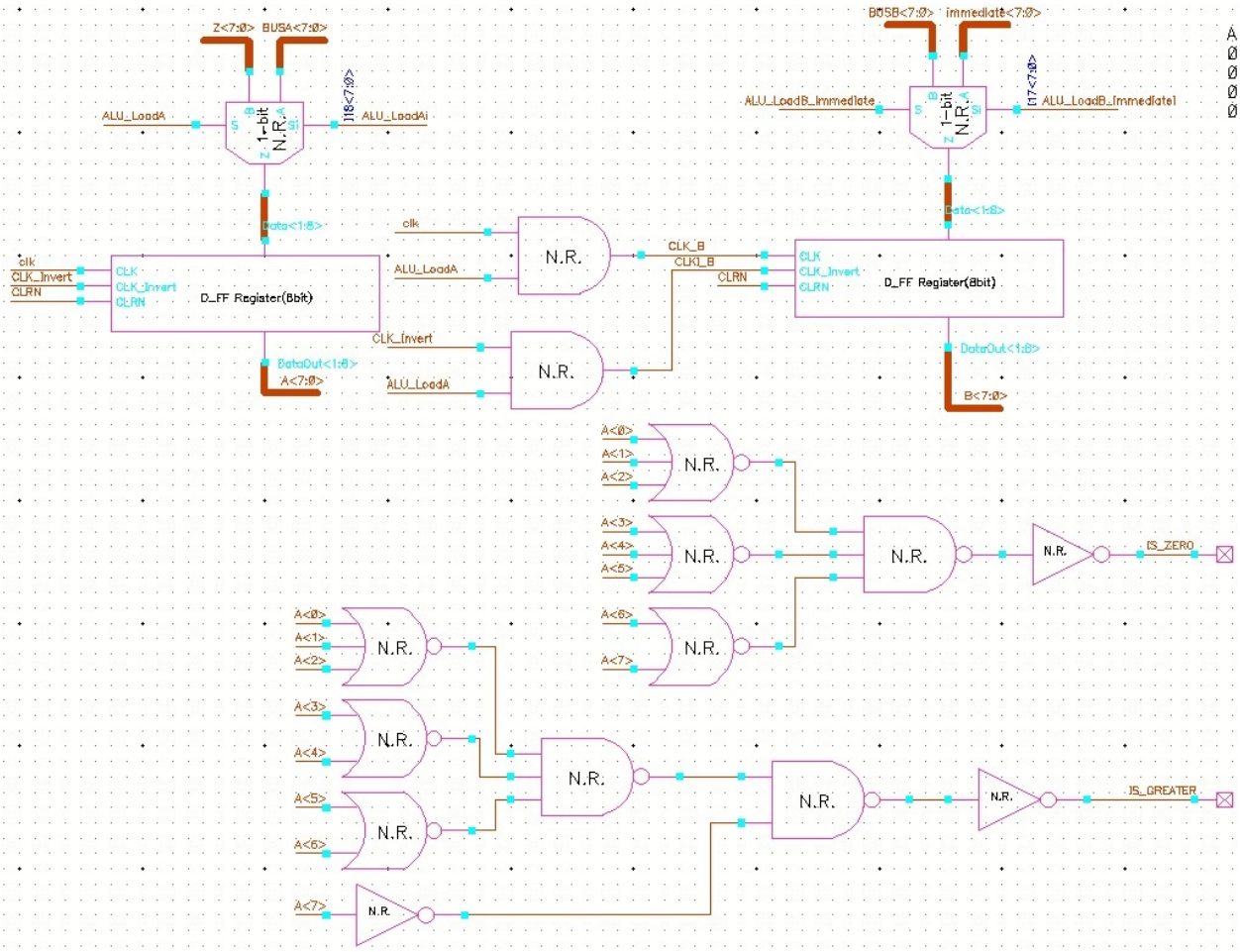
The ALU is also responsible for generating the signals for the jump instructions. It can return a true or false value for whether the value in Register A is zero, or whether it is greater than zero. Other components of the ALU include standard logic gates for AND, OR, NOT, and XOR.

The largest component of the ALU is the Full Adder. We designed an And-Or-Invert (AOI) Adder in order to save layout space. A Carry Look-ahead Adder was our first choice because it performed the operations with a minimum amount of clock cycles, but it required too large of a space to fit on our chip. The Adder has input and output signals for Input A, Input B, Carry In, Carry Out, and Sum. Each Carry In bit is generated from the previous bit's Carry Out, except for bit 0. For addition, the first bit's Carry In is 0, so the first sum is A + B + 0. Subtraction is achieved through binary two's complement where A - B = A + B' + 1. To accomplish this with our adder, the B bit is inverted, and then the first bit's Carry In is 1.

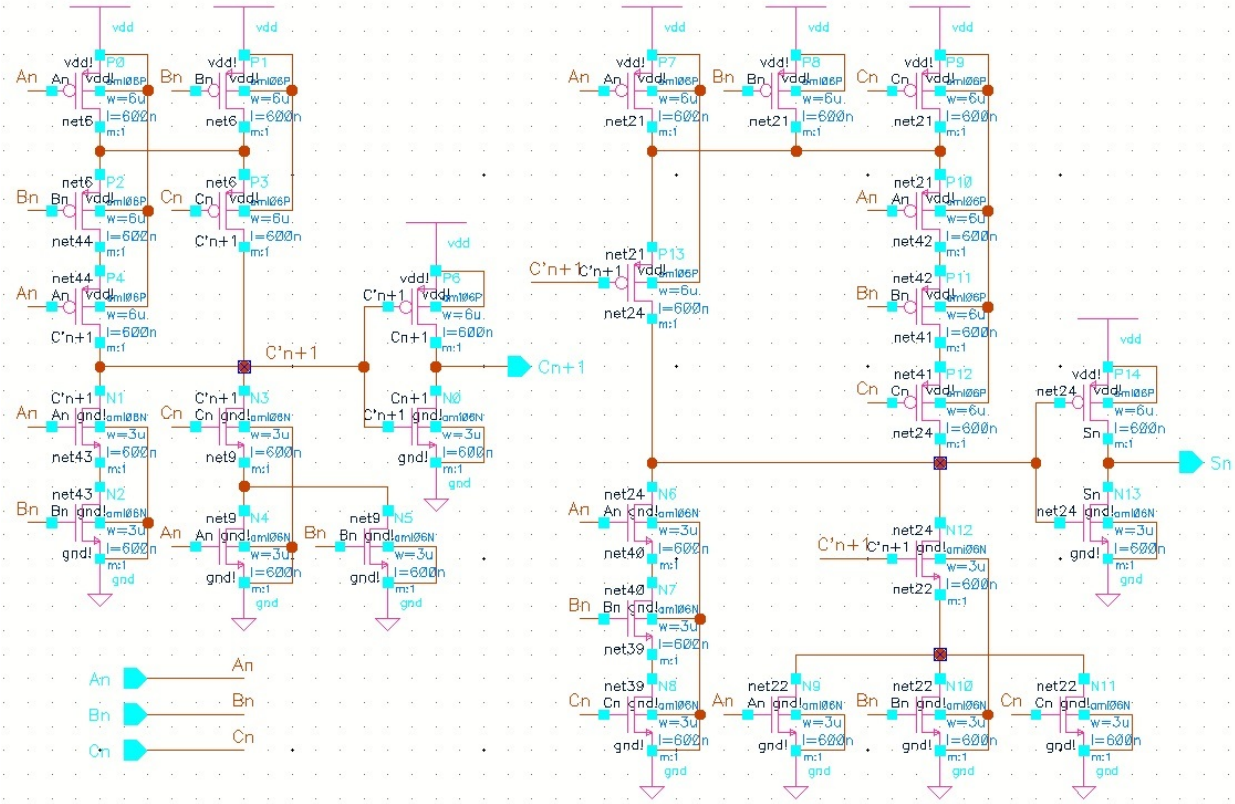
The final component of the ALU is the Barrel Shifter. Each bit is designed with a 4-1 Multiplexer which feeds in a value based on the operation to be performed. If it is a left shift, then the previous bit's value is shifted into the current bit. If it is a right shift, then the next bit's value is shifted into the current bit.

Overall ALU Schematic (Broken into 3 images)





1-bit Full Adder Schematic

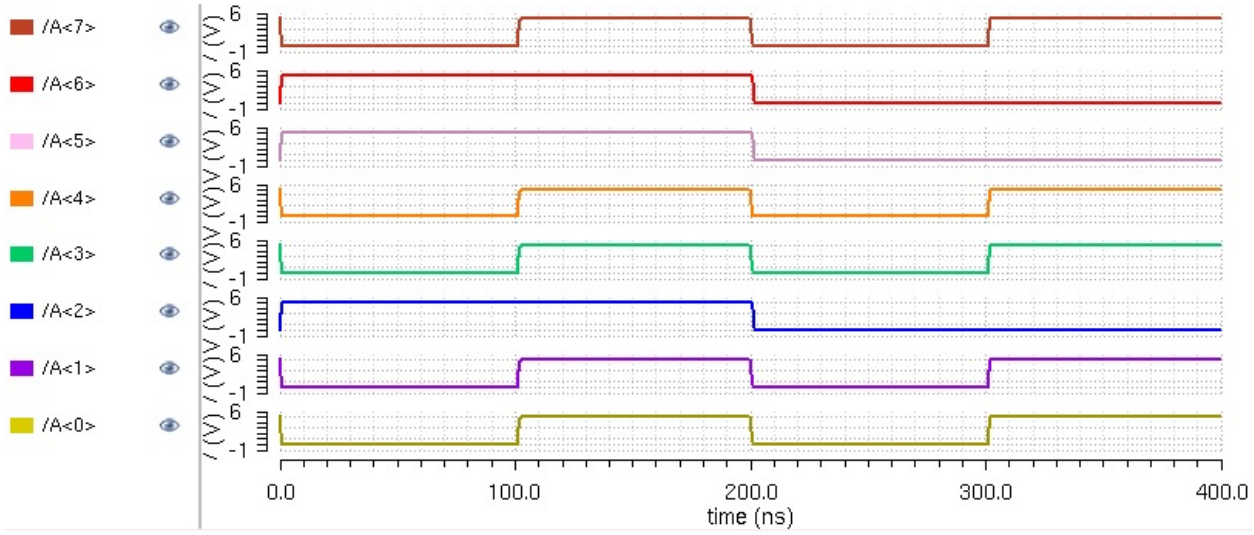


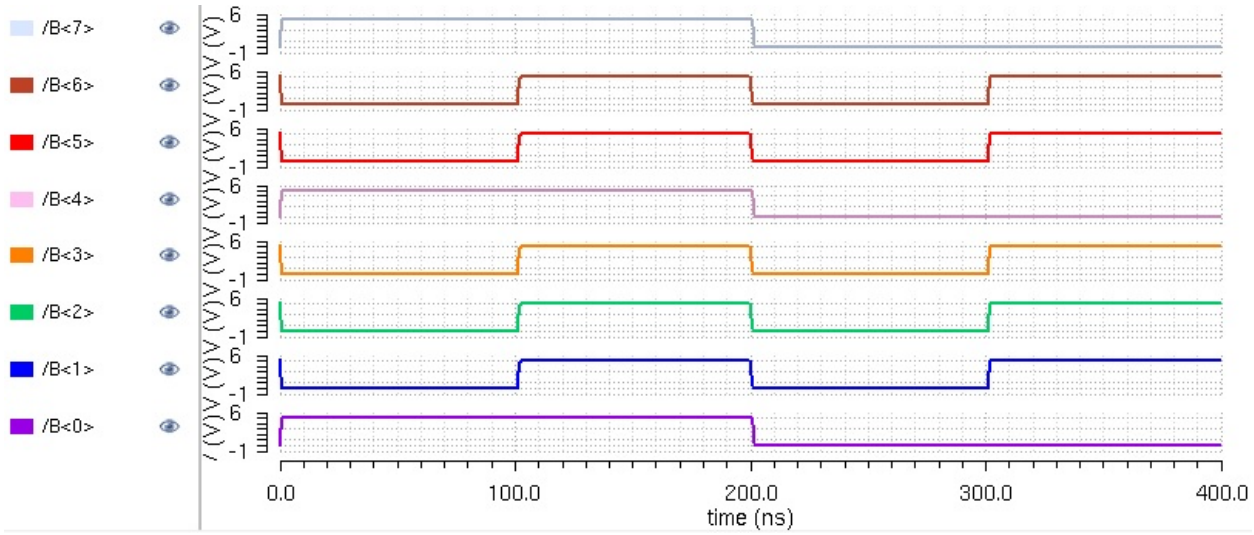
6.2 Simulations

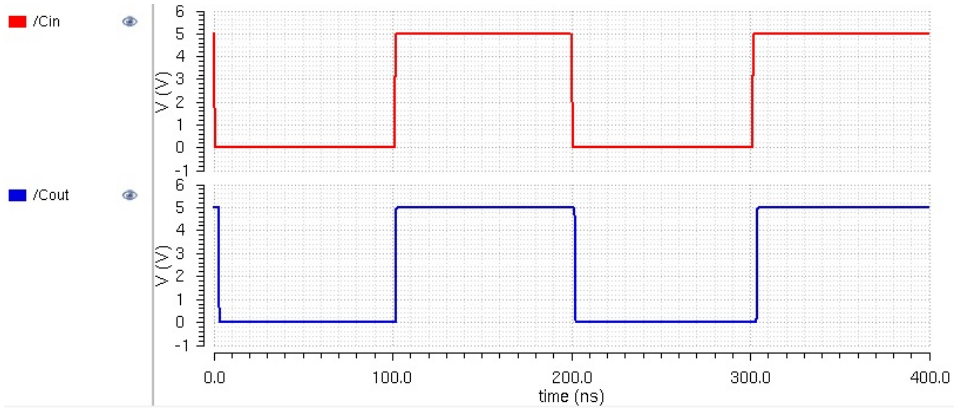
The next section will cover a few simulations demonstrating the operation of the ALU.

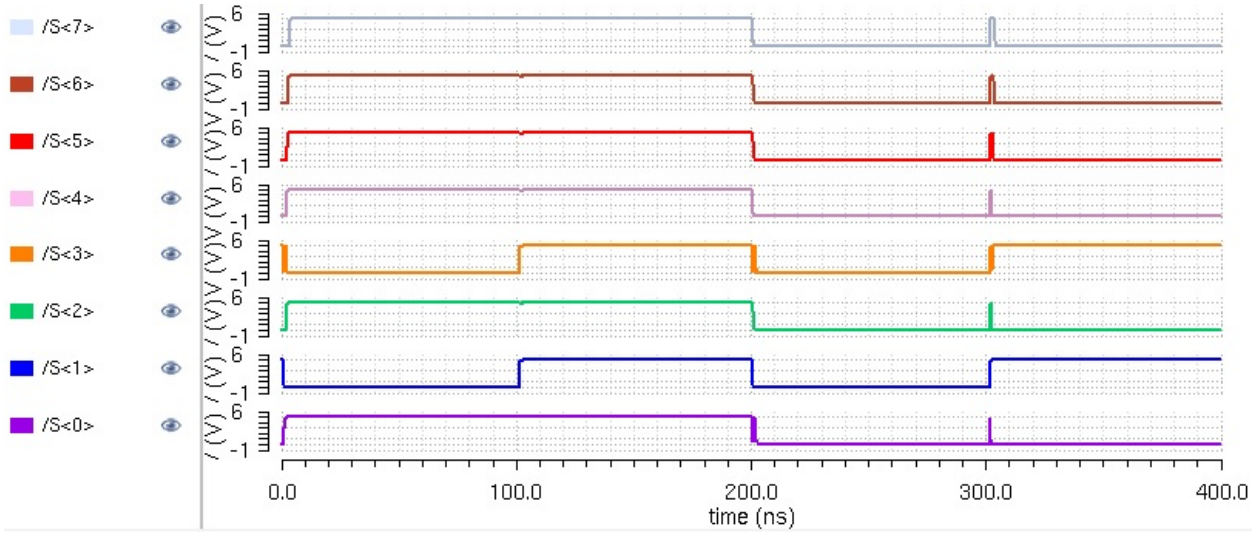
ADD Simulation

The following is a simulation for four addition operations, each occurring every 100 ns.


/A<7>
/A<6>
/A<5>
/A<4>
/A<3>
/A<2>
/A<1>
/A<0>
6
-1
0.0
100.0
200.0
300.0
400.0
time (ns)


/B<7>
/B<6>
/B<5>
/B<4>
/B<3>
/B<2>
/B<1>
/B<0>
6
-1
0.0
100.0
200.0
300.0
400.0
time (ns)


/Cin
/Cout
V (V)
6
5
4
3
2
1
0
-1
0.0
100.0
200.0
300.0
400.0
time (ns)



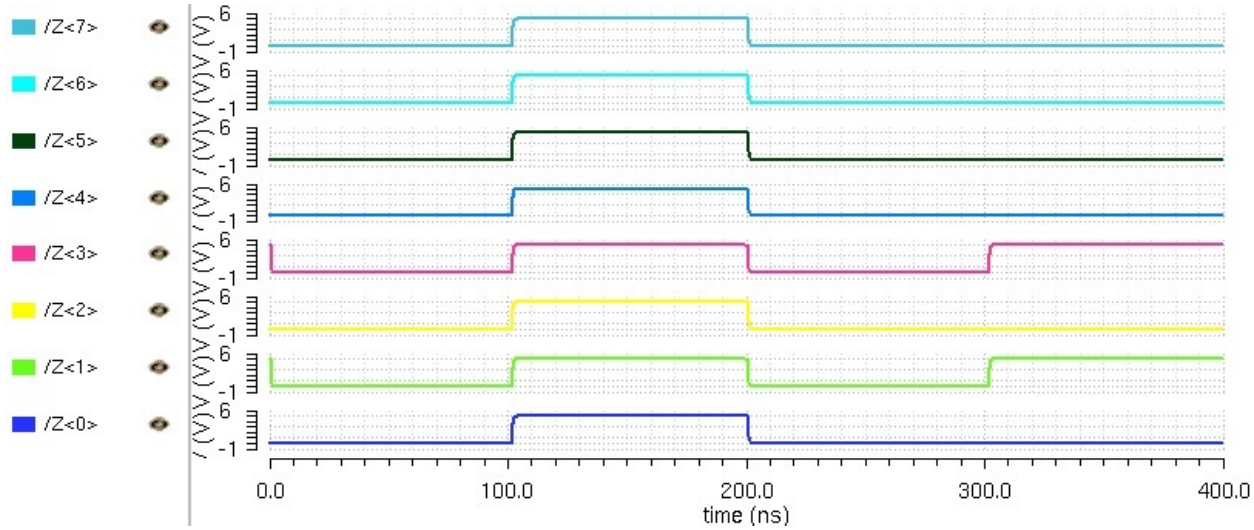
A few analyses of the simulations are as follows:

Time: 0-100 ns
A = 01100100 = 100 (in decimal)
B = 10010001 = 145 (in decimal)
Cin = 0
S = 11110101 = **245** (in decimal)
Cout = 0

Time: 300-400 ns
A = 10011011 = 155 (in decimal)
B = 01101110 = 110 (in decimal)
Cin = 1
S = 00001010
Cout = 1

Since we have Cout = 1, we bring the 1 in front of the S, giving us a value of
Sum = 100001010 = **266** (in decimal)
AND Simulation

Using the same values for A and B, we simulate the AND operation and the results are below:



Time: 300-400 ns
A = 10011011
B = 01101110
Z = **00001010**

It can be seen that the ADD function works properly.

7. Project Results

A PCB circuit board was designed using Altium Designer. Components were purchased and soldered to the manufactured PCB. When our chip was received from MOSIS we tested it and found that it mostly worked. There was only one issue with the output pins being driven to VDD for one clock cycle every time an output command was executed. This problem is fixed in the design files we have produced now. The solution was to add a single inverter to invert the clock signal and eliminate the timing issue.

Overall our Senior Design project was successful.

8. Prototype Cost Estimate

8.1 Cost Estimate

Commercial and personal microchip fabrication prices can cost a very large amount of money. We will be fabricating our microchip through MOSIS. Since we are doing an educational project,

MOSIS will be fabricating the chip completely for free. The only cost on our part will be to fill out a survey response commenting on the quality of the operation of the chip in order to help MOSIS perfect their products.

Other costs incurred for the project will be the price of a PCB board, which will depend on the size of the board, and the price of the microcontroller that we will be using to test our chip.

If this were a commercial project, we can make an estimate of how much this prototype would cost. Using round numbers, our team spent well over 200 hours designing this chip. At a wage of $30/hour, this is a cost of $6,000. Dr. Baker mentioned a commercial cost of $6,500 for 40 chips, without packaging. Additional costs for the Testing, such as the PCB board can be estimated as $500. In very round numbers we therefore estimate a commercial cost for this prototype in the range from $13,000 to $15,000, assuming no serious additional costs are incurred.

Appendix A

A.1 Control Signal Generation For All Instructions

This is an internal working document used to describe the exact control signals required at each step of an instruction. Note that the first two steps (step0 and step1) are always used to fetch the low and high bytes of the instruction.

This document will list the outputs asserted by the control unit for each instruction cycle. The fetch cycle is listed first, and is assumed for all instructions. The Step counter value is listed on the left as three binary digits.

Special signals not listed in Datapath_Comments are Step_Reset, which resets the control unit's step counter, and Step_Pause, which will prevent the step counter from automatically incrementing (used for multi-step instructions like the shifting instructions).

Two additional control signals need to be added to the datapath. The Input_BusA and Output_BusA signals will read and write data from external ports to bus A.

Note about comparison instructions
These instructions use the "Register Instruction Format" with one register operand and a 9-bit immediate constant. The operand register is compared to zero and if the condition is true, jumps to the memory location indicated by the immediate value.

[Alternative design: compare registerA with literal zero, which will eliminate the extra control line and simplify these instruction]
USE THIS DESIGN!!

IS_GREATER is the contents of RegA (inside ALU) are greater than zero (signed).
IS_ZERO is asserted when the contents of RetB are zero.

Note for shift and rotate instructions: there is a 3-bit down counter within the ALU that is loaded with ALU_LoadCounter from the lower 3 bits of opcode. When this reaches zero it sends signal COUNTER_ZERO.

RF_WriteImmediate is another new control signal that directs the RF to load from the lower 8 bits of the IR instead of from BusA.

IMPORTANT:
The RF_WriteImmediate signal activates a TG that puts the signal on the BusA.
This is a very important point!!
//END IMPORTANT

Fetch Cycle

Step Ctr. Signals asserted for each step
000 - PC_Update, PC_ReadLow ;We do not need ReadMem_Enable, because we use separate enable logic to load the PC
001 - PC_Update, PC_RealHigh ; Load second byte into IR.

ADDI
010 - ALU_LoadA, ALU_LoadBImmediate, RF_ReadEnable
011 - ALU_Control[1] ; Add code sent to ALU
100 - ALU_Output, RF_WriteEnable, RegisterA_Select, Step_Reset

ADD
010 - ALU_LoadA, RF_ReadEnable ;not loading immediate this time
011 - ALU_Control[1]
100 - ALU_Output, RF_WriteEnable, RegisterA_Select, Step_Reset

SUB
010 - ALU_LoadA, RF_ReadEnable
011 - ALU_Control[1], ALU_Control[0]
100 - ALU_Output, RF_WriteEnable, RegisterA_Select, Step_Reset

AND
010 - ALU_LoadA, RF_ReadEnable
011 - ALU_Control[2]
100 - ALU_Output, RF_WriteEnable, RegisterA_Select, Step_Reset

OR

010 - ALU_LoadA, RF_ReadEnable
011 - ALU_Control[2], ALU_Control[0]
100 - ALU_Output, RF_WriteEnable, RegisterA_Select, Step_Reset

NOT
010 - ALU_LoadA, RF_ReadEnable
011 - ALU_Control[0]
100 - ALU_Output, RF_WriteEnable, RegisterA_Select, Step_Reset

XOR
010 - ALU_LoadA, RF_ReadEnable
011 - ALU_Control[2], ALU_Control[1]
100 - ALU_Output, RF_WriteEnable, RegisterA_Select, Step_Reset

IN
010 - RF_WriteEnable, Input_BusA, Step_Reset, RegisterA_Select [Why? Because for convenience, the output of the arithmetic instructions is to the first register….

OUT
010 - RF_ReadEnable, Output_BusA, Step_Reset, RegisterA_Select[?] ;Same goes for OUT command.

JE
010 - RF_ReadEnable, ALU_LoadA, RegisterA_Select

011 - (if !IS_ZERO) Step_Reset ;if regA is NOT zero, we stop
011 - (if IS_ZERO) PC_Update, PC_LoadImmediate, Step_Reset
;if we are at zero, load IR

JG
010 - RF_ReadEnable, RegisterA_Select, ALU_LoadA
011 - (if !IS_GREATER) Step_Reset
011 - (if IS_GREATER) PC_Update, PC_LoadImmediate, Step_Reset
; Jump to new location

JGE
010 - RF_ReadEnable, ALU_LoadA, RegisterA_Select
011 - (if !IS_GREATER && !IS_ZERO) Step_Reset
011 - (if IS_GREATER || IS_ZERO) PC_Update, PC_LoadImmediate, Step_Reset

JNE
010 - RF_ReadEnable, ALU_LoadA, RegisterA_Select
011 - (if IS_ZERO) Step_Reset

011 - (if !IS_ZERO) PC_Update, PC_LoadImmediate, Step_Reset

LOAD
010 - AddrSelect[address from IR], ReadMem_Enable, RegisterA_Select, RF_WriteEnable, Step_Reset

NOP
010 - Step_Reset [?] ;possibly move to step 4 (100) for greater delay?

ROR
010 - ALU_LoadA, RF_ReadEnable, ALU_LoadCounter
011 - if (!COUNTER_ZERO) ALU_Shift[0], ALU_Shift[1], PauseStep
011 - if (COUNTER_ZERO) ALU_Output, RF_WriteEnable, RegisterA_Select, Step_Reset

SET
010 - RF_WriteImmediate,RegisterA_Select, Step_Reset, RF_WriteEnable

SHL
010 - ALU_LoadA, RF_ReadEnable, ALU_LoadCounter
011 - if (!COUNTER_ZERO) ALU_Shift[0], PauseStep
011 - if (COUNTER_ZERO) ALU_Output, RF_WriteEnable, RegisterA_Select, Step_Reset

SHR
010 - ALU_LoadA, RF_ReadEnable, ALU_LoadCounter
011 - if (!COUNTER_ZERO) ALU_Shift[1], PauseStep
011 - if (COUNTER_ZERO) ALU_Output, RF_WriteEnable, RegisterA_Select, Step_Reset

STOP
010 - GateClock

STORE
010 - WriteMem_Enable, AddrSelect, RF_ReadEnable, Step_Reset, RegisterA_Select

Simulation example of entire chip

The following simulation runs the following code:

0x00 IN R5
0x02 SET R6, 0xf3

0x04 NOT R2, R6
0x06 XOR R3, R2, R5
0x08 JG R3, 0x0C ; *
0x0A OUT R3
0x0C STOP

*This jump starts executing at 3.4us, and jumps to the stop instruction, which loads opcode at 2.9us, causing the control unit to generate the GateClock signal.