John Enyeart
Mohammed Al-Karawi
Ruben Medina
Faculty Advisor: Dr. R. Jacob Baker

8-bit MIPS Processor

Introduction

A MIPS processor is one version of a reduced instruction set computer (RISC). It is a model that is studied often in university computer architecture courses because it has enough complexity to include all the main aspects of modern processors, but isn't overly complex so as to bog students down in the details. MIPS processors have been around since the early 1980's and gained much popularity in the 1990's as it was estimated that one third of all RISC processors used the MIPS architecture, including embedded systems for the Sony PlayStation 2 and the PlayStation Portable. [1]

Our project is an 8-bit MIPS processor. Our motivation for doing this project is not to come up with something that hasn't been done before or to come up with a cheaper alternative of an existing technology. Our motivation is to produce a processor that can be used for educational purposes in university classrooms. The steps that we took in the design and fabrication of this processor could be documented and used as a set of labs for an electronics class, for example. Everything we are using in this project is freely available for educational purposes. The design software is available for free online, and even the chip fabrication through Metal Oxide Semiconductor Implementation System (MOSIS) is free for educational projects. Therefore, as a tool for students and teachers, it has a large potential value. The design and testing of the processor is also a good application of skills we have accumulated over the course of our college career, requiring knowledge of electrical and computer engineering in order to accomplish it.

System Specification

The MIPS processor has three main components:

- Data path
- Controller
- ALU control

Datapath: Since the processor is an 8-bit processor, the datapath is eight bits wide. We are using a very regular design so the datapath is divided in eight symmetric bitslices. Each bitslice contains an ALU as well as a register leading to there being eight 8-bit registers available for the processor to use. Since the control signals need to travel across all eight bitslices, we have a zipper to buffer and drive the control signals into the data path.
Controller: The MIPS controller is responsible for decoding the fetched instructions from memory and sending the control signal to the data path, where they would be driven by the zipper, and the ALU control signal to the ALU control. The Controller is also responsible for the memory writes

and program counter. It is worth mentioning that the controller acts as finite state machine. This concept will be expanded on in the controller section.
ALU control: As mentioned above, the actual ALU lies at the end of the bitslice, one ALU per bitslice. The ALU controller receives ALUOp, two bits, that determine the operation that the ALU needs to carry out. The ALU controller then sends the control signals to the ALU in order for each operation to be carried out. The ALU takes three control signals in order to determine the function the ALU needs to carry.

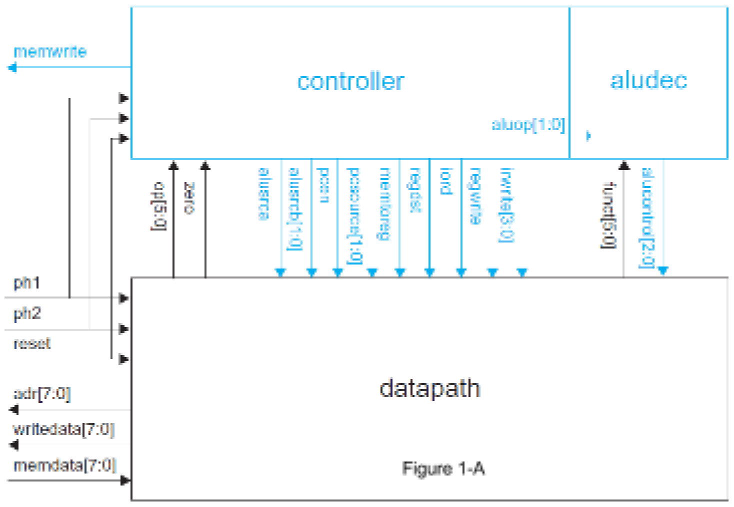


Figure 1-A

Figure 1-A [2] shows a block diagram of main components and how they relate to each other. Aludec block refers to the ALU control. While Figure 1-B [3] shows a tree diagram that shows the elements of the main components. The standard library would contain most of the elements under the bitslice branch.

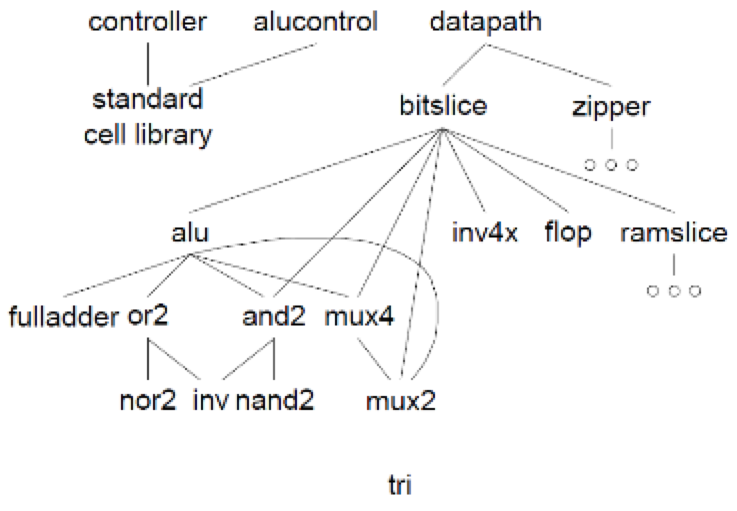


Figure 1-B

System components:-

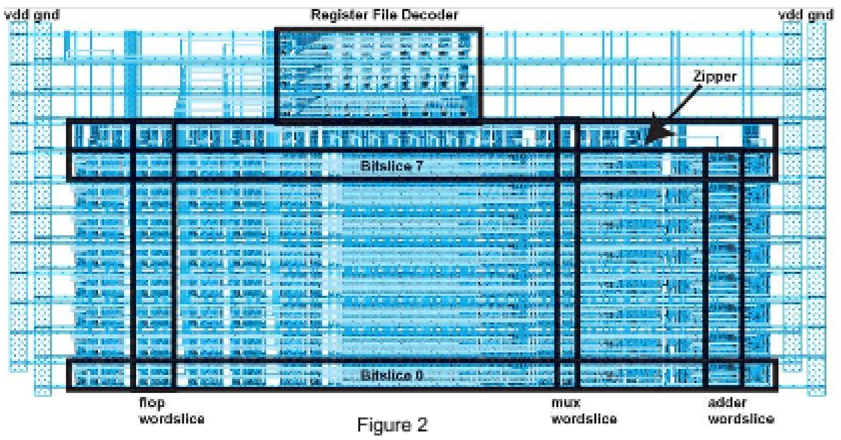


Figure 2

Datapath: Figure 2 [2] reflects the initial design for the datapath. The main components of the datapath are the Bitslice, Zipper and ALU.

Bitslice: Each bitslice of the datapath is used to store and retrieve data from the register within it. Data flows from left to right. The control signals flows over the top from the zipper. The bitslice can be divided into few components as seen in Figure 3, and addressed below:

- The address mux: Contains the address of the upcoming instruction. Since this MIPS processor is multi-cycle, each bitslice is one bit of the the address for the upcoming instruction. Figure 3 [3], adrmux.
- Instruction Flops: The address mux is followed by four flops which hold the instruction for the four cycles of instructions that need execution. figure 3, IR[3:0]. The flops are followed by the memory data register. Figure 3, MDR.
- Register File interface: This is made up of the write data multiplexer. Figure 3, writemux. The source registers. Figure 3, srcB & srcA. The source registers hold the values of the operands for the ALU.
- Register file: Contains four dual sram flip-flops used to store 8-bits of data read from memory to be used by the processor when required
- Program Counter: The program counter points to the next instruction set. The program counter is composed of a multiplexer used to select the next value of the program counter flip-flop and an AND gate that can be used to reset the pc counter to zero by a reset signal or by program restart. Figure 3, PC.

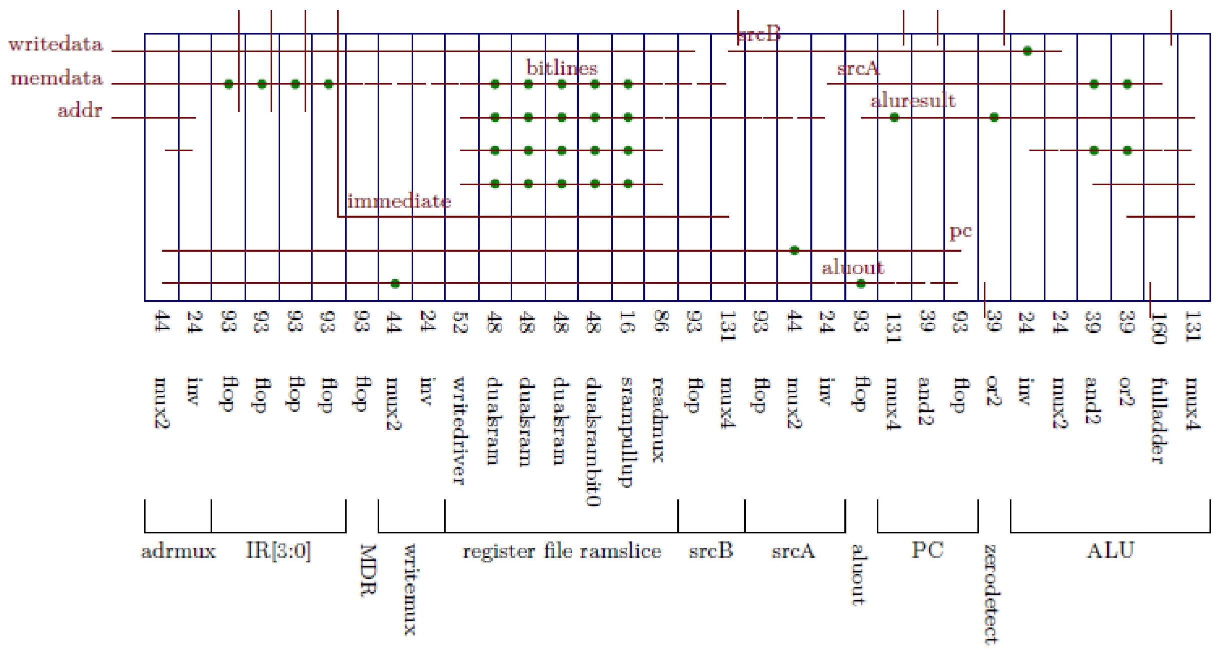


Figure 3: Datapath slice plan.

ALU: The arithmetic logic unit (ALU) is used to execute any instructions sent to the processor and thus determines what functions the processor will be able to execute. The ALU design we chose has five functions:

- ADD: Add two digits A + B
- SUB: Subtract one digit from another A - B
- AND: Logic AND function
- OR: Logic OR function
- SLT: Set less than. For example, it produces a 1 if A is less than B and a 0 otherwise.

Using these basic functions, the processor can perform more complex functions such as add immediate, branch if equal, jump, load byte, store byte, etc by working in conjunction with the internal registers and external memory.

Zipper: The control signals need to travel across all of the eight bitslices so they need to be strong. The zipper acts as the middle-man between the controller and the datapath. It takes signals from the controller and generates signals to select registers in the ramslices of the datapath. It consists of multiplexers, buffers, and decoders. It also provides the complement of the control signals so the bitslice would not require local inverters.

ALU Controller: The ALU controller takes the function signal from the controller and then generate a control signal for the ALU which selects one function to perform (ADD, SUB, AND, OR, or SLT).

MIPS Controller: The MIPS controller uses a Finite-State-Machine (FSM) to generate the control signals, and the register selects. The instruction set used by the processor is the standard MIPS instruction set which is 32-bits per instruction. Because each instruction is 32-bits, each instruction must be fetched using four, 8-bit instruction fetches. The controller fetches the instruction of an off chip memory. As mentioned above, the MIPS design that we used is a multi-cycle implementation so the first four states of the FSM are simply instruction reads. Figure four [2] provides an example of the state transition of a 32-bit instruction set MIPS controller FSM. The state diagram for the MIPS processor is given in figure 4.

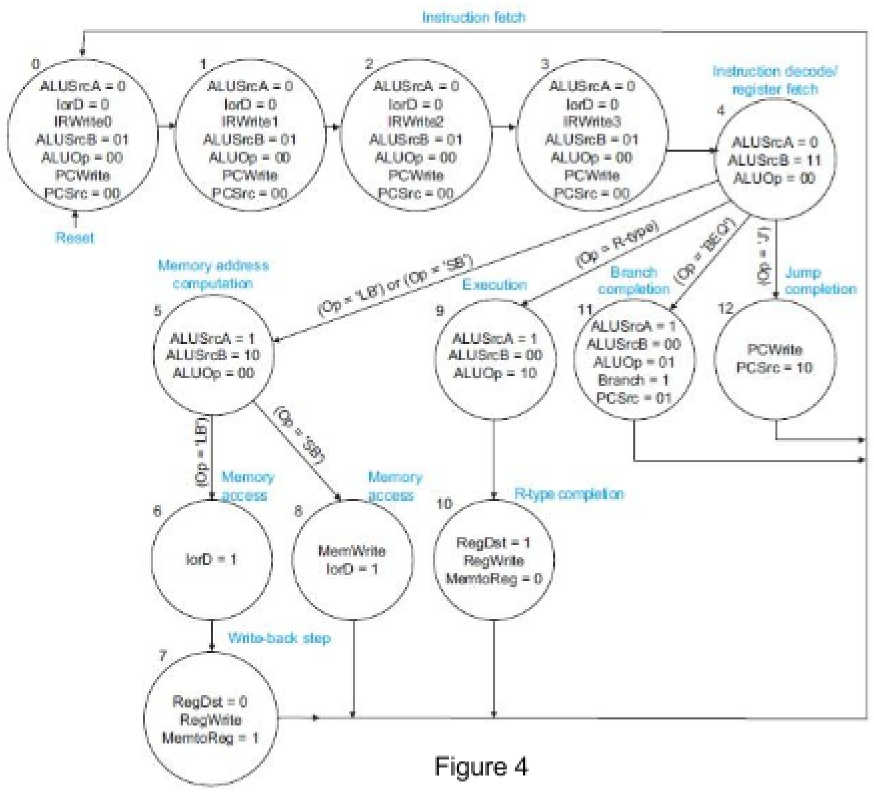


Figure 4

System Architecture

As mentioned before, the processor is an 8-bit MIPS processor running the 32-bit MIPS instruction set. The processor is designed from the transistor level up using the MOSIS ON Semiconductor C5 300nm process. The C5 process is designed for 5 volt applications and has 3 metal layers, 2 polysilicon layers, and a high resistance layer [4]. As we are starting at the transistor level all logic gates, flip-flops, and other components have been custom designed by ourselves for use in our processor. The MIPS and ALU controller components of our design are synthesized using standard cells created by us along with VHDL code that describes their operation.

Functional Analysis and Requirements

To design our processor we are using the Electric VLSI design system with the IRSIM simulation plugin. The C5 process we are going to use to fabricate our processor has a set of design requirements that must be followed to allow our chip to be fabricated. Using Electric, we are able to run design rule checks against these requirements on our design that verify that we are adhering to the design rules set by the MOSIS fabrication service. We are also able to use Electric to check that the transistor layouts we create match the schematic of each component of our processor. As fabrication of our design will be costly and time-consuming it is important that we simulate and verify as much of the design as possible to ensure that when we receive the fabricated chip it works as was designed. Using the IRSIM simulation plugin with Electric, we are able to simulate and analyze every component of our design at the transistor level to verify that they are working correctly.

Subsystem/Component Alternative Generation and Trade-offs

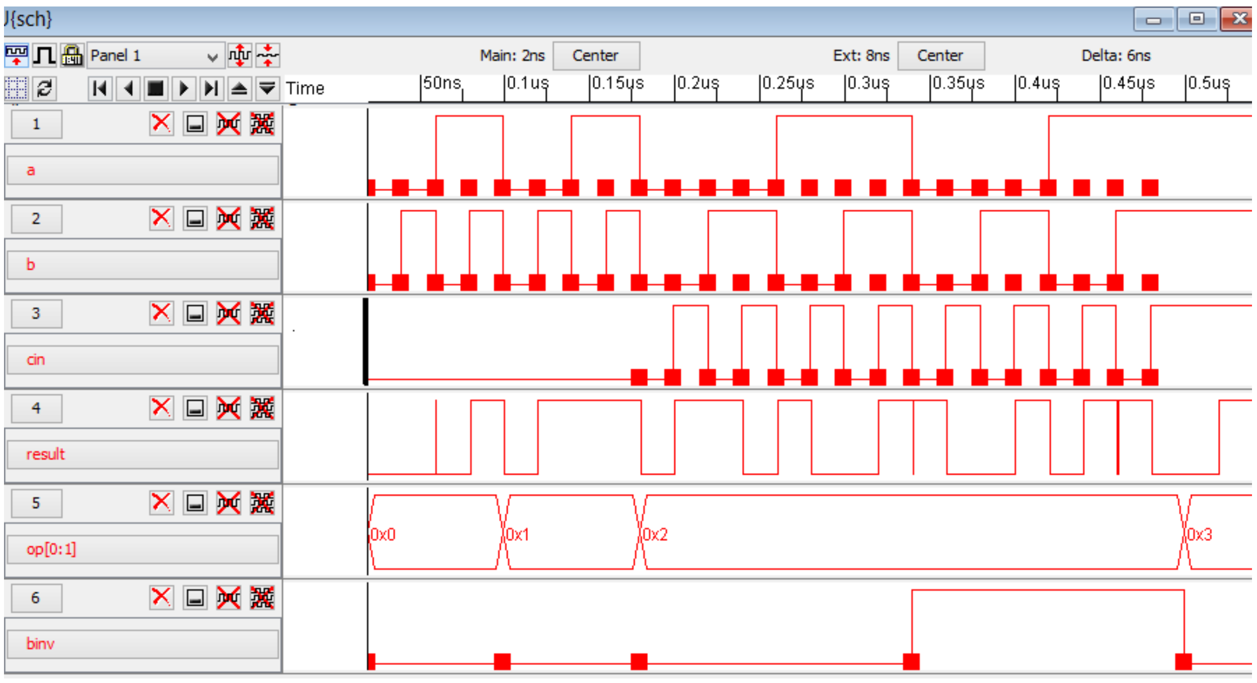
The main trade off was selecting a certain MIPS microarchitecture. The microarchitecture determines how the processor executes the instruction set. The three main microarchitectures are [5]:

1. Single Cycle Microarchitecture: Executes each instruction in one clock cycle. This microarchitecture has the simplest control, but since it executes every instruction in one clock cycle then the speed of the processor is determined by the slowest instruction. It is also space intensive because it requires extra hardware.
2. Multi-Cycle Microarchitecture: Executes each instruction in many clock cycles. this allows fast instruction to take few clock cycles while slow ones to take longer clock cycles. It also saves on space by reusing expensive hardware such as the ALU and adder. However it requires more complicated logic than the single cycle microarchitecture.
3. Pipelined Microarchitecture: This architecture uses a single cycle architecture adds a pipeline to the execution. Increasing the throughput of the execution tremendously. It however needs added logic to deal with dependencies between multiple cycles and pipeline.

Even though most modern processors use some sort of pipeline architecture, we decided to go with the Multi-Cycle architecture. We believe that it provides enough challenge, but also have the ability to finish it and test it within two semester.

Simulations Results

All simulations of our design are done using the IRSIM simulation plugin in conjunction with Electric.
For the simulation of our ALU, we created a command file containing the input signals and function selection signals which can be found in the appendix. The simulation of our ALU schematic can be seen below:

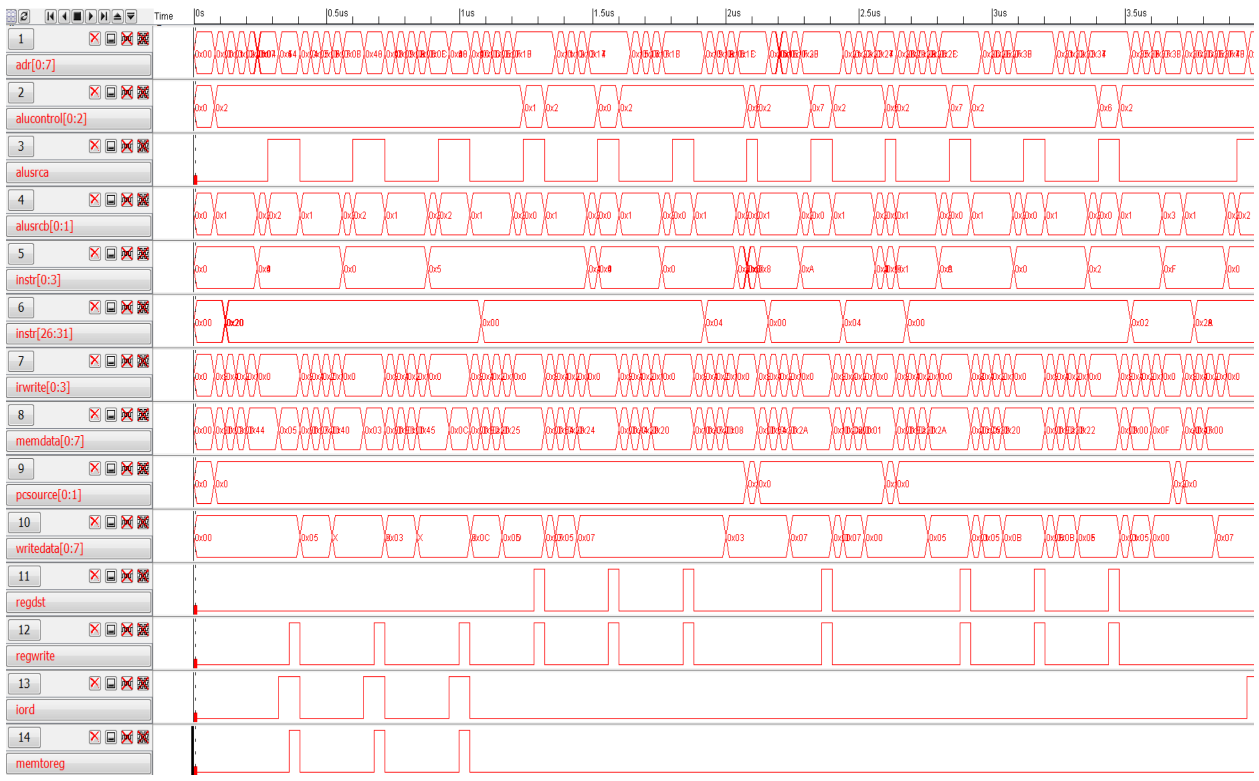


OpCode	Operation
00	AND
01	OR
10	ADD/SUB
11	SLT

The subtraction operation is performed by selecting the inverted b input for the full adder of the ALU and setting cin to 1 to add the two's complement of b performing subtraction. The inverted b input is generated by the zipper.

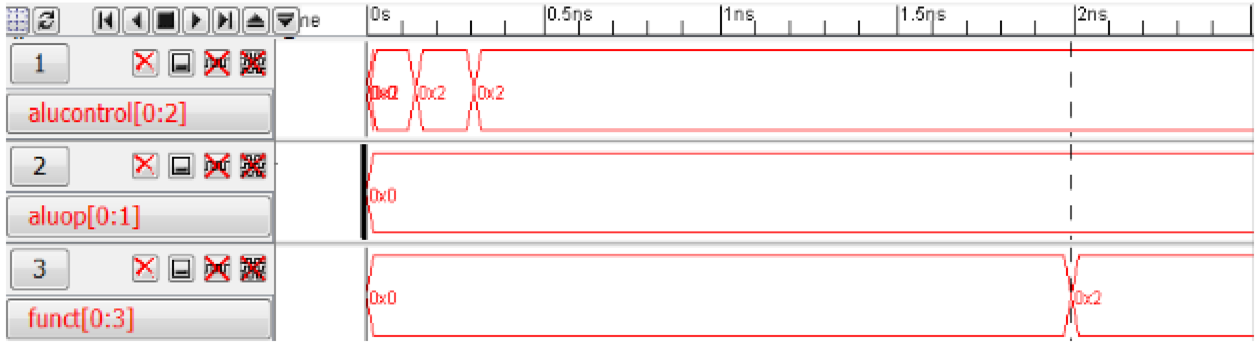
In order to verify that every part of the processor was working correctly, a command files was created to test each single part. Here are the tested part with their simulation included below.

Datapath:

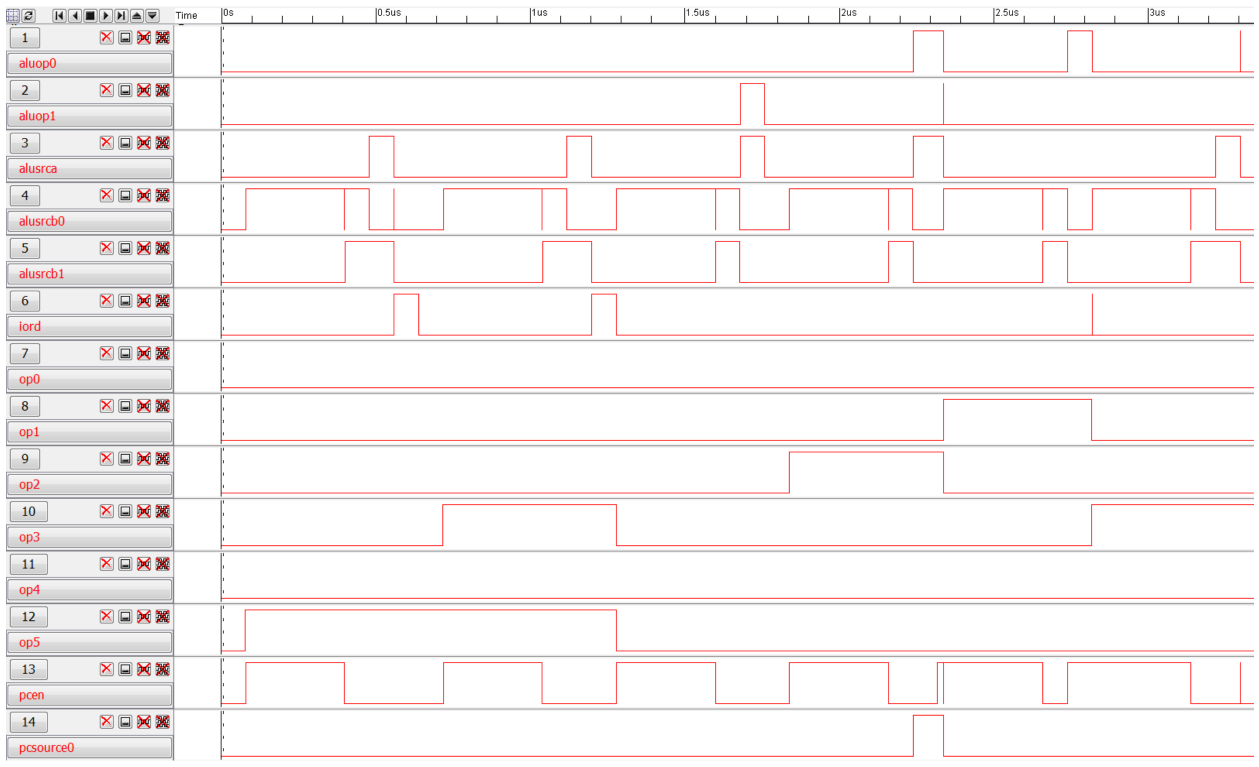


The file can not be seen clearly, but all operations were verified to be working correctly.

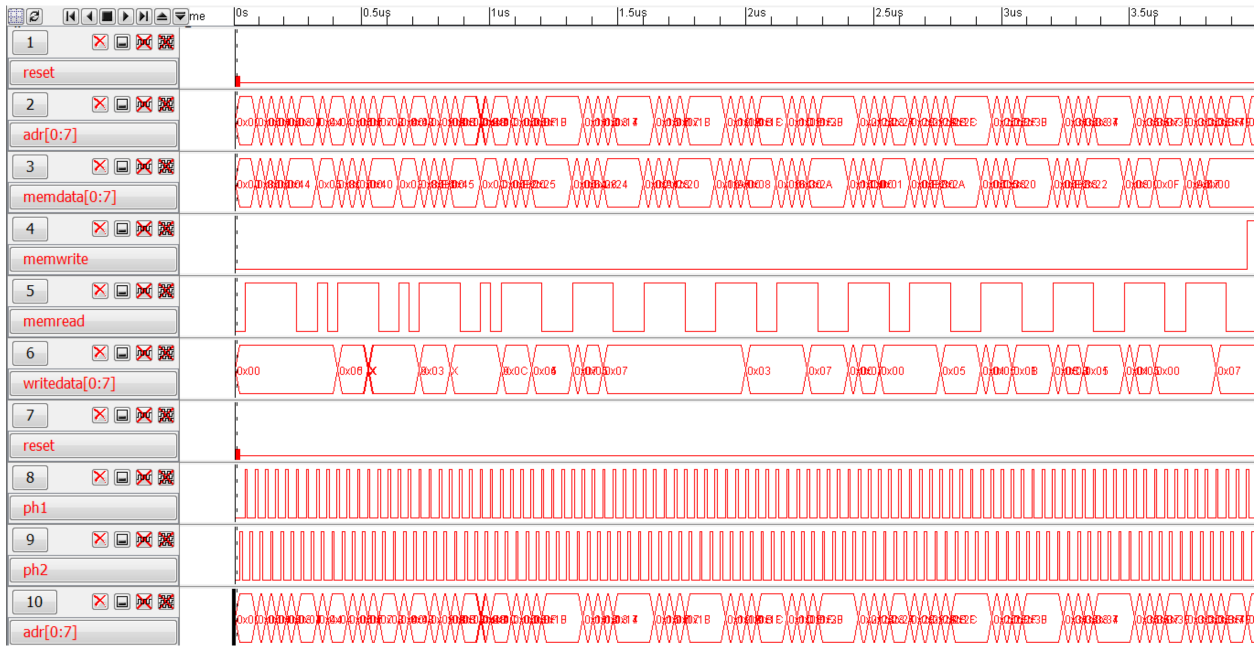
ALU control:



MIPS Controller:



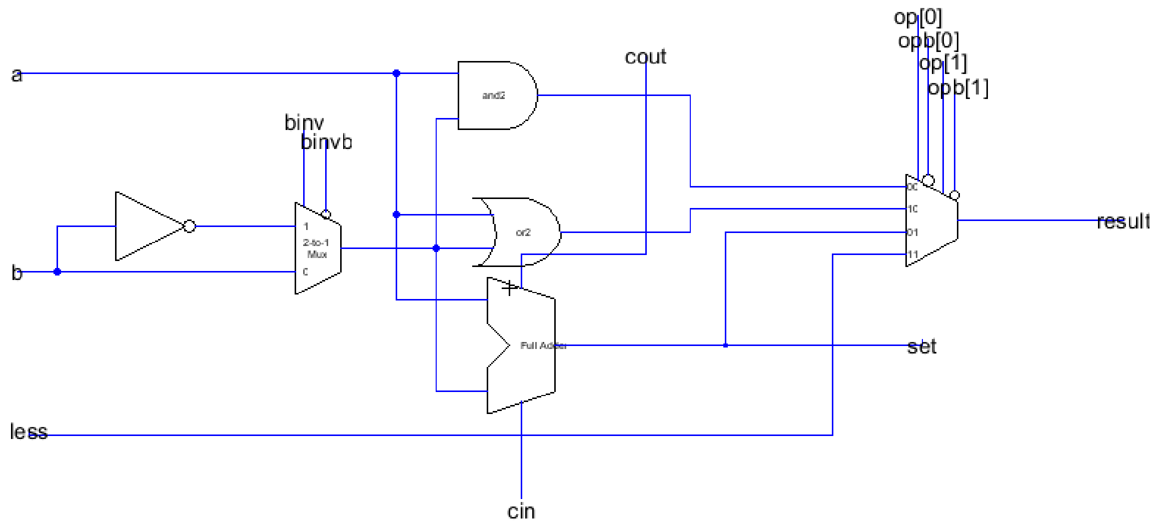
MIPS simulation:



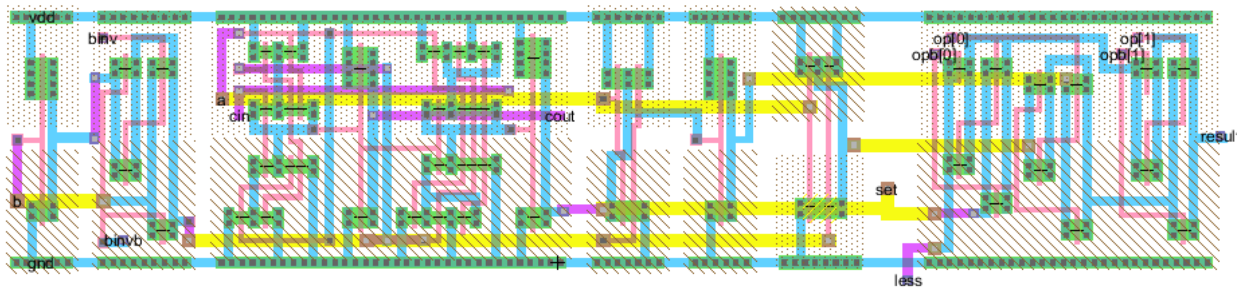
The command files are rather lengthy so they were not included in the report. They can be found at <https://dl.dropboxusercontent.com/u/63128830/Simulations.zip>

Presentation of Design

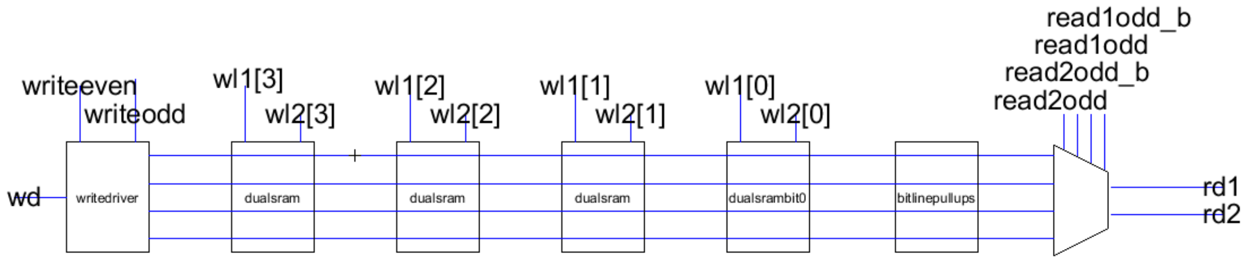
ALU Schematics:



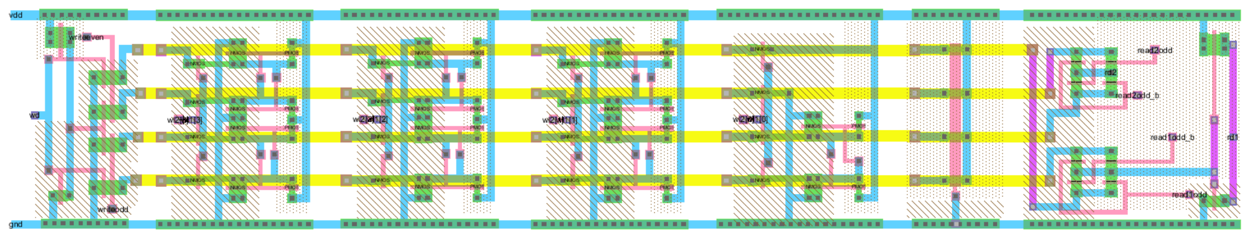
ALU Layout:



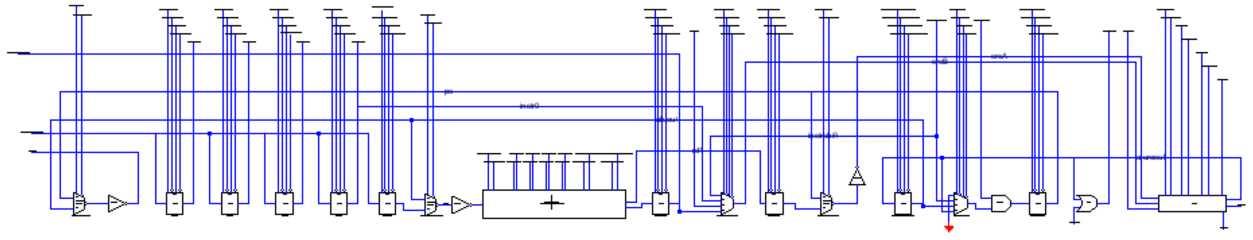
Ramslice schematic:



Ramslice layout:

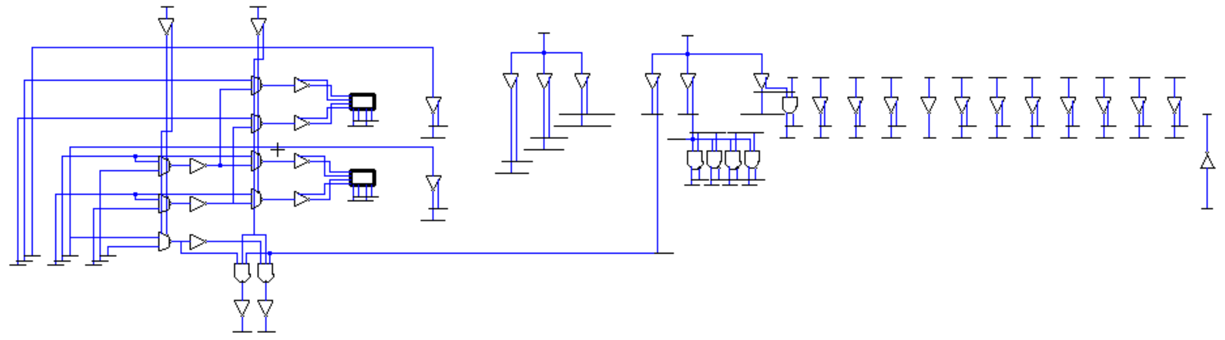


Bitslice Schematics:

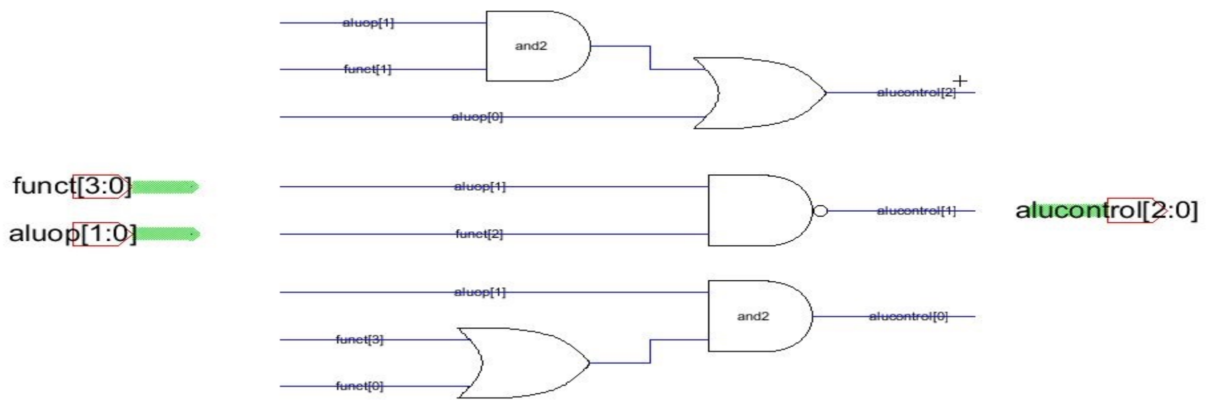


Bitslice Layout:

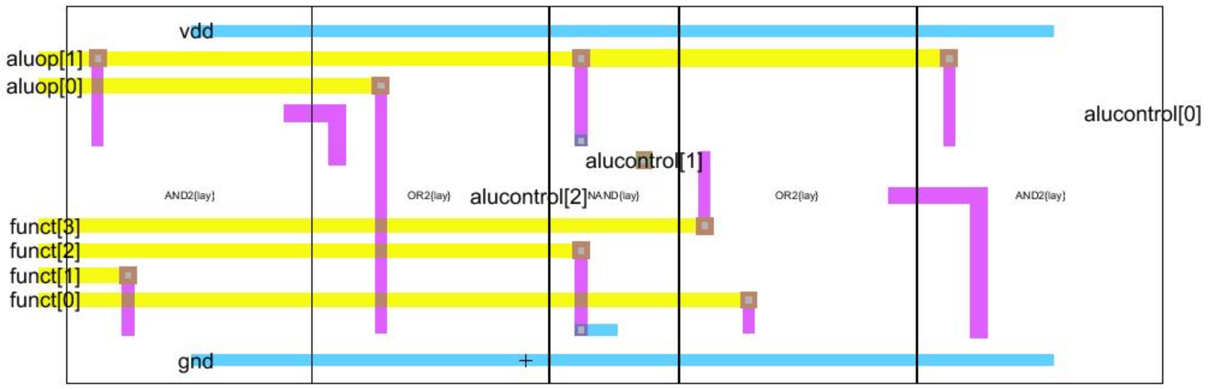
Zipper Schematics:



ALU control Schematics:

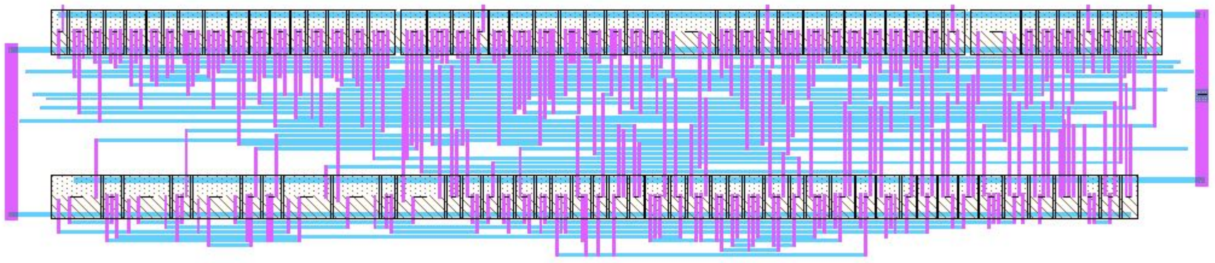


ALU control Layout:



Controller:

Since the controller was generated with HDL then only the layout will be included.



The overall layout is quite big so most of the details will not be visible if it is included here. The overall layout includes the connection to the pad frame. Even thought the overall layout is not included, the jelip for the final design can be found be accessed from <https://dl.dropboxusercontent.com/u/63128830/MIPS_PROC_RMJ.jelib> Jelib files can be opened using the free CAD tool Electric. For a detailed tutorial to using Electric refer to <http://cmosedu.com/cmos1/electric/electric.htm>

Testing Process

Unlike microcontrollers, processors lack memory so we would have to depend on memory located off chip to contain instructions and store results. In order to test our processor we developed a PCB that houses both the processor and a microcontroller, ATmega328P**,** the MIPS instructions were sent to the microcontroller which in turn sends them to the processor. The memory pins of the processor were connected to an LED bar and we checked the outputs to verify that we obtained the expected results for each instruction sent in. Luckily, everything worked as expected.

User Manual

Introduction

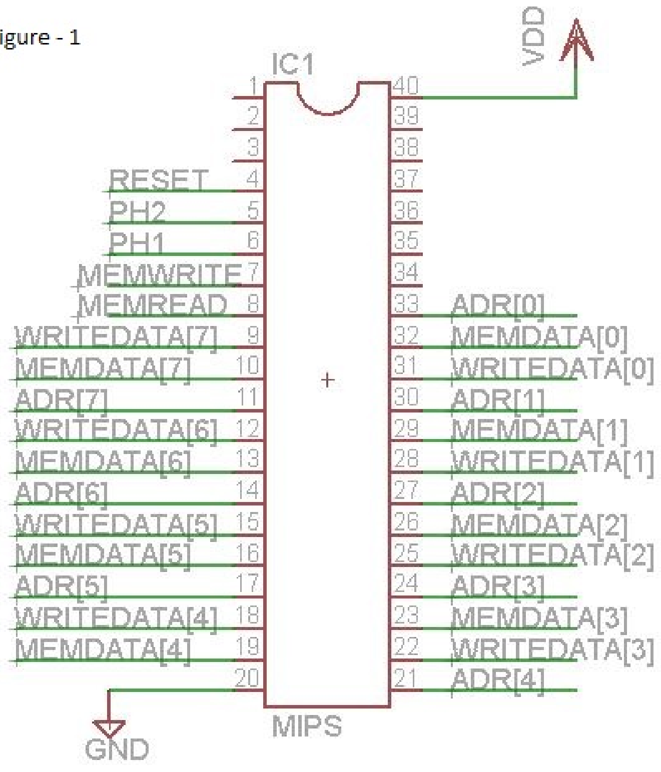
A MIPS processor is one version of a reduced instruction set computer (RISC). It is a model that is studied often in university computer architecture courses because it has enough complexity to include all the main aspects of modern processors, but isn't overly complex so as to bog students down in the details. MIPS processors have been around since the early 1980's and gained much popularity in the 1990's as it was estimated that one third of all RISC processors used the MIPS architecture, including embedded systems for the Sony PlayStation 2 and the PlayStation Portable. [1]

The current MIPS processor is designed as an educational tool for students to further study the MIPS architecture. The scale is large by today's standards--only 300 nm, but this makes it better as an educational tool because the transistors still generally adhere to the square law equations. This makes the design easier to understand and implement. The ALU functions and instruction set are also limited compared to what is possible, and students using our design are free to add or exchange functionality as they see fit. Overall, the design is very flexible and allows students to explore the MIPS architecture.

Processor Specifications

Design Scale	300 nm
Ideal voltage VDD	5 V
Pin Layout	See Figure 1

Figure - 1



ALU Functions

1. ADD: Add two digits; A + B
2. SUB: Subtract one digit from another; A - B
3. AND: Logic AND function
4. OR: Logic OR function
5. SLT: Set less than; A < B → "1", A ≥ B → "0"

Other functions that the student could add or substitute include NOR, NOT, XOR, etc.

Instruction Set:

The instruction set for MIPS can be accessed through <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>

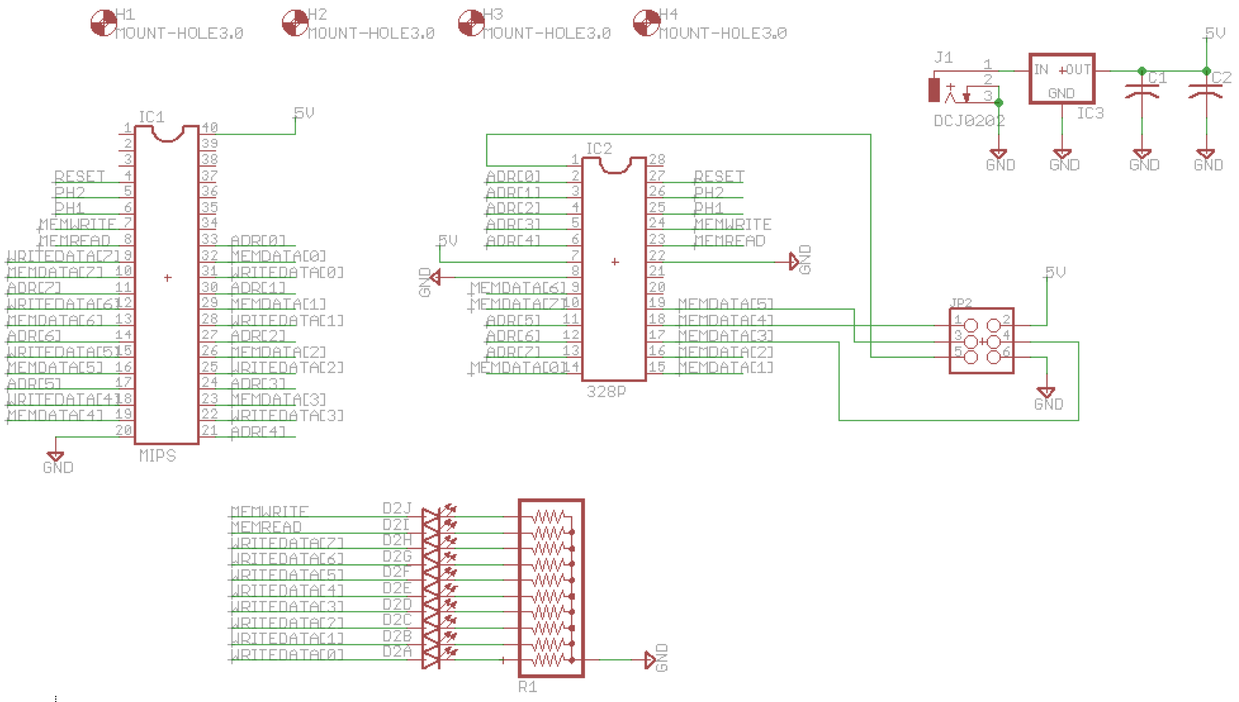
The instructions that the processor supports our based on the state diagram of the MIPS controller component. As such, the processor supports the Load, Store, register to register operations such as AND, ADD, OR, XOR, SUB, etc, and the BEQ (branch if equal) and J (Jump) instructions.

Instruction Timing

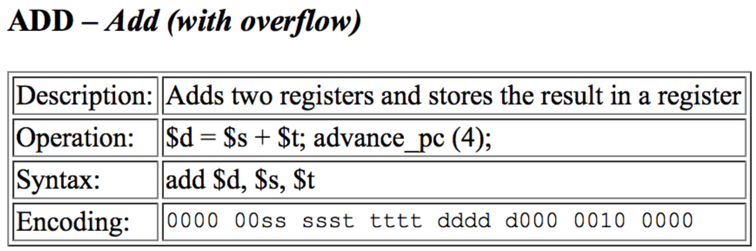
The processor is able to run correctly between 1 - 25M Hz.

1. Operation and testing instructions

To test the processor, we used a microcontroller in combination with the processor and an LED display array in order to verify that instructions were executed correctly. In order to accomplish this, the microcontroller is programmed to provide a two-phase clock signal to the processor and also send binary formatted assembly instructions from the aforementioned instruction set. The circuit setup in order to accomplish this can be seen below:



For example, to test the ADD instruction, the instruction encoding must be looked up on the assembly instruction webpage. After doing so, we can see that the encoding for the ADD instruction is the following [6]:



ADD – ***Add (with overflow)***

Description:	Adds two registers and stores the result in a register
Operation:	\$d = \$s + \$t; advance_pc (4);
Syntax:	add \$d, \$s, \$t
Encoding:	`0000 00ss ssst tttt dddd d000 0010 0000`

So, to create the hex formatted instruction that the microcontroller must send, one must replace the s, t, and d parts of the encoding with the appropriate registers within the processor. For example, to add the values in registers one and two and then store the result in register three, replace the s in the encoding with 00001, the t section with 00010 and the d section with 00011. This would result in 0000 0000 0010 0010 0001 1000 0010 0000 which can be represented in hex as 0x00221820. As this is a 32-bit instruction and the processor is only 8-bits, the instruction must be split into four when it is sent by the microcontroller to the processor. Within the source code for the microcontroller, all instructions are stored in an array which is then sent from the microcontroller and read by the processor at the end of every two-phase clock cycle. For similar example code which we used to test the ADD instruction see Appendix A.

Appendix A

```
/*
 * addtest.c
 *
 * Syntax:
 * add $d, $s, $t
 * Encoding:
 * 0000 00ss ssst tttt dddd d000 0010 0000
 *
 */

#define F_CPU 8000000UL
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

int j=-1;
int pcount=0;
int clocks=0;
unsigned char instruction[100];

// Changeable Variables
int A=42;
int B=17;

void loadinstructions(unsigned char instr[])
{
		int i=0;
		// lb $2, 68($0)	# initialize value in register 2 to A	80020044
		instr[i]=0x80;
		i++;
		instr[i]=0x02;
		i++;
		instr[i]=0x00;
		i++;
		instr[i]=0x44;
		i++;
		instr[i]=0x44;
		i++;
		instr[i]=0x44;
		i++;
		instr[i]=A;
		i++;
		instr[i]=A;
		i++;

		// lb $3, 69($0)	# initialize value in register 3 to B	80030045
		instr[i]=0x80;
		i++;
		instr[i]=0x03;
		i++;
		instr[i]=0x00;
		i++;
		instr[i]=0x45;
		i++;
```

```
        instr[i]=0x45;
        i++;
        instr[i]=0x45;
        i++;
        instr[i]=B;
        i++;
        instr[i]=B;
        i++;

        // add $4, $2, $3     # add value in register 2 and 3 and store result in register 4                 00432020
        instr[i]=0x00;
        i++;
        instr[i]=0x43;
        i++;
        instr[i]=0x20;
        i++;
        instr[i]=0x20;
        i++;
        instr[i]=0x20;
        i++;
        instr[i]=0x20;
        i++;
        instr[i]=0x20;
        i++;

        // sb $2, 0($2)     # store value in register 4 to address $2 (write to led bar)       a0440000
        instr[i]=0xa0;
        i++;
        instr[i]=0x44;
        i++;
        instr[i]=0x00;
        i++;
        instr[i]=0x00;
        i++;
        instr[i]=0x00;
        i++;
        instr[i]=0x00;
        i++;
        instr[i]=0x00;
        i++;
        instr[i]=0xFF;
        i++;
}

void tcnt_init()
{
  /* TIMER1 Delays for 256 prescaler (tccr1b=0x04)
  1s = 0x85EE
  0.5s = 0xC2F7
  0.25s = 0xE17C
  125ms = 0xF0BE
  63ms = 0xF85F */

  TCNT1 = 0xFFFE; // For delay
  TCCR1A = 0x00;  // WGM = 0 for normal mode
  TCCR1B = 0x01;  // Normal mode, prescaler 1
  TIMSK1 = 0x01;  // enable interrupt on timer1 overflow for clocking clock
```

```
    sei();          // enable global interrupts
}


int main(void)
{
                    // set up ports for output
                    DDRB=0xFF;
                    DDRC=0x1C;
                    PORTC=0x10;

                    // load instructions to be sent to processor
                    loadinstructions(instruction);
                    tcnt_init();

                    while(1)
                    {
                    }
}

ISR (TIMER1_OVF_vect)
{
            TCCR1B = 0x00;  // Turn timer off
                    if (pcount==0)
                            PORTC = PORTC ^ 0x04;
                    else if (pcount==2)
                            PORTC = PORTC ^ 0x08;
                    else if (pcount==3 || pcount==1)
                            PORTC = PORTC & 0x10;

                    pcount++;

                    if (pcount==4) {
                            pcount=0;
                            clocks++;

                            if (clocks==3)
                                    PORTC = PORTC ^ 0x10;
                            if (clocks>=3)
                                    j++;

                            if (PINC==0x02) {
                                    pcount=5;           // stop clocking the processor
                              TIMSK1 = 0x00;  // disable interrupt on timer1 overflow for clocking clock
                            }
                    }
                    PORTB=instruction[j];
                    TCNT1=0xFFFE;           // for delay
                    TCCR1B = 0x01;  // Normal mode, prescaler 1
}
```

References

[1] Rubio, Victor . *A FPGA Implementation of a MIPS RISC Processor for Computer Architecture Education* . New Mexico State University , 2004. Web. <http://www.ece.nmsu.edu/~jecook/thesis/Victor_thesis.pdf>.

[2] Harris, David, and Neil Weste. *CMOS VLSI DESIGN A Circuits and Systems Perspective*. 4th . Boston, Massachusetts: Addison-Wesley., 2011. Print.

[3] Gupta, Pallav. Department of Electrical & Computer Engineering 2009, *8-bits simplified MIPS processor lab manual,* Villanova University.

[4] http://www.mosis.com/vendors/view/on-semiconductor/c5

[5] Harris, David M., and Sarah L. Harris. *Digital design and computer architecture*. Waltham, MA: Morgan Kaufmann, 2013. Print.

[6] Frenzel, James . "MIPS Instruction Reference." *MIPS Instruction Reference*. N.p., 10 Sept. 1998. Web. 24 Apr. 2014. <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>.