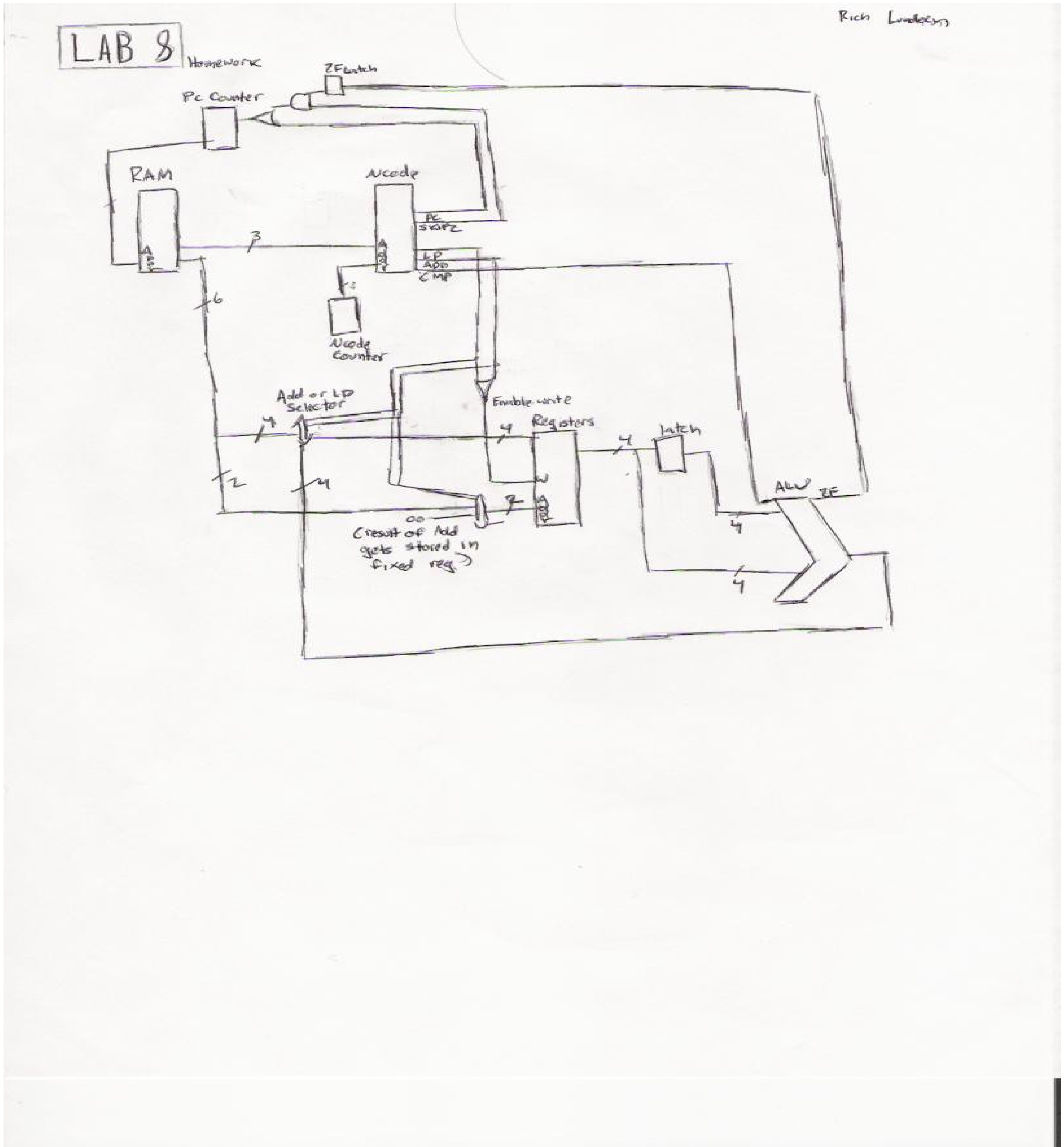
Lab 8 – System Integration

Rich Lundeen
CS 585
Computer Architecture

Project Description

The goal of this project was to integrate all parts covered throughout the lab. Similar to lab7, lab8 fetches microcode from a second memory device. Extending lab7, however, it also performs very basic operations, including add, eq, nop, ld, skipz, and halt. There are 4 physical registers, which will be referred to as 00, 01, 10, and 11.



Instructions Set

As per the lab, the instruction set consists of the following 9 bit instructions:

Inst	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
add	opcode2	opcode1	opcode0	regA1	regA0	regB1	regB0		
eq	opcode2	opcode1	opcode0	regA1	regA0	regB1	regB0		
ld	opcode2	opcode1	opcode0	regA1	regA0	data3	data2	data1	data0
nop	opcode2	opcode1	opcode0						
skipz	opcode2	opcode1	opcode0						
halt	opcode2	opcode1	opcode0						

Machine code is the assembly code translated into bytes. Below is the machine code for this architecture. The bytes are numeric values that map directly into the microcode.

```
000 - Opcode 0 (add)   = 00000000 = line 0
001 - Opcode 1 (ld)    = 00100000 = line 32
010 - Opcode 2 (eq)    = 01000000 = line 64
011 - Opcode 3 (nop)   = 01100000 = line 96
100 - Opcode 4 (halt)  = 10000000 = line 128
101 - Opcode 5 (skipz) = 10100000 = line 160
```

These values were chosen somewhat arbitrarily. However the numbers are a direct mapping to the location of the microcode. 000 (add) maps to line 0 in the microcode, and so forth. These values begin at increments of 32 since the instructions are the first three bits of a nine bit instruction.

Microcode

The microcode is used to set a series of flags depending on the instructions. These flags are used for control, and are defined as follows.

Flag	Decimal Value	Hexadecimal Value
PC	01	01
ADD	02	02
EQ	04	04
LD	08	08
SKIPZ	16	10
SET1	32	20
SET2	64	40
NOPE (halt)	128	80

The Hexadecimal value is important because it is the format that can be read by multimedia logic. These values were chosen so only one bit is set at any given time.

ADD

The ADD instruction has the following format:

Inst	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
add	opcode2	opcode1	opcode0	regA1	regA0	regB1	regB0		

The opcode is always 000, which references the ADD address in microcode. ADD simply adds Register A to Register B, and the result is stored in Register 00. This is set statically.

For example, consider:

```
000000100
```

which translates to the hexadecimal values:

```
0 04
```

The first 0 is stored in a separate memory device than the 04 (the memory devices in multimedia logic devices read 2 hexadecimal values by default. This instruction translates to something like the assembly-like instruction:

```
add 00 01
```

which adds register 00 to register 01.

The microcode for the ADD instruction is defined in the text file opcode which is read by the microcode memory device. It is defined as follows:

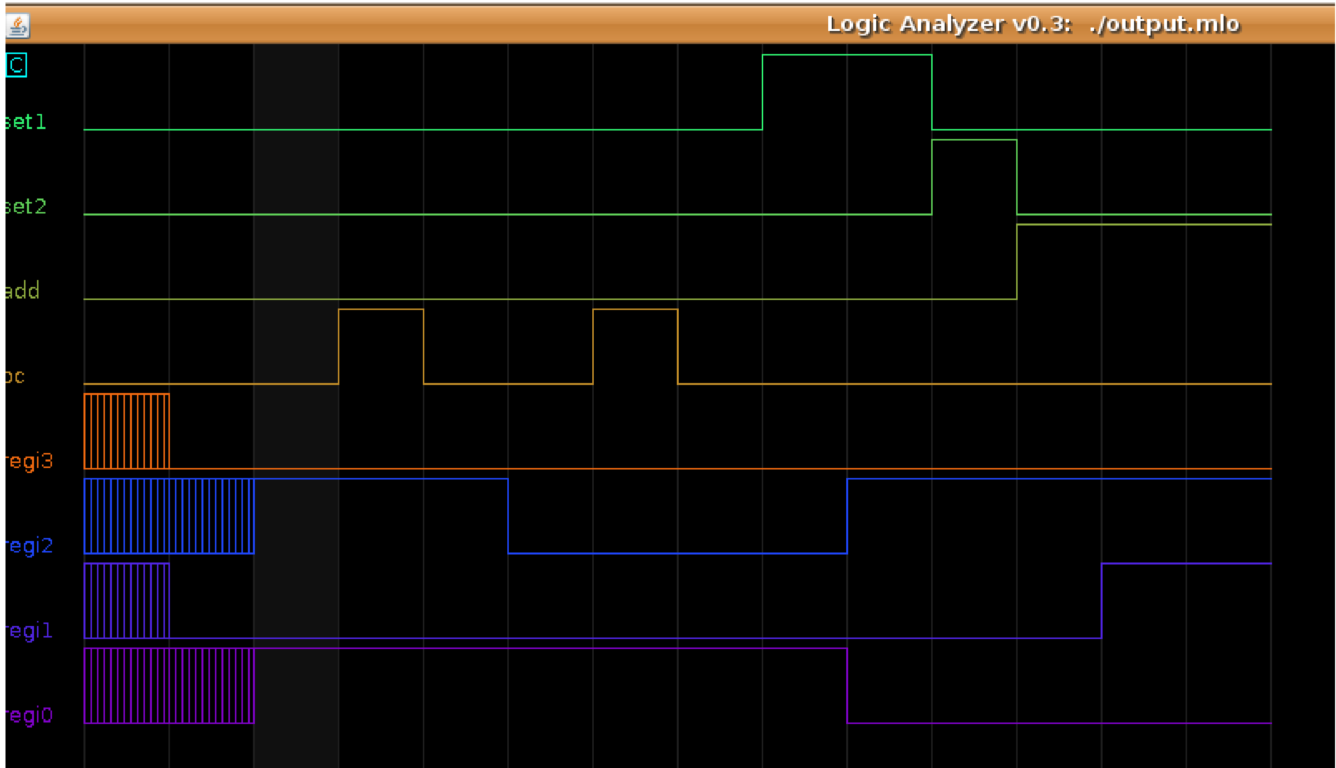
```
20
40
02
01
```

This translates into the following flags being set:

```
set1
set2
add
pc
```

The set1 instruction loads the data from the regA portion of the instruction into a data latch. set2 enables the data from the regB portion of the instruction to be loaded into a separate data latch. The data is passed through an ALU, which adds the data together. This data is stored by using the add instruction, which selects the result to be added into register 00 (which is where the results always go) and enables the write.

Below is a timing diagram of relevant flags. Although not shown here, previous to this instruction 0101 was loaded into register 00, and 0001 was loaded into register 01. Here register 00 and register 01 are being added together immediately after the second pc. The result is stored in regi3 through regi0.



EQUAL

The Equal (EQ) instruction has the following format:

Inst	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
eq	opcode2	opcode1	opcode0	regA1	regA0	regB1	regB0		

The (EQ) instruction is extremely similar to the ADD instruction, with some slight differences. First, there is no write back to register 00 like there is in ADD. Second, the operation performed is subtraction, and if the result is zero the zero flag (ZF) is set.

The microcode for EQ is defined as follows, on line 64 of the opcode text file.

20
40
04
01

which translates to the following flags being set:

SET1
SET2
EQ
PC

As an example consider the following instruction.

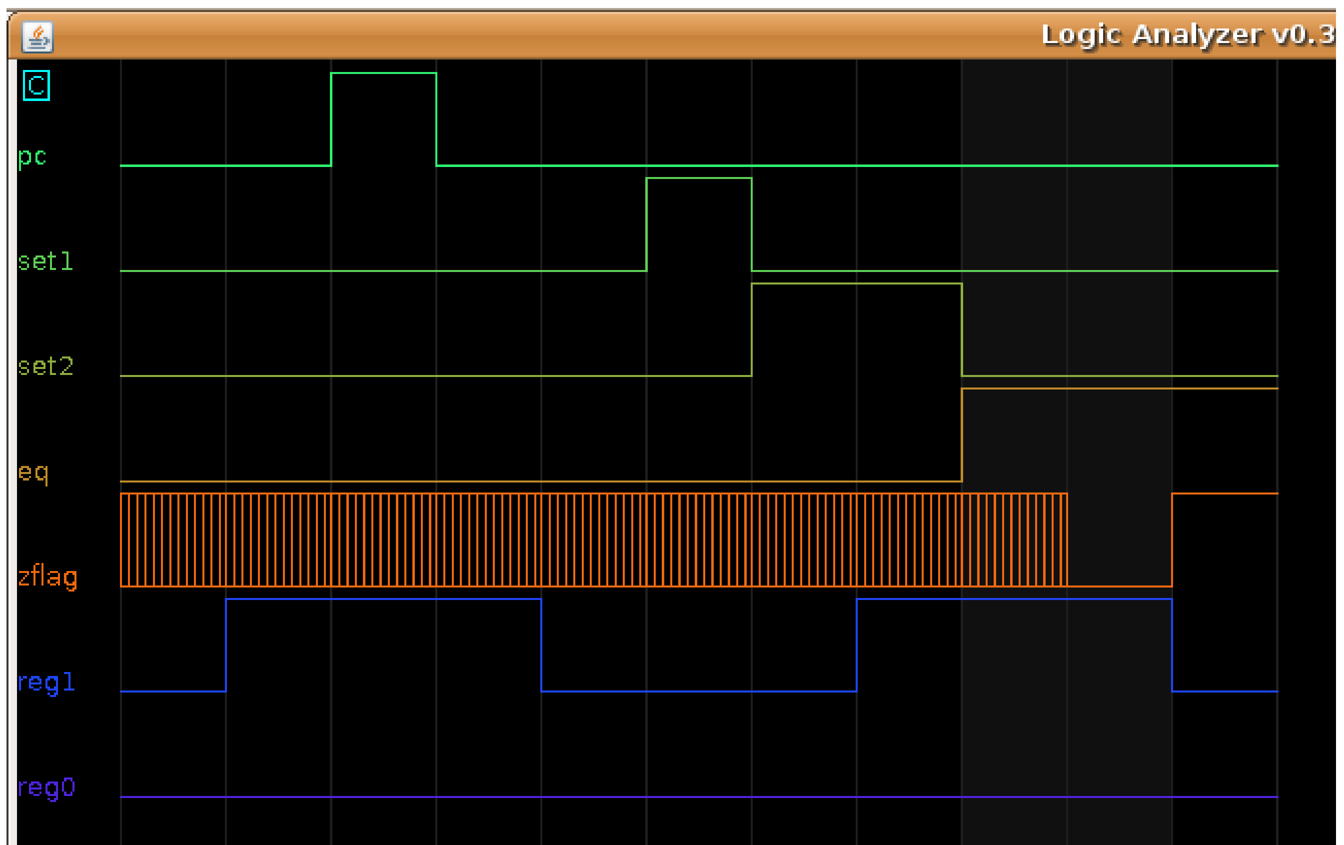
```
010001000   (0 88 hex)
```

Which translates to the assembly-like instruction

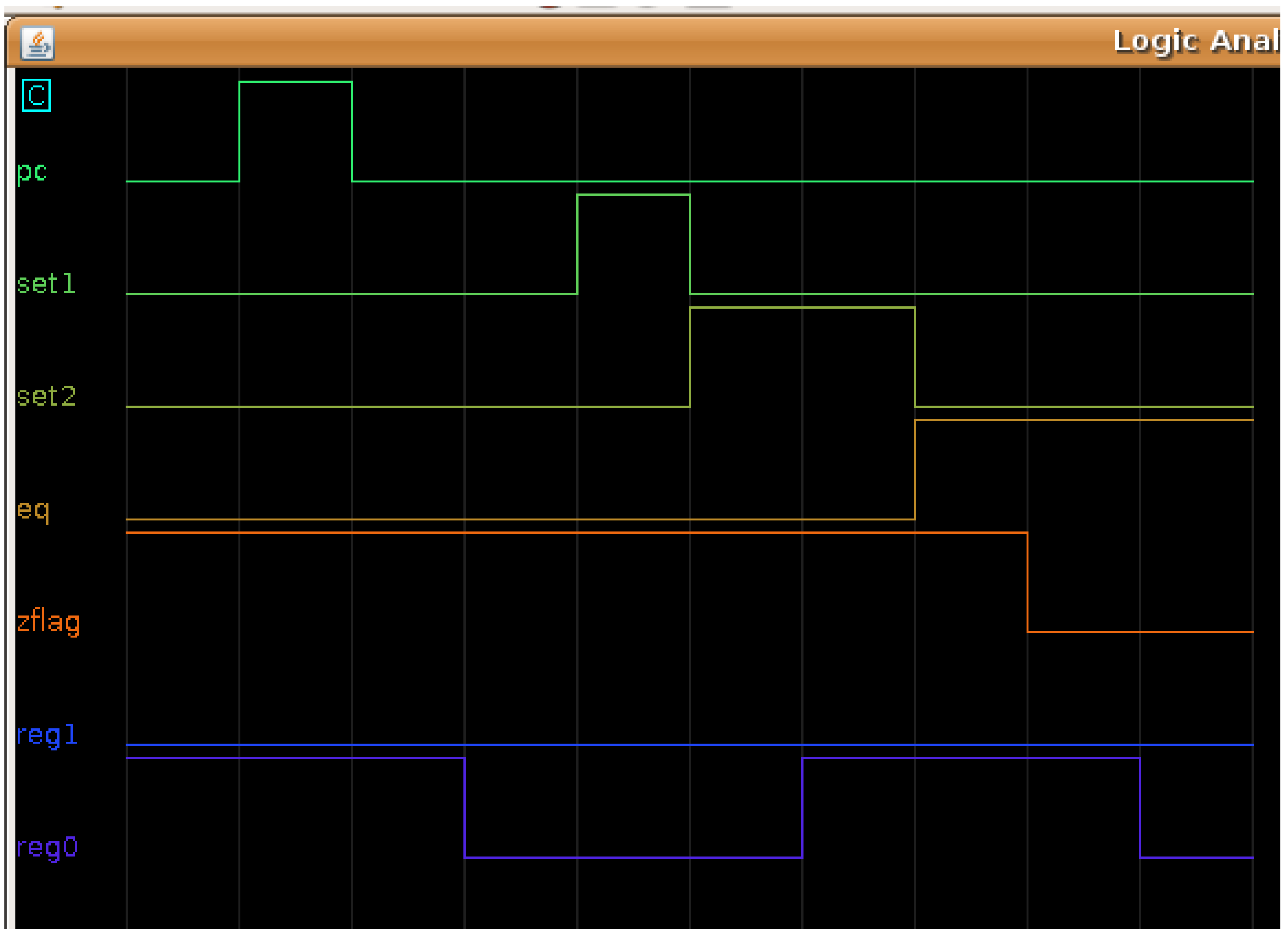
EQ (010) reg 00 (00) reg 10 (10) 00 (padding)

If the two registers contain equal values, the zero flag is set.

Below is a timing diagram of relevant flags. This diagram shows two register values (register 00 and register 10) being compared. In this first diagram, the values are in fact equal and the zero flag is set.



In this second diagram two values are compared that are not equal and the zf is unset (it was set from the first diagram earlier in the program).



LOAD

The LOAD (LD) instruction in this architecture was the first one implemented, and it is relatively straight forward. The LD instruction has the following format:

Inst	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ld	opcode2	opcode1	opcode0	regA1	regA0	data3	data2	data1	data0

The opcode for LD is 001, which references the LD address in microcode which is line 32 of the opcode file. 00100000 is 32 (the opcode constitutes the three most significant bits of the address). The microcode for LD is the following:

```
08
01
```

Which translates to

```
LD
PC
```

The most complicated part of this instruction is the selection logic, which coordinates the LD and ADD values. The only two times a write can occur to a register is one of these two instructions, and the LD flag is used in determining which values should be written.

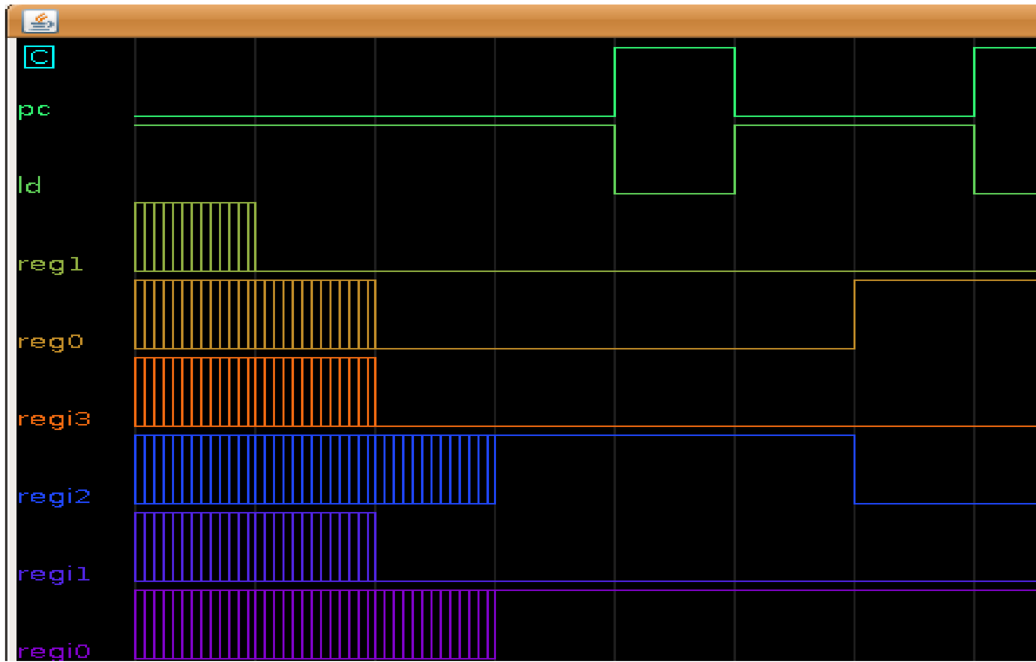
As an example consider the instruction in RAM

```
001000101 (Binary)
0 45 (Hex)
```

Broken up, this instruction translates to the assembly-like code

```
LD (001) 00 (register 00) 0101 (the value 0101)
```

Below is a timing diagram of relevant flags. This particular timing diagram shows the value 0101 being loaded into register 00.



NOP

Inst	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
nop	opcode2	opcode1	opcode0						

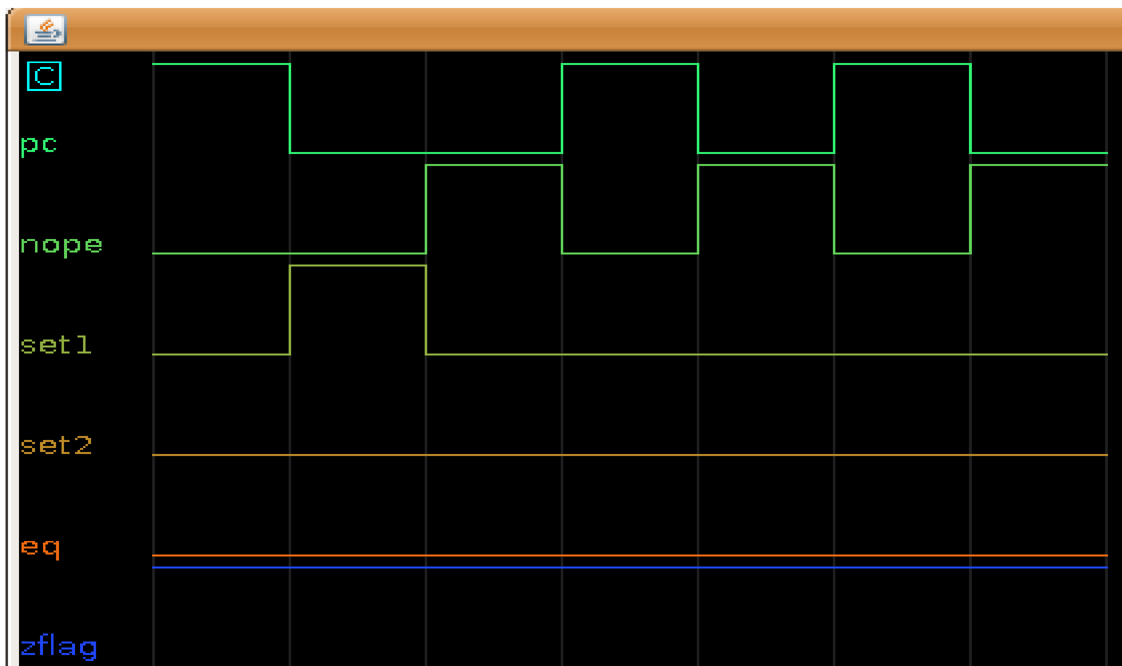
The nop instruction is extremely simple. The only thing it does is increment the program counter. The opcode for nop is 011, which can translate to c0. The instruction points at the microcode located at line 96 of the opcode file. Because I had extra flags, I set an extra nop flag, so there is a signal when a nop is generated (the halt instruction uses the same flag, but does not increment the pc afterward). Here is the microcode on line 96.

80
01

which sets the flags

nop
pc

Below is a timing diagram. The most relevant flags are the pc and nope. However, some other values are included to show that nothing significant is happening elsewhere.



SKIPZ

The SKIPZ instruction has the following format:

Inst	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
skipz	opcode2	opcode1	opcode0						

SKIPZ is an instruction that, if the ZF is set, will skip the next instruction. If the ZF is not set the next instruction will be executed.

The opcode for SKIPZ is 101. Part of this instruction must necessarily be set in the second RAM device.

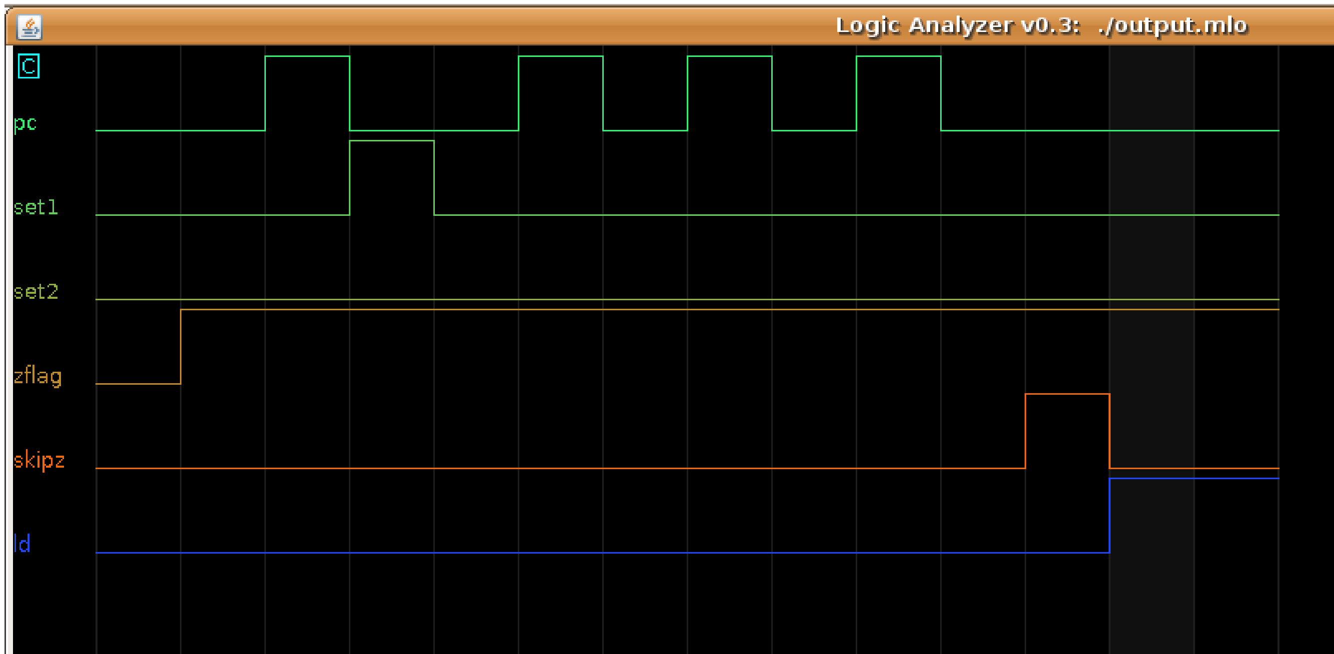
The opcode points to the following microcode, found on line 160 of the opcode file.

```
10
01
```

Which simply sets the SKIPZ flag and then increments the program counter. While the opcode is simple, the hardware logic is a bit more complicated. Although theoretically this instruction should be doable without an ALU, I was unable to implement it correctly using Multimedia Logic (probably due to my short comings). I opted to use a brute-force ALU technique to keep track of the current location added to the number of executed SKIPZs (which are true if both the SKIPZ and the zero flags are set).

Here is a timing diagram of a SKIPZ with the corresponding commented code. This is executed after the EQ above where the registers are equal, and ZF is set. In the code, the next instruction is an add which should be skipped. Next is a LD, which should be executed. i.e., the code corresponds to the following:

```
EQ reg00 reg01
...
skipz
add blah (should be skipped)
ld value
```

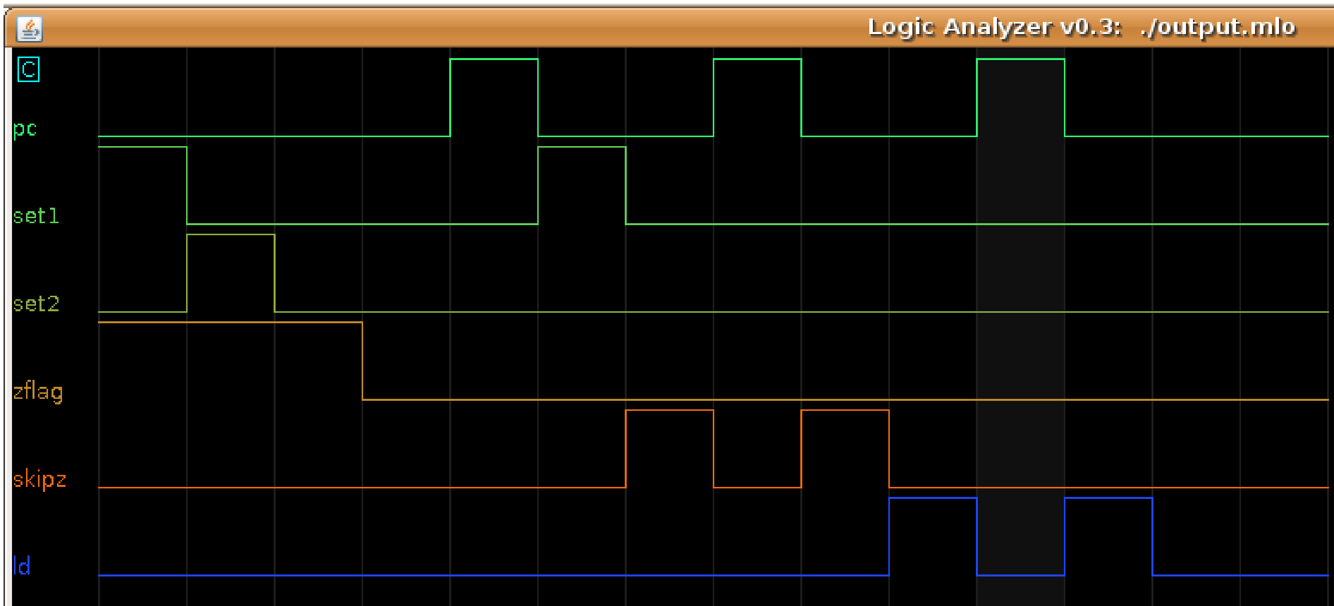


As is visible in the diagram, the ADD is skiped, and the next instruction is LD (otherwise the set1 and set2 flags would be set).

As another example of a SKIPZ, consider the situation where ZF is not set. The following diagram is a snapshot of the following events:

```
EQ reg 01 02 (not equal, ZF unset)
...
SKIPZ
LD blah
HALT
```

In this case, the LD should be executed because the ZF is not set, and this turns out to be the case.



HALT

The HALT instruction has the following format:

Inst	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
halt	opcode2	opcode1	opcode0						

HALT is similar to NOP, except that no instructions are executed after the NOPE flag is set i.e. the PC is never incremented.

The opcode for HALT is 100. It resides on line 128 of the file opcode. The only microcode instruction is to set the NOPE flag. This flag does not control anything, and simply is a notification of this fact. For a timing diagram see the last SKIPZ section.

EXAMPLE PROGRAM

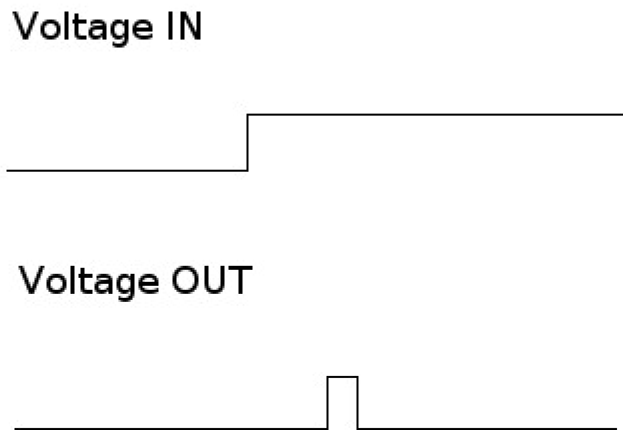
This example is loaded into the memory at the time of submission. This section is a dissection of that program.

```
001000101  | 001 ld 00 reg00 0101 value (hex 0 45)
001010001  | 001 ld 01 reg01 0001 value (hex 0 51)
000000100  | 000 add 00 reg00 01 reg01 00 padding (hex 0 04)
001100110  | 001 ld 10 reg10 0110 value (hex 0 66)
               (same as reg00 has after the add)
010001000  | 010 eq 00 reg00 10 reg01 00 padding (hex 0 88)
011000000  | 011 nop 000000 padding (hex 0 c0)
011000000  | 011 nop 000000 padding (hex 0 c0)
011000000  | 011 nop 000000 padding (hex 0 c0)
101000000  | 101 skipz 00000 padding (hex 1 40)
               (this should jump the next ADD instruction)
000000100  | 000 add 00 rega 01 regb 00 padding (hex 0 04)
               (ADD should have been skipped)
001010000  | 001 ld 01 reg01 0000 value (hex 0 50)
010000100  | 010 eq 00 rega 01 regb 00 padding
               (unset the z flag since !=)
101000000  | 101 skipz (hex 1 40)
               (the next instruction should now be hit)
001000101  | 001 ld 00 rega 0101 value (hex 0 45)
               should be hit)
100000000  | 100 halt 000000 padding (hex 1 00)
```

The simplest way to verify this architecture somewhat works is to go through this example carefully, viewing the lights that LEDs that result in real time. It is difficult to load these into the logic analyzer at the same time, there is too much data and there was insufficient time over the semester for me to fix that and add the functionality to scroll down.

INSUFFICIENCIES

The biggest shortcoming of this program is the write instruction flag on the register device. I was unable to figure out an elegant way of accomplishing this. Theoretically, circuitry could be added that has a slight delay, and a single burst based on the input.



Although this is a seemingly simple device, I was unable to reproduce it in multimedia logic, despite hours of effort. The delay is necessary, because if it is not there it introduces a race condition and does not work. The single burst is necessary for similar reasons. Although I attempted to make the circuit asynchronous by various combinations of tristate devices and flip flops, after hours of solutions that should have worked but did not, I added a push button to substitute for this logic (it was pressed when a write was necessary). This worked flawlessly. After some tuning I swapped the push button out with an oscillator, which seems to work roughly 95% of the time (based on 40 trial runs). If the rise or fall come at the wrong time, bad data is written to a register. I think this is acceptable, as most modern computers are wrong at least 10% of the time based on my own experience.