# Lab Report 3

**Chris Dobson**
**EEL4713**

# Introduction

The main purpose of this lab was to design several different components that will be used later in the course. A 32 bit adder was designed for use in the ALU and perhaps other locations. The 32 bit ALU was designed with 8 different functions. The ALU control logic to interpret the instructions and the function field to direct the ALU was designed. Finally, a 32 bit register file with 32 different registers was designed for the general purpose registers.

# Design and Testing

## 32 Bit Adder

Two 32 bit inputs (in0 and in1) are continuously added together with the result output to at 32 bit port (O). There is no carry or overflow flag, so more was needed in the adder used in the ALU.

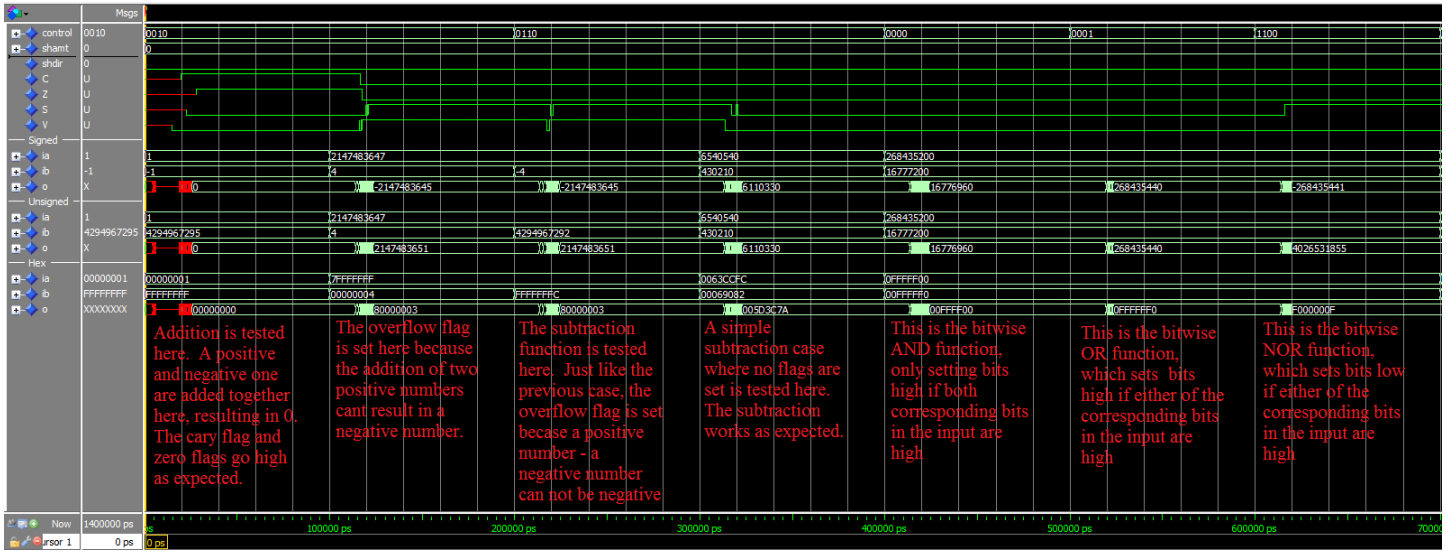
### Simulation

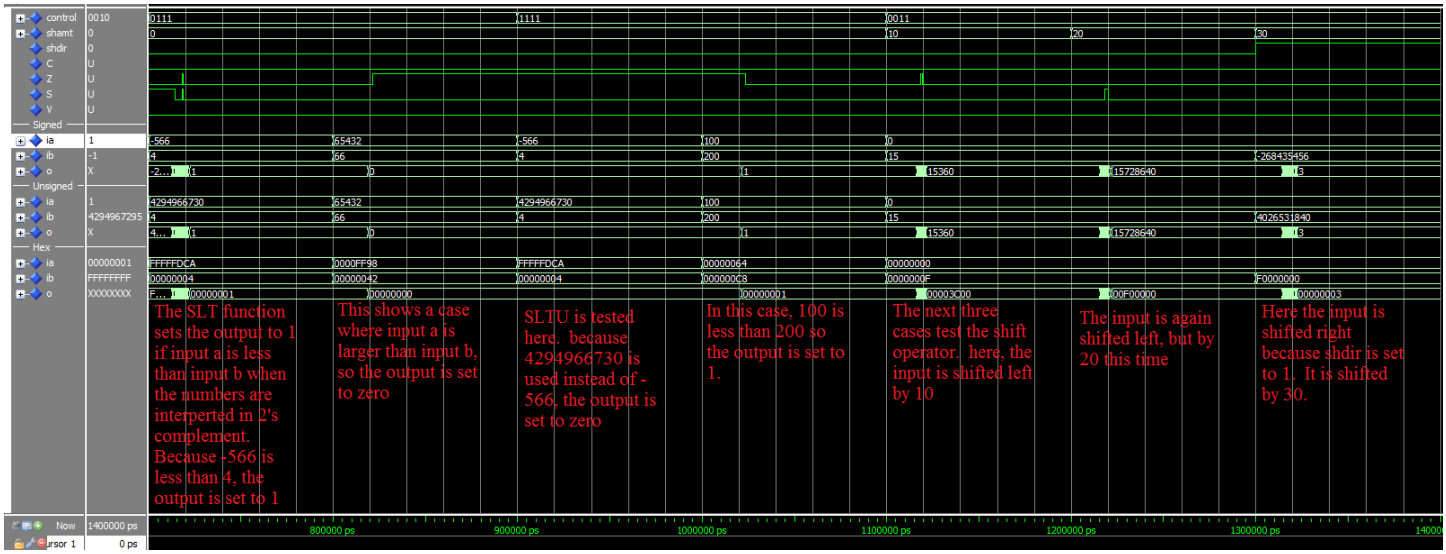

This simulation tests several different possible input combinations, as specified in the lab documents.

## ALU

The ALU performs one of 8 different math functions on two 32 bit inputs depending on a select signal. There is a single 32 bit output as well as 4 different output flags. C is the carry out, Z is the zero flag, S is the negative flag, and V is the overflow flag.

### Simulation





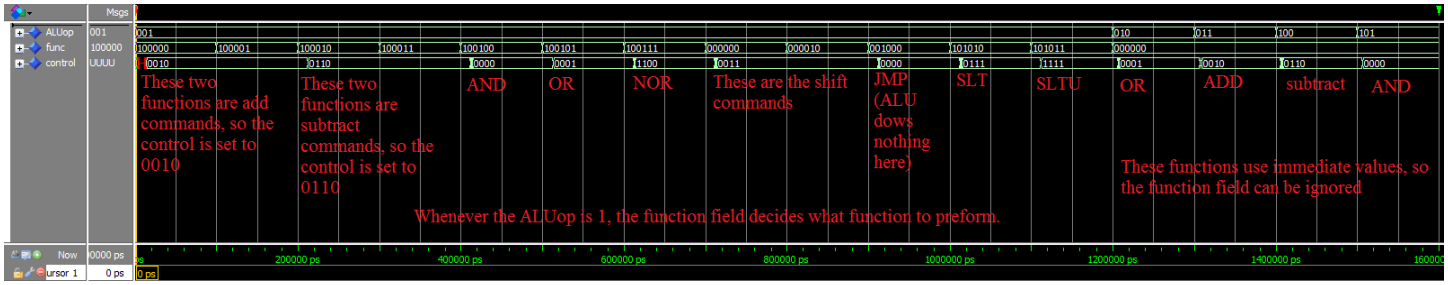

Each of the ALU's 8 different functions are tested here, with all of them operating as expected.

## ALU Control

The ALU control is in charge of decide which function the ALU need to implement depending on the instructions received from the main control logic as well as the function field. The control was just mapped out to the following truth table and implemented with logic equations. The 3 bit ALUop and the 6 bit funct field are inputs and the control is outputted to the ALU.

| instruction | ALUop | | | funct | | | | | | control | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 1 | 0 | 5 | 4 | 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 |
| add | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| addu | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| sub | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| subu | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| AND | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| OR | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| NOR | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| sll | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| srl | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| jr | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | - | - | - | - |
| slt | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| sltu | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| or | 0 | 1 | 0 | - | - | - | - | - | - | 0 | 0 | 0 | 1 |
| add | 0 | 1 | 1 | - | - | - | - | - | - | 0 | 0 | 1 | 0 |
| subtract | 1 | 0 | 0 | - | - | - | - | - | - | 0 | 1 | 1 | 0 |
| AND | 1 | 0 | 1 | - | - | - | - | - | - | 0 | 0 | 0 | 0 |

### Simulation

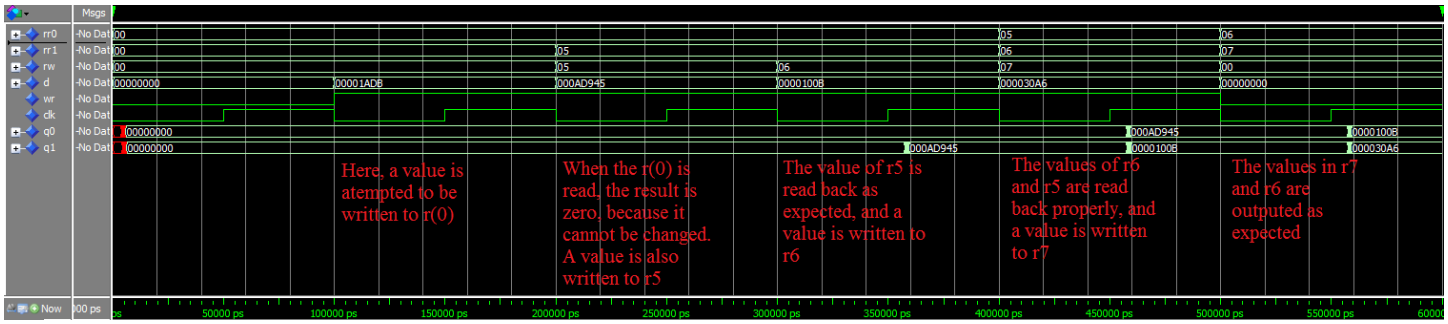



The simulation tests all the different valid ALUop and func combinations.

## Register File

The register file is in charge of storing and returning the values of all the registers. The r0 register cannot be written to and a read of it always returns 0.

**Simulation**




This simulation first attempts to write a value to r0, but the write doesn't go through. It then writes and reads back three other registers.

# Text Book Questions (non-revised 4th edition)

4.2.1) a) The memory instruction, register file, and ALU can be used, but the registers and ALU need some modification to work with 4 registers.

b) The instruction memory, register file, and ALU can be used for this operation.

4.2.2) a) The register file needs to be modified to enable three simultaneous reads. The ALU also needs to be modified to support three inputs for an add instruction.

b) The blocks can be used as is (assuming this ALU can do shifts, if it can't, that would need to be added).

4.2.3) a) The ALU control signal may need to be extended to support an additional function.

b) No new signals are necessary (unless the ALU was extended to do shifts, then ALU the control signal may need to be extended to support an additional function).

4.8.1) a) OR, $2, $0, 128 -- if $2 is equal to 128, then the bit is not stuck low. If $2 is equal to 0, then the bit is stuck low. (bit7 is in the immediate field of the operation.)

b) Set the data at a know memory address to zero, 255 for example. Then run:
lb $2, 255($zero) -- if $2 is equal to zero, then the signal is not stuck at zero. If $2 is equal to 255, then the address from the instruction with the offset added to it was stored into $2, meaning the mem to reg bit was stuck at 0

4.8.2) this test must be different from the previous one because the bit must be set to the opposite value of the stuck state it is being tested for.

a) OR, $2, $0, 0 -- if $2 is equal to 128, then the bit is stuck high. If $2 is equal to 0, then the bit is not stuck high. (bit7 is in the immediate field of the operation.)

b) This can't be tested for because for an expected value to come out of memory, the memory output must be enabled. Because the memory is always disabled when memtoreg signal is set to 0, the value stored in a register would be unpredictable. A special debugging instruction could be added to make this testable.

4.8.3) a) This could be done, but it would significantly reduce the performance of the processor. Many of the math functions that use the immediate field would only take one additional instruction to fix, though bitwise instruction would require loading a register and then doing the operation. If all the registers were in use, one would have to be temporarily stored in memory. The load and store instructions that use offset functionality would be tricky, the destination would have to be reduced by 128 if it used bit 7, and a register would have to be used to add that offset back on.

b) This cannot be worked around because the registers would never be able to read from the ALU, preventing any of the math functions from working.

# Bug Spim

The "lui" command was found to be defective. Instead of loading the most significant 16 bits, it loads the least significant. The code used to find this is below. A bad address error was generated on the second line, and the register was manually checked and found to be wrong.

```
lui $2, 0x1000
sw $3, 0($2)
```

The "sh" command was found to only store 8 bits opposed to 16. The following code revealed the error when the registers were analyzed after the program was run. When 255 was spotted in $16 instead of -1, that appropriate memory address was scanned and found to hold the wrong data.

```
addi $2, $zero, -1
sw $2, 0($3)            #3 was set to 0x1000 0000
sh $2, 4($3)
sb $2, 8($3)
lw $15, 0($3)
lh $16, 4($3)
lb $15, 8($3)
```

It was found that "beq" actually branches when the its two registers are not equal, and does not branch when its two registers are equal. The code that found the error is below. After the code was run, $4 was checked and found to be 1 instead of 0.

```
addi $2, $zero, 55
addi $3, $zero, 55
add $4, $zero, $zero
beq $2, $3, BEQ_TEST
addi $4, $zero, 1
BEQ_TEST
```

It was found that "jal" does not set $31 to the proper return address. It operates just like a regular jump instruction. The error was found in the following code when $31 was found to be zero.

```
add $4, $zero, $zero
jal JAL_TEST
addi $4, $zero, 1
JAL_TEST
```

The ALU bug was not found. All of the core instructions were tested one at a time with multiple different inputs by one and compared to spim. Overflows and sign changes were tested whenever applicable as well, but the error was unseen.