The g3l Instruction Set Architecture

Meta

Author: Gregory N. Schmit

Date: 26 Nov 2017

Introduction

The goal of the g3l instruction set architecture (ISA) is to provide a simple application programming interface (API) to execute a single program that finds the largest number (and index of) in a given array in memory, and finds the remainder of a division operation (again, stored in memory).

My primary goal was to reduce the amount of instructions provided by the ISA.

Answers to Questions

1. I did not complete the goals of the project. I got hung up on Logicworks being a garbage software program and not being able to check that any step actually works the way through. The major bug was how logicworks handles the busses (since I used that to make design easier).
2. ISA Review -- see below.
3. I did not implement PC and branches because of being sidetracked by trying to get my other stuff working. It would've been easy (I had it down on paper) -- just an adder and a mux for branches and another mux with a loopback for the halt instruction.
4. Structure Overview (everything is in the CPU file):
    - Registers
    - Main ALU
    - Control Circuit
    - Ancillary PC control circuits
5. Program execution results: negative, ghostrider.
6. Answers to the Questions In these questions, I am no longer looking for you to convince me that your design is wonderful, but rather I am looking at how effectively you critique your own design.
    i. No, but I wish that I did. I would've optimized two registers to be able to swap on a condition, and then I would've optimized a shifting instruction (for the second

program).

ii. I don't know, but I do know that I screwed up on the second program since I never used the shifting method, so the dynamic instruction count would be massive.
iii. I was terrible at optimizing for ease of design. The number of muxes for swapping registers is quadratic in the number of registers, so that was a headache. I also required a lot of constants in the ISA (more control circuitry), so I should've made things more generic (at the expense of instruction length).
iv. It would be very hard to extend my design to a multi-cycle implementation because things are not segregated very well already. For example, my swap on condition instruction requires the ALU to run but the operation happens in the regfile, so that would require possibly another ALU.
v. That is kinda what I optimized for, and so my ISA turned out to be complex in silly ways that could've been avoided.
vi. I believe that my swap on condition instruction takes the longest since it requires the aLU to run and the swap muxes to resolve before the swap happens.
vii. I wouldn't have implemented any swap register instructions, so more memory read/writes would have happened.
viii. I didn't get to actually running any programs because I was trying to make the circuits function and I could not. I think the problem was something to do with how logicworks handles the busses.
ix. Reflect on this project(1-3) experience:
    a. I realized that Logicworks is an awful POS software program that I never want to use again. I didn't like graphically building the circuits; I would've rather had a language to write everything in. I didn't like how if you changed the subcircuit it didn't sync everywhere.
    b. Focus on building everything very early to give yourself time to tshoot. Don't use Logicworks if you have another option.
    c. I got to understand the complexities of CPU datapath and control. The understanding that comes from designing it yourself is something that is hard to get by simply studying an existing system.

Registers (8 total 12-bit registers)

There are eight 12-bit (tribble) registers.

- r0 - zero
- r1 - 3rd large stack
- r2 - 3rd large stack
- r3 - 3rd large stack
- r4 - 3rd large stack entry
- r5 - for loop counter
- r6 - mem ptr

- r7 - mem target

Instructions

Instruction	Argument	Description	Encoding (5-bit)	Section
6set0000		Set $r6 to 0b000000000000	0b11000	11xxx - Setting + Halt
6set0111		Set $r6 to 0b000000000111	0b11001	11xxx - Setting + Halt
7set0110		Set $r7 to 0b000000000110	0b11010	11xxx - Setting + Halt
7set0101		Set $r7 to 0b000000000101	0b11011	11xxx - Setting + Halt
7set0011		Set $r7 to 0b000000000011	0b11100	11xxx - Setting + Halt
7set0010		Set $r7 to 0b000000000010	0b11101	11xxx - Setting + Halt
7set0001		Set $r7 to 0b000000000001	0b11110	11xxx - Setting + Halt
halt		Halt	0b11111	11xxx - Setting + Halt
neginf	$rlox	Set low register ($rlox is element of $r0-$r3) to lowest number	0b000xx	000xx - Special Set to Negative Infinity

incr6		Increment $r6	0b00100	0010x - Incrementing
decr5		Decrement $r5	0b00101	0010x - Incrementing
8decr6		Decrement $r8 by 6	0b00110	00110 - Special Increment
lt6hi4		Load tribble pointed to by $r6 to $r4	0b01000	0100x - Load from Memory
lt6hi5		Load tribble pointed to by $r6 to $r5	0b01001	0100x - Load from Memory
st37		Store $r3 to memory pointed to by $r7	0b01010	01010 - Special Store to Memory
swplt12		Swap $r1 and $r2 if $r1<$r2	0b10000	100xx - Register Swap
swplt23		Swap $r2 and $r3 if $r2<$r3	0b10001	100xx - Register Swap
swplt34		Swap $r3 and $r4 if $r3<$r4	0b10010	100xx - Register Swap
swp36		Swap $r3 and $r6	0b10011	100xx - Register Swap
sub37		Subtract $r4 from $r3, store in $r3	0b10100	10100 - Special Subtract
b5nzj7		Branch if $r5 != 0 to PC-$r7	0b01100	011xx - Branching

b34neqj7		Branch if $r3 != $r7 to PC-$r7	0b01101	011xx - Branching
b34ltejf7		Branch if $r3 < $r4 to PC+$r7	0b01110	011xx - Branching
j2		Jump two instructions backwards (PC-2)	0b01111	011xx - Branching