The g3l Instruction Set Architecture

Meta

Author: Gregory N. Schmit

Date: 1 Oct 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).

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

Format of Memory and Instructions

Registers and memory are tribble-addresssable. The bus for instructions is 8-bits long, however only 5-bits are relevant.

.data starts at `0b00000000` and .text starts at `0b10000000`. Memory is only 256 tribbles total.

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	11000	11xxx - Setting + Halt
6set0111		Set $r6 to 0b000000000111	11001	11xxx - Setting + Halt
7set0110		Set $r7 to 0b000000000110	11010	11xxx - Setting + Halt
7set0101		Set $r7 to 0b000000000101	11011	11xxx - Setting + Halt
7set0011		Set $r7 to 0b000000000011	11100	11xxx - Setting + Halt
7set0010		Set $r7 to 0b000000000010	11101	11xxx - Setting + Halt
7set0001		Set $r7 to 0b000000000001	11110	11xxx - Setting + Halt
halt		Halt	11111	11xxx - Setting + Halt
neginf	$rlox	Set low register ($rlox is element of $r0-$r3) to lowest number	000xx	000xx - Special Set to Negative Infinity
incr6		Increment $r6	00100	0010x - Incrementing

decr5		Decrement $r5	00101	0010x - Incrementing
8decr6		Decrement $r8 by 6	00110	00110 - Special Increment
lt6hi4		Load tribble pointed to by $r6 to $r4	01000	0100x - Load to Memory
lt6hi5		Load tribble pointed to by $r6 to $r5	01001	0100x - Load to Memory
st37		Store $r3 to memory pointed to by $r7	01010	01010 - Special Store to Memory
swp36		Swap registers $r3 and $r6	10000	10000 - Special Register Swap
sub37		Subtract $r4 from $r3, store in $r3	10001	10001 - Special Subtract
b5nzj7		Branch if $r5 != 0 to PC-$r7	01100	011xx - Branching
b34neqj7		Branch if $r3 != $r7 to PC-$r7	01101	011xx - Branching
b34gtejf7		Branch if $r3 <= $r4 to PC+$r7	01110	011xx - Branching
j2		Jump two instructions backwards (PC-2)	01111	011xx - Branching

Answers to Questions

1. How large is the main memory?
   - Since everyting is tribble-addressable, there is a theoretical limit to 4096 tribbles. However, the hadware only implements 256 tribbles, 128 of which are `.data` and 128 of which are `.text`.
2. In what ways did you optimize for dynamic instruction count (goal 1)
   - I implemented a partial-bubble-sort algorithm which provides linear complexity (like the naive 3-loop search method), but drastically reduces the complexity constant since after you find on of the largest three values, fewer computations are required on the rest of the data.
3. In what ways did you optimize for ease of design (goal 2)?
   - I focused on reducing the quantity of instructions that are available, and as a result, all of the instructions are restricted to specific values. This ensures that the hardware design will be simple, even if it requires some hardcoding of constant values.
4. If you optimized for anything else, what and how?
   - I got my instruction size down to 5-bit. This was a bit of code golf.
5. Your chief competitor just announced a load-store ISA with two explicit operands (one source register is also used as destination), four registers (i.e., a 2-bit register specifiers), and 16 instructions (4 opcode bits). Explain why your ISA is better.
   - My ISA is simpler and uses less memory for a program, since the instructions are all 5-bit. Also, my ISA optimizes a partial bubble sort algorithm, which the competitors does not.
6. What do you think will be the bottleneck in your design? That is, what don't you have that you will miss the most if you were to have to write other longer programs?
   - g3l optimizes a single program (really, two programs), but writing other programs would be painful, since the instructions are not general at all.
7. What would you have done differently if you had 2 more bits for instructions?
   - None, since I optimized my instructions to make them shorter.
8. 2 fewer bits?
   - None, since 6-bits is still longer than my instruction set.

Source Code

```
.data  # 0x00
A: .tribble 5250
B: .tribble 5235
buffer0: .tribble 0
R: .tribble -1
buffer1: .tribble 0
third_largest_value: .tribble 0b100000000000
third_largest_index: .tribble 0b100000000000
Array1Count: .tribble 10
Array1: .tribble 200, -3, 15, 9, -1200, 45, 0, 6, 12, -2
```

```
.text
f_thirdlarge:
  # get value and index of third largest value in array
  # set r6 to array count pointer (7)
  6set0111
  # set bubble stack to neg infinity
  neginf $r1
  neginf $r2
  neginf $r3
  lt6hi5  # set 5 to array count (deref r6)
  incr6  # increment r6 to get it to point to the array
  # setup r7 to point to for loop (relative: 6 instructions back)
  7set0110
  for_item_in_array:
    lt6hi4  # load item to r4 (overwrite, lc $r4, $r6)
    # bubble up item in r1-r4
    swplt34
    swplt23
    swplt12
    incr6  # increment r6
    decr5  # decrement r5
    b5nzj7
  7set0101  # set r7 to point to value store
  st37  # load r3 to memory of r7
  6set0111  # set r6 to array count pointer (7)
  lt6hi5  # set 5 to array count (deref r6)
  incr6  # increment r6 to get it to point to the array
  # setup r7 to point to for loop (relative: 6 instructions back)
  7set0010
  for_item_in_array2:
    lt6hi4  # load item to r4 (overwrite, lc $r4, $r6)
    incr6
    b34neqj7
  8decr6
  swp36 # swap R3 and R6
  7set0110  # set r7 to third largest index pointer
  st37  # store value in r7

f_remainder:
  # find remainder of A/B
  zero6  # set r6 to 0
  neginf $r3  # set r3 to lowest possible
  lt6hi4  # deref 6, load to r4 (A)
  swplt34  # (A is now in r3)
  incr6  # go to next addr (B)
  lt6hi4  # deref 6, load to r4 (B)
  7set0011  # set 7 to 2 (for branch)
  while_a_lte_b:  # if A>=B sub B from A, repeat
    b34gtejf7
    sub34
    j2
  end_while:
  # store A to remainder in memory
  # 7set0011 -- already there!
  st37
halt
```