Elate

Goals strived for: Booth’s multiplication, 3-bit registers, 3-bit opcodes

Goals achieved: 4-bit opcodes, 5-bit registers, hashing for large numbers

Overall philosophy: do less work

Devs

Melissa Ly A11908353

Alan Chen A12045305

Sawandeep Kaur A11601962

### Instruction Formats

R type

* eg: push r1
* 4 bits of opcode followed by 5 bits of register specification

I type

* eg: push immediate 0
* 4 bits of opcode followed by 5 bits of immediate specification

B type

* eg: blt loop
* 4 bits of opcode followed by 5 bits defining a label address

### Operations Supported

**General Operations**

* set\_memory\_ptr [imm]: 0000
  + The immediate hashes to another number, so only key memory locations can be accessed.
  + This operation sets the memory pointer (r0) to the hashed number, then pushes the contents of that memory location into the stack.

|  |  |  |  |
| --- | --- | --- | --- |
| Immediate | Hashed location | Immediate | Hashed location |
| 00000 | 1 | 00111 | 19 |
| 00001 | 2 | 01000 | 20 |
| 00010 | 3 | 01001 | 32 |
| 00011 | 4 | 01010 | 95 |
| 00100 | 5 | 01011 | 127 |
| 00101 | 6 | 01100 | 128 |
| 00110 | 7 |  |  |

* store [reg]: 0001
  + loads the value in the register to where the memory pointer (r0) is currently pointing
* push [reg]: 0010
  + pushes the register into the stack
* add [reg]: 0011
  + adds the two top items in the stack
  + stores the result in the register
  + pops the addends
  + sets the overflow flag
* set [reg]: 0100
  + stores the topmost value on the stack inside the register
* push\_immediate [imm]: 0101
  + pushes the immediate onto the stack
* pop [imm]: 0110
  + pop the immediate number of items from the stack
* blt [label]: 0111
  + checks if the top value on the stack is less than the penultimate value on the stack
    - If true, branch to label
    - Else, continue
* inc [reg]: 1000
  + increments the value in the register by 1
  + If the register is 00000, also push the contents of the register to the stack

**Product Operations**

* and\_and\_shift [reg]: 1001
  + The value in the register is an index, i
  + Performs a bitwise 'and' with the i'th bit (starting from the LSB) from the penultimate stack item and the top stack item
  + Logical shifts the result 8-i bits to the right and pushes the result to the stack
  + Shifts the result of the bitwise 'and' left by i and pushes that to the stack
* add\_overflow [reg]: 1010
  + Adds the bit from the overflow to the register
  + Adds the top value in the stack to the register
  + Pops the top value

**String Match Operations**

* contains [label]: 1011
  + Check if the top value on the stack is a substring of the penultimate value on the stack
    - If so, branch to target address

**Closest Pair Operations**

* sub [reg]: 1100
  + Subtracts the two top items on the stack
  + Stores the result in the register
* abs[reg]: 1101
  + Gets the absolute value of the value in the register
  + Stores the result in the register
  + Pushes the value to the stack

**Other Operations**

halt: 1110

to be determined: 1111

### Internal Operands

15 are registers are supported, and each one is eight bits wide. One register, r0, is designated for the memory location. Whenever the value inside this register changes, the value that the address is pointing to is pushed onto the stack. We also have one flag for overflow. It's set on the "add" instruction and is used to carry bits in the product problem.

### Control Flow (Branches)

We have one less-than branch for specifically navigating loops, while the contains instruction also branches based on the results of the contains function. Instead of having the target address in the instruction, we hash the branch address in a similar fashion to the hashing of memory locations explained above. They are detailed in the table below.

|  |  |
| --- | --- |
| Hash code | Label / Address |
| 00000 | A\*B |
| 00001 | lowerAB\*C |
| 00010 | upperAB\*C |
| 00011 | loop |
| 00100 | next |
| 00101 | incj |
| 00110 | incarray |
| 00111 | endinner |
| 01000 | innerloop |
| 01001 | outerloop |

### Addressing Modes

We have absolute addressing since all of our branches branch to a specific, pre-known address. When not branching, we support sequential execution.

### Main Memory

We use the stack stack to store data, so we don't need any memory beyond the 128 bytes where the input for our problems is stored.

### Optimizing Dynamic Instruction Count

We reduced dynamic instruction count by stringing our most-used commands together. For example,

there is no instance where we would shift a number left without shifting it right, so we combine those in one instruction and push the results onto the stack. This works around the requirement that we may not write to more than one register at a time, since the stack is not a register.

### Optimizing for Ease of Design

In order to optimize for ease of design, we customized our instructions so that we could reduce several convoluted operations into a single instruction. This makes for cleaner code and less room for error.

### Optimizing for Short Cycle Time

We optimized for short cycle time by minimizing the times we need to access memory (a costly endeavour).

We did not optimize for anything else.

### Competition

Our ISA is superior to a load-store ISA with two operands as while we only have one operand per instruction, each instruction performs many operations. We support 32 registers (magnitudes more than four).

### Biggest General-Purpose Weakness

The biggest general-purpose weakness in our design is that our instructions take advantage of the specificness of the tasks, which allows us to combine many operations but would overstep many general-purpose programs.

### Two Bits

Given two more bits, we could increase the number of instructions we have. That way we can include instructions that would decrease the instruction count in our programs, thus reducing the execution time of each of the programs.

If we had only seven bits for instructions, we would have fewer registers and need to place even heavier use on the stack and the stack pointer.

### Classification

We have a stack machine.

### Example

For example, C = A + B would look like the following:

|  |  |  |
| --- | --- | --- |
| // A is in r1  // B is in r2  // C is in r3  push r1  push r2  add r3 | Stack:  top ------ bottom  r1  r2, r1  [empty] | Machine code  0010 00001  0010 00010  0011 00011 |

### Product

Pseudocode:

A = (1); // r1

B = (2); // r2

C = (3); // r3

int lowerSumAB = 0; // r4

int upperSumAB = 0; // r5

// i = r6

for (int i = 0; i < 8; i++) { // A\*B

sum = A[i] & B;

sumLowerAB += sum << i; // set overflow flag

sumUpperAB += overflow;

sumUpperAB += sum >> 8-i;

}

for (int i = 0; i < 8; i++) { // lowerAB \* C

sum = lowerAB[i] & C;

sumLowerLowerABC += sum << i; // set overflow flag

sumUpperLowerABC += overflow;

sumUpperLowerABC += sum >> 8-i;

}

for (int i = 0; i < 8; i++) { //upperAB \* C

sum = upperAB[i] & C;

sumUpperABC += sum << i;

// don't care about upper upper ABC because that's overflow

}

sumUpperABC += sumUpperLowerABC;

(4) = sumUpperLowerABC;

(5) = sumLowerLowerABC;

Assembly code:

push\_immediate 0

set r1 // i = 0

set r2 // lowSumAB = 0

set r3 // upperSumAB = 0

set r4 // sumLowerLowerABC = 0

set r5 // sumUpperLowerABC = 0

set r6 // sumUpperABC = 0

set\_memory\_ptr 1 // push A onto stack

set\_memory\_ptr 2 // push B onto stack

// ----------------------------------------------------------------------------------------------------------------

A\*B: // B is on the top of the stack, with A right below it

and\_and\_shift r1 // sum = A[i] & B; push sum >> 8-1, push sum << i

push r2

add r2// sumLowerAB = sum << i + r2; pop sum<<i and r2 and set overflow

add\_overflow r3// if sumLower += results in overflow, add it and sum >> 8-i to upper sum, pop

inc r1 // i++

// check for branch

push r1

push\_immediate 7

// state of the stack: 7, i, B, A, 0

blt A\*B // branch to beginning of loop unless i > 7 (i == 8)

// -------------------------------------------------------------------------------------------------------------------

push 0

set r1 // i = 0

set\_memory\_ptr 3 // push C onto stack

push r2 // push the lower bits of A\*B onto stack

// current state of the stack: lowerAB, C, 0

// there are more, but we'll forget about them for now because there are no requirements for memory leaks

// -----------------------------------------------------------------------------------------------------------------

lowerAB\*C: // same routine as before

and\_and\_shift r1 // sum = lowerAB[i] & C; push sum >> 8-1, push sum << i

push r4

add r4// sumLowerLowerABC = sum << i + r4; pop sum<<i and r4 and set overflow

add\_overflow r5 // if sumUpLowABC += results in overflow, add it and sum >> 8-i to upper sum, and pops

inc r1 // i++

// check for branch

push r1

push\_immediate 7

// state of the stack: 7, i, C, lowerAB, 0

blt lowerAB\*C // branch to beginning of loop unless i > 7 (i == 8)

// --------------------------------------------------------------------------------------------------------------------

push 0

set r1 // i = 0

set\_memory\_ptr 3 // push C back onto stack instead of popping; fewer instructions and less chance

//for mistakes

push r3 // push the upper bits of A\*B onto stack

// current state of the stack: upperAB, C, 0

// -------------------------------------------------------------------------------------------------------------------------

upperAB\*C: // same routine as before

and\_and\_shift r1 // sum = upperAB[i] & C; push sum >> 8-1, push sum << i

push r6

add r6// sumLowerLowerABC = sum << i + r6; pop sum<<i and set overflow

// don't care about the rest because it's overflow

inc r1 // i++

// check for branch

push r1

push\_immediate 7

// state of the stack: 7, i, B, A, 0

blt upperAB\*C // branch to beginning of loop unless i > 7 (i == 8)

// r7 will be the upper bits in location 4

push r5

push r6

add r7 // r7 = r5 + r6, or upper + upperLower

// r4 will remain the lower bits

set\_memory\_ptr 4 // move r0 to 4

store r7 // stores r7 into [r0]

set\_memory\_ptr 5

store r4 // stores r4 into [r0]

Regardless of the numbers multiplied, this program takes 205 instructions.

String Match

Pseudocode:

string str = string\_to\_find; //4-bit string being searched for

int num = 0; (add $t1, $0,$0) //number of entries in the array that contain the 4-bit string

for(int i = 0; i < arr.length; i++) {

if(arr[i].contains(str))

num++;

}

return num;

Assembly code:

push\_immediate 0

set r1 //bit index

push\_immediate 0

set r2 //number of matches in the array of strings

push\_immediate 32

push\_immediate 0

set r3

set\_memory\_ptr 6 // 4-bit string

set\_memory\_ptr 32 //arr[0]

Set\_memory\_ptr 64

push r3

loop:

pop 2

contains next

inc r2

next:

set\_memory\_ptr 6

Set\_memory\_ptr 32

inc r0

set\_memory\_ptr 64 //array length

inc r3

push r3

blt loop

set\_memory\_ptr 7

store r4

The dynamic instruction count of this program is 588 for all values.

### Closest Pair

Pseudocode:

int closest = max\_int;

for(int i = 0; i < arr.length - 1; i++) {

for(int j = i+1; j < arr.length; j++) {

if(i < 1)

closest = abs(arr[j] – arr[i])

if(abs(arr[j] – arr[i]) < closest)

closest = abs(arr[j] – arr[i]);

}

}

return closest;

Assembly Code:

set\_memory\_ptr 255

set r1 //r1 = closest

push\_immediate 0

set r2 //r2 = i

push\_immediate 1

set r3 //r3 = 1

push\_immediate 19

set r4 //r4 = 19 = arr.length - 1

push\_immediate 20

set r5 //r5 = 20 = arr.length

set\_memory\_ptr 128

set r6 //r6 = a[0]

outerloop:

push r3

push r2

//branch if i < 1, so that if i = 0 the memory pointer is not incremented since it is already at 128 and //a[0] is at 128

blt incj

//set the memory pointer to the right location by incrementing the pointer until it reaches index i so //that it is at a[i]

set\_memory\_ptr 128 //reset memory pointer location to a[0]

push\_immediate 0

set r7 // r7 = 0 (counter)

incarray:

inc r0 //increment memory location to the appropriate location by looping

set r6 //until r7 = r2(counter = i)

inc r7

push r2

push r7

blt incarray //branch if r7 < r2 (counter < i )

incj:

push r2

push r3

add r8 //r8 = i + 1 = j

//loop through the array subtracting a[j] from a[i] and if it is less than the value stored in r1 (closest) //set r1 to this difference, continue looping while j < array length

innerloop:

push r6 //r6 = a[i]

inc r0 //r0 = a[j]

sub r9 //r9 = a[j] - a[i]

abs r9 //r9 = abs(a[j] - a[i])

push r1 //r1 = closest

blt endinner //branch if r1 < r9 (closest < abs(a[j] - a[i]))

push r9

set r1 //set r1 to r9 (closest = abs(a[j] - a[i])

endinner:

push r5

inc r8 //j++

push r8

blt innerloop //branch if r8 < r4 (j < 20)

//increment i and continue looping through the array if it is less than array length - 1 since the last //element in the array will already have been compared to the rest of the elements

push r4

inc r2 //i++

push r2

blt outerloop //branch if r2 < r4 ( i < 19)

//put the calculated closest distance into memory location 127

set\_memory\_ptr 127 //sets memory pointer to 127

store r1 //stores r1 to location 127 (r1 = closest)

The dynamic instruction count of this program is 3,200.