**CSE 141L - Lab 1**
Frank Mezzatesta A98035022
Angela To A10657395

January 23 2015

# FMAT 8-BIT INSTRUCTION SET ARCHITECTURE



# 1. INTRODUCTION

The FMAT Instruction Set Architecture is an 8-bit redesign of the standard MIPS Instruction Set. Our ISA is uniquely implemented to allow certain performance optimizations.

The FMAT ISA provides 8 bit wide instructions and 16 internal registers, which although is somewhat limited in instruction set breadth and memory, is sufficient for the implementation and execution of our three basic programs (product, string_match, and closest_pair). The FMAT Instruction Set supports essential operations including basic computational, logical, data control and transfer operations standard to the MIPS architecture as well as a few specialized instructions unique to the FMAT ISA that enables extended register functionality and use in light of the restricted width of our 8-bit processor. For instance, special operations such as **focus**, **otype rframe**, and **otype sframe** enables the user to effectively use more registers and thus handle more data than what a normal 8 bit instruction space would conservatively allow. In addition, FMAT aims to reduce overall dynamic instruction count thereby hopefully producing more efficient looped code and speedups.

# 2. INSTRUCTION FORMATS

<table>
<tr><th colspan="11">INSTRUCTION TYPES SPECIFICATION</th></tr>
<tr><th rowspan="2">TYPE</th><th rowspan="2">NAME</th><th colspan="8">ADDRESS FORMAT IN BITS</th><th rowspan="2">DESTINATION</th></tr>
<tr><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>RR</td><td>REGISTER REGISTER</td><td colspan="4">OPCODE</td><td colspan="2">REG#</td><td colspan="2">REG#</td><td>Reuse RS</td></tr>
<tr><td>AF</td><td>ADDRESS FLAG</td><td colspan="4">OPCODE</td><td>FRAME</td><td colspan="2">REG#</td><td>FLAG</td><td>Specified</td></tr>
<tr><td>A</td><td>ADDRESS</td><td colspan="4">OPCODE</td><td colspan="2">FRAME</td><td colspan="2">REG#</td><td>Implicit</td></tr>
<tr><td>I</td><td>IMMEDIATE</td><td colspan="4">OPCODE</td><td colspan="4">IMM</td><td>Implicit</td></tr>
<tr><td>II</td><td>IMMEDIATE IMMEDIATE</td><td colspan="4">OPCODE</td><td colspan="2">IMM</td><td colspan="2">IMM</td><td>Implicit</td></tr>
<tr><td>IF</td><td>IMMEDIATE FLAG</td><td colspan="4">OPCODE</td><td colspan="3">IMM</td><td>FLAG</td><td>Implicit</td></tr>
</table>

<table>
<tr><th colspan="11">INSTRUCTION TYPES EXAMPLES</th></tr>
<tr><th rowspan="2">TYPE</th><th rowspan="2">EXAMPLE INSTRUCTION</th><th colspan="8">ADDRESS FORMAT IN BITS</th><th rowspan="2">DESTINATION</th></tr>
<tr><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>RR</td><td>ADD</td><td colspan="4">0001</td><td colspan="2">00</td><td colspan="2">01</td><td>reg[0][0]</td></tr>
<tr><td>AF</td><td>INCDEC</td><td colspan="4">0011</td><td>0</td><td colspan="2">00</td><td>0</td><td>reg[0][0]</td></tr>
<tr><td>A</td><td>FOCUS</td><td colspan="4">1011</td><td colspan="2">11</td><td colspan="2">00</td><td>$R</td></tr>
<tr><td>I</td><td>LOWER</td><td colspan="4">1100</td><td colspan="4">1101</td><td>reg[$R]</td></tr>
<tr><td>II</td><td>JUMPIF</td><td colspan="4">1111</td><td colspan="2">00</td><td colspan="2">10</td><td>N/A ($PC)</td></tr>
<tr><td>IF</td><td>SHIFT</td><td colspan="4">1110</td><td colspan="3">100</td><td>0</td><td>reg[$R]</td></tr>
</table>

# 3. OPERATIONS

| INSTRUCTIONS | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| TYPE | OPCODE | INSTR | 1 bit | 1 bit | 1 bit | 1 bit | EXTCODE | EXTINSTR | FLAG | |
| | | | | | | | | | if 0 | if 1 |
| IF | 0000 | OTYPE | (extinstr)<IMM> | | | (flag)<IMM> | 000 | NOP | | |
| IF | 0000 | OTYPE | (extinstr)<IMM> | | | (flag)<IMM> | 001 | HALT | | |
| IF | 0000 | OTYPE | (extinstr)<IMM> | | | (flag)<IMM> | 010 | TBD | | |
| IF | 0000 | OTYPE | (extinstr)<IMM> | | | (flag)<IMM> | 011 | OVRFLW | | |
| IF | 0000 | OTYPE | (extinstr)<IMM> | | | (flag)<IMM> | 100 | CLEAR | 0s | 1s |
| IF | 0000 | OTYPE | (extinstr)<IMM> | | | (flag)<IMM> | 101 | NSET | 0 | 1 |
| IF | 0000 | OTYPE | (extinstr)<IMM> | | | (flag)<IMM> | 110 | RFRAME | 0 | 1 |
| IF | 0000 | OTYPE | (extinstr)<IMM> | | | (flag)<IMM> | 111 | SFRAME | 2 | 3 |
| RR | 0001 | ADD | (reg#)<IMM> | | (reg#)<IMM> | | | | | |
| RR | 0010 | SUB | (reg#)<IMM> | | (reg#)<IMM> | | | | | |
| AF | 0011 | INCDEC | (frame)<IMM> | (reg#)<IMM> | | (flag)<IMM> | | | inc | dec |
| RR | 0100 | AND | (reg#)<IMM> | | (reg#)<IMM> | | | | | |
| RR | 0101 | OR | (reg#)<IMM> | | (reg#)<IMM> | | | | | |
| A | 0110 | NOT | (frame#)<IMM> | | (reg#)<IMM> | | | | | |
| AF | 0111 | SEED | (frame)<IMM> | (reg#)<IMM> | | (flag)<IMM> | | | $CL | $CR |
| RR | 1000 | MOVE | (reg#)<IMM> | | (reg#)<IMM> | | | | | |
| RR | 1001 | LOAD | (reg#)<IMM> | | [(reg#)<IMM>] | | | | | |
| RR | 1010 | STORE | [(reg#)<IMM>] | | (reg#)<IMM> | | | | | |
| A | 1011 | FOCUS | (frame)<IMM> | | (reg#)<IMM> | | | | | |
| I | 1100 | LOWER | <IMM> | | | | | | | |
| I | 1101 | UPPER | <IMM> | | | | | | | |
| IF | 1110 | SHIFT | (amount)<IMM> | | | (flag)<IMM> | | | left | right |
| II | 1111 | JUMPIF | ($J#)<IMM> | | (condition)<IMM> | | | | | |

## FURTHER INSTRUCTION LISTING/ANALYSIS:

**OTYPE NOP -** Does not read, and does not write.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| `otype nop` | `zzz` | `otype nop      # nothing happens` |

**OTYPE HALT -** Stop the machine from executing the current program.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| `otype halt` | `stop execution` | `otype halt     # end of program` |

**OTYPE TBD -** It is a mystery.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| otype tbd | ??? | otype tbd # ??? |

**OTYPE OVRFLW -** Adds the overflow bit to the focused register if flag is 1, subtracts if flag is 0.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| otype ovrflw, 0 | if flag = 0 (add):<br>$[$R]++<br><br>if flag = 0 (substract):<br>$[$R]-- | # $[0][0] = $[0][1] = 200<br>add 0, 1 # result will overflow<br>otype ovrflw, 0# $[$R]++ |

**OTYPE CLEAR -** Clear the bits of $[$F][0] to 0s if flag is 0, or 1s if flag is 1.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| otype clear, (fill)<IMM> | $[$R] <= 0 - fill | otype clear, 0 # $[$F][0] is now all zeros |

**OTYPE NSET -** Set the $N bit.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| otype nset, 1 | $N <= 1 | otype nset, 1 # $N is now 1 |

**OTYPE RFRAME -** Changes the currently-focused-on registers to the specified REG FRAME.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| otype rframe, (frame)<IMM> | $F <= frame | otype rframe, 1 #$F = 1 |

**OTYPE SFRAME -** Changes the currently-focused-on registers to the specified SYS FRAME.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| otype sframe, (frame)<IMM> | $F <= frame+2 | otype sframe, 1 #$F = 3 |

**ADD:** Adds the contents of two registers, rs and rt, and stores the result back into register rs.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| add rs, rt | $[rs] <= $[rs] + $[rt] | # $1 = 1, $2 = 2<br>add 1, 2 # $1 = 1 + 2 |

**SUB:** Subtracts the contents of two registers, rs and rt, and stores the result back into register rs.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| sub rs, rt | $[rs] <= $[rs] - $[rt] | # $1 = 2, $2 = 1<br>sub 1, 2 # $1 = 2 - 1 |

**INCDEC:** Increments or decrements (depending on flag) the contents of the register specified by the integer immediates. If flag = 0, the value of the register is incremented, else if flag = 1, the value of the register is decremented.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| incdec (frame)<IMM>, (reg)<IMM>, (flag)<IMM> | if flag = 0 (Increment):<br>$[frame][reg]++ | # $[0][1] = 1<br>incdec 0, 1, 0 # $[0][1]++ |

| | | |
|---|---|---|
| | if flag = 1 (Decrement):<br>$[frame][reg]-- | # $[1][3] = 1<br>incdec 1, 3, 1 # $[1][3]-- |

**AND:** Performs a bitwise AND operation on the contents of registers rs and rt; stores the results back into register rs.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| and rs, rt | $[rs] <= $[rs] & $[rt] | # $1 = 1, $2 = 1<br>and $1, $2 # $1 = 1 & 1 |

**OR:** Performs a bitwise OR operation on the contents of registers rs and rt; stores the results back into register rs.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| or rs, rt | $[rs] <= $[rs] \| $[rt] | # $1 = 1, $2 = 1<br>or $1, $2 # $1 = 1 \| 1 |

**NOT:** Performs a bitwise NOT operation on the contents of registers rs; stores the results back into register rs.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| not rs | $[rs] <= !($[rs]) | # $1 = 1<br>not $1 # $1 = 0 |

**SEED -** Sets $CL or $CR (depending on the value of the flag immediate) to be the register address or value (depends on whether $CL or $CR) of the register specified by the integer immediates - frame# and reg#. If flag = 0, register $CL is set to the address of $[frame#][reg#], else if flag = 1, the value of the register $CR is set to value of $[frame#][reg#]. This instruction is meant to optimize loop performance, by preemptively storing conditional comparison values beforehand.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| seed (frame)<IMM>, (reg)<IMM>,<br>(flag)<IMM> | if flag = 0 ($CL):<br>$CL <= (frame<<2) \| reg<br><br>if flag = 1 ($CR):<br>$CR <= $[frame][reg] | seed 0, 1, 0 # $CL = 0001<br><br>seed 1, 3, 1 # $CR = $[1][3] |

**MOVE:** Copies the contents of register rt into register rs.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| move rs, rt | $[rs] <= $[rt] | # $1 = 0, $2 = 1<br>move $1, $2 # $1 = 1 |

**LOAD**: Loads Byte (8 Bits) from main memory into register rs from the address of register rt.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| load rs, rt | R[rs] <= mem[offset] | load $1, $2 # $1 = mem[$2] |

**STORE:** Store Byte (8 Bits) from contents of register rt into the memory address, rs

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| store rs, rt | mem[rs] <= $[rt] | store $1, $2 # mem[$1] = $2 |

**FOCUS:** Sets $F to the address of $[frame#][reg#]

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|

| | | |
|---|---|---|
| focus (frame)<IMM>, (reg)<IMM> | $[$F] = (frame<<2) \| reg | focus 3, 0 # $F = 00001100 |

**LOWER:** Sets the lower four bits of the current "focused" register to the immediate value.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| lower <IMM> | $[$F] \| IMM | # $[0][0] = 1111 1111<br>lower 5 # $[0][0] = 1111 0101 |

**UPPER:** Sets the upper four bits of the current "focused" register to the immediate value.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| upper <IMM> | $[$F] \| (IMM<<4) | # $[0][0] = 1111 1111<br>upper 5# $[0][0] = 0101 1111 |

**SHIFT:** Shift the bits of the current "focused" register by the specified amount, in the specified (flag) direction. Left if a 0 flag, right is a 1 flag. An amount of 0 has a special meaning of 8, since we will never have a use for shifting a value by 0 places.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| shift (amount)<IMM>,<br>(direction)<IMM> | $[$F] <= $[$F]<<amount | # $[0][0] = 1111 1111<br>shift 4, 1 # $[0][0] = 0000 1111 |

**JUMPIF:** Jumps to the address stored in $[3][reg] if $CR is related to $CL by the immediate comparison code. Comparison codes: 0: equal, 1: less than, 2: greater than, 3: will always return true. If you need to simply jump without checking for a condition, set #2 to be 3. If you want to negate the comparison, use otype nset to set $N to 1.

| SYNTAX | MEANING | EXAMPLE |
|---|---|---|
| jumpif (reg)<IMM>, (compare)<IMM> | if ($CL compare,$C $CR):<br>$PC <= $[3][reg] | # $CL > $CR, $C = 0<br>jumpif 3, 2 # jump to $[3][3]<br><br>jumpif 3, 3 # jump to $[3][3] |

# 4. Internal Registers

In our design, we utilize 16 registers: $reg[0,1,2,3,4,5,6,7] are usable, $reg[8,9,10,11] are implicitly used by the system, and only ever written to indirectly through otype instructions. $reg[12,13,14,15] are usable, but should only be used for their specific purpose, to hold jump addresses. Each register holds 8 bits.

<table>
<tr><th colspan="16">REGISTERS</th></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
<tr><td colspan="4">FRAME 0<br>(REG FRAME 0)</td><td colspan="4">FRAME 1<br>(REG FRAME 1)</td><td colspan="4">FRAME 2<br>(SYS FRAME 0)</td><td colspan="4">FRAME 3<br>(SYS FRAME 1)</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>N|CL</td><td>CR</td><td>R</td><td>F</td><td>J0</td><td>J1</td><td>J2</td><td>J3</td></tr>
</table>

**<u>WHAT ARE REGISTER FRAMES?</u>**

The machine starts off in register frame 0. If you mention any 2-bit register address when in register frame 0, here are the registers you're talking about

00 register 0
01 register 1
10 register 2
11 register 3

It would be nice to have more registers. We obviously can't access more than 4 registers though, since we can only afford to support up to 2-parameter instructions if the register addresses can each fit in 2 bits. This is where register frames come in. You can change your register frame with the **otype rframe #1** and **otype sframe #1** instructions. If I issue **otype rframe 1**, then my register frame will now be 1 (frame number is stored in the $F implicit register). NOW, if I mention 2-bit register addresses in any instructions, I will actually be talking about

00 register 4
01 register 5
10 register 6
11 register 7

There are certain instructions (instructions that can spare more bits for a single register address) that directly address registers.

## REG FRAME REGISTERS

These are general-purpose registers that the programmer may use however they wish.

## SYS FRAME REGISTERS

These are special registers that are written to ahead of time. Their values are called up to add extra information when we issue complex instructions later on. It's okay to read from and write to SYS FRAME 1 registers, but you really shouldn't be writing to SYS FRAME 0 registers. You *can*, but there shouldn't be a need to, since these values are written to implicitly after certain **otype** instructions are run; using the appropriate **otype** instruction, you may set these registers from anywhere in a program with only one instruction, but manually focusing on a SYS FRAME 0 register, setting its lower bits, and setting its upper bits would take three instructions. Because not all these values take up a full 8 bits, these registers are sometimes shared.

**R [4bits] (Register)**: the address of the focused register, because the letter F was already taken.
**N [1bit] (Negation)**: whether comparisons in **jumpif #1, #2** instructions should be negated, 0 for normal, 1 for negated. We need this simply because we don't even have a single bit to spare in the **jumpif #1, #2** instruction.
**F [2bits] (Frame)**: the current frame, simply describes our current frame value: {0,1,2,3}
**CL [4bits] (Compare Left)**: the address of the register used on the left side of the comparison (dynamic value).
**CR [8bits] (Compare Right)**: the value used on the right side of the comparison (static value).

These values are primarily intended for use with the **jumpif #1, #2** instructions, so that (1) we have enough bits to specify the comparison we're making and (2) so that the comparison can be done in the same instruction as the jump, by specifying which values we wish to compare *outside of the loop*.

**J0, J1, J2, J3 [each 8bits] (Jumps)**: 8-bit addresses, pointing to locations that we wish to jump to when using loops or conditionals. For example, when we want to write a loop, we would set $J0 and $J1 ahead of time so that we can save those bits for other purposes when we actually issue the **jumpif #1, #2** instruction.

## "THAT'S TOO TROUBLESOME"

Our original design only used 4 registers. You may write programs that ignore REG FRAME 1. You might have to perform extra loads and stores than you otherwise would have had to perform, though. You must interact with the SYS FRAMEs. The SYS FRAMEs were included in order to make loops take up fewer instructions. You could, for example, use memory, through loads and stores, to keep track of everything you need, but instructions like jumpif will only pay attention to SYS FRAME 1 registers. These constructs are a necessity because we are unable to fit an adequate amount of information into the parameter slots of each 8-bit instruction. We must store some information beforehand.

# 5. CONTROL FLOW (BRANCHES)

**WHAT TYPE OF BRANCHES ARE SUPPORTED**

FMAT ISA does not support branches (in the conventional sense) rather it provides a different approach to controlling data flow. We support a **jumpif reg<IMM>, compare<IMM>** instruction that takes an immediate register value in order to designate which register address the operation will jump to, and a conditional comparison value that determines whether that jump should occur. In effect, FMAT's single **jumpif** operates as a **bg, bl, beq, ba** combined in a single statement and does so depending on the set comparison value. For instance, if the comparison flag is set to 0, we will be comparing $CL == $CR (similar to **beq**); if the comparison flag is set to 1, then we will be comparing $CL < $CR (similar to **bl**), etc. If the comparison flag is set to 3, no regard will be taken to the condition and we will be jumping directly to the specified register address (similar to **ba**). (Refer to Section 3 for a more detailed specification). The contents of registers $CL and $CR are seeded before the comparison / jump occurs.

**HOW ARE TARGET ADDRESSES CALCULATED**

Due to the bit constraints of our design, were not able to follow our **jumpif** operation with an immediate address; instead, we compute the target address of the jump by taking the address of $[3][reg] (Frame 3, Register reg) as specified by the immediate register value, reg.

**MAXIMUM BRANCH DISTANCE SUPPORTED**

From any point in a program, you can branch to any other point from instruction 0 to instruction 255. Because we must specify the absolute jump address in 8 bits, we are limited to 256 addresses. If we find a need to jump farther in future programs, there are ways we could revise our ISA to allow different types of jumps.

# 6. ADDRESSING MODES

**SUPPORTED MEMORY ADDRESSING MODES**

**Register Addressing:** FMAT ISA supports register addressing for its operations in which the address of a single register is set or "focused" on beforehand. For instance, **focus 0, 1** designates the register to be used for an **incdec** operation. Register addressing is used in particular for instructions that require the movement or setting of a large amount of data (or more so that we can support for FMAT's 2-bit register).

**Base (Memory) Addressing:** Base addressing is used for FMAT's **load** and **store** instructions which each take in two register values, **rs** and **rt**. Here, the address is computed as address <= $[rt], where **rt** is the base register.

**HOW ARE MEMORY ADDRESSES CALCULATED?**

Memory addresses are absolute, so if we load and pass in $[0][0], we grab the value from $[0][0], say 134, and access mem[134], which is the 134th Byte in memory.

# 7-16. QUESTIONS AND ANSWERS

7. **How large is the main memory?**

   256 Bytes: main memory is byte-addressable with 2^8=256 addresses

8. **In what ways did you optimize for dynamic instruction count?**

   The main goal of this ISA is to require as few operations inside a loop as possible. We've gotten loops down to very low line-count amount, which will make up for any setup required before the loop.

9. **In what ways did you optimize for the ease of design?**

   To be honest, ease of design took a backseat on this one. Although we allow instruction names to be up to 6 characters (ease of readability) and we include common instructions found in 32-bit architectures (or at least mimic them), we had to introduce some complexity in order to work successfully with a tiny 8-bit instruction space that was to still be capable of handling Bytes of data.

**10. In what ways did you optimize for short cycle time?**
We only include simple instructions. For example, we do *not* include a multiply, divide, or modulus instruction. Probably our slowest instruction (besides memory accesses) will be the jumpif instruction, which has to compare the values of two registers, a $C flag, and write to the program counter. We only allow the load and the store instructions to access memory; every other instruction may only deal with parameters and register contents.

**11. If you did optimize for anything else, what was optimized and how? (optional)**
The only optimizations worth noting are the tight loop optimizations.

**12. Why is your ISA better than a competitor's ISA (2 explicit operands =>1 source operand, 1 source/destination operand; 4 registers => 2-bit register specific; 16 instr =>4 opcode bits)**
Our ISA matches the 2 explicit operands, and goes beyond. Certain instructions that are data-heavy accept one, large, 4-bit operand, while others accept three operands and contain flags. The format depends on the opcode. We offer 8 general purpose registers for our programmers. In total, we offer 16 registers, but of those, 8 are general purpose, 4 are implicit, and 4 are specialized. We offer 23 instructions; the 0000-opcode instruction has been extended to include, in actuality, 8 sub instructions, each of which either need no operands, or merely need a single, 1-bit flag operand.

**13. What do you think will be the biggest general purpose weakness in your design?**
Much setup is required before any real work can be done. Jump addresses must be seeded ahead of time. Comparative **jumpif #1, #2** instructions must have their operands seeded ahead of time. Although this leads to tight loops, it also shows quite obviously in a bulky static instruction count. Another weakness that must be mentioned is that we can only access 256 jump locations and 256 Bytes of main memory, because jump addressing and main memory addressing are both absolute.

**14. What would you have done differently if you had 2 more bits for instructions/ 2 fewer bits?**
The nice thing about this ISA is that it's ways of handling data can be generalized to even fewer bits. We could run the same system on a 6-bit instruction width, for example. Our ways of handling limited operand space include setting implicit registers ahead of time, and register frame shifting. If we had 2 more bits, not too much would have actually changed. Maybe a few instructions could have benefitted from not having to call upon implicit registers. We also would have had 4x more addressable jump locations and 4x more addressable main memory locations. If we had 4 more bits, everything would have changed. If we had 4 more bits, we would have actually been able to work directly with Bytes right in our instruction operands, so the majority of this ISA would be different; we might not have had a need for frame shifting at all.

**15. Can you name/classify your machine in any of the classical ways (i.e stack machine, accumulator, register, load-store)?**
Our machine is a load-store architecture.

**16. Give an example of assembly language instruction in your machine, translate to machine code.**

# simple program to add mem[0] and mem[1], and place result in $[0][0]

```
# we're dealing with REG FRAME 0 here
line01      otype rframe, 0   # $F = 0
line02      focus 1, 0        # $R = &$[1][0]

# seed mem[0] into $CL
line03      otype clear, 0    # $[1][0] = 0
```

```
line04      load 0, 0        # $[0][0] = mem[0]

# load mem[1] into $CR
line05      incdec 1, 0, 0   # $[1][0]++ (now equals 1)
line06      load 1, 0        # $[0][1] = mem[1]

line07      add 0, 1         # $[0][0] = $[0][0] + $[0][1]

line08      otype halt       # EOP, exit program

# machine code:
line01  0000 1100
line02  1011 0100
line03  0000 0110
line04  1001 0000
line05  0011 1000
line06  1001 0100
line07  0100 0001
line08  0000 0010
```

# 17. PROGRAM: PRODUCT

Example:
```
252 * 170 * 85 = 3641400
= (11 0111) 1001 0000 0011 1000 = 36920
```

**A * B * C = (AB>>8 * C)<<8 + ((AB<<8)>>8 * C)**

```
ok 11111100 * 10101010 * 01010101
ok 1010011101011000 * 01010101
ok (1010011100000000 + 01011000) * 01010101

ok (10100111 * 100000000 * 01010101) + (01011000 * 01010101)
ok (10100111 * 01010101)<<8 + (01011000 * 01010101)
ok 0011011101110011<<8 + 0001110100111000

(00110111)0111001100000000 + 0001110100111000
```

Register Assignment:

| frame | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| reg | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 |
| var | ptr | A,B,C | ABlo | prodlo | i | N/A | ABhi | prodhi |

```
Math note: doing this calculation with 8-bit registers
A * B * C = (AB>>8 * C)<<8 + ((AB<<8)>>8 * C)
prodhi <= (ABhi * C)
prodlo <= (ABlo * C)
```

Pseudo Code:
```
AB = 0
for i from 0 to A
       ABlo += B (overflow into ABhi)
for i from 0 to C
       prodhi += ABhi
       prodlo += ABlo (overflow into prodhi)
```

FMAT Assembly:
```
       INIT:
              !-- Initialize variables and seed $CL and $CR
line001       otype  rframe, 0     # REG FRAME 0
line002       otype  clear, 0      # $reg0(ptr) = 0
line003       move   2, 0          # $reg2(ABlo) = 0
line003       move   3, 0          # $reg3(prodlo) = 0
line004       incdec 0, 0, 0       # $reg0(ptr) = 1
line005       otype  rframe, 1     # REG FRAME 1
line006       otype  clear, 0      # $reg0(i) = 0
line007       seed   1, 0, 0       # Seed $reg0(i) into $CL
line008       move   2, 0          # $reg2(ABhi) = 0
line009       move   3, 0          # $reg3(prodhi) = 0
line010       otype  rframe, 0     # REG FRAME 0
line011       load   1, 0          # $reg1(A) <= MEM[ptr=1]
line012       incdec 0, 0, 0       # ptr++
line013       seed   0, 1, 1       # Seed $reg1(A) into $CR
line014       load   1, 0          # $reg1(B) <= MEM[ptr=2]
```

```
line015       incdec 0, 0, 0      # ptr++

              !-- Set up conditional negation flag
line016       otype  nset, 1      # if the condition is NOT met, break

              !-- Set jump(to) addresses
line017       focus  3, 0         # $J0 store &LOOP1 (line024)
line018       upper  1
line019       lower  8
line020       focus  3, 1         # $J1 store &END1 (line030)
line021       upper  1
line022       lower  14

              !-- Set up overflow target
line023       focus  1, 2         # focus on ABhi

LOOP1($J0):
line024       jumpif 1, 2         # jump to $J1(END1) if NOT i < A
line025       otype  nop          # we can actually do without this, but it's clearer with it
line026       add    2, 1         # ABlo += B (overflow into ABhi)
line027       otype  ovrflw, 0    # ABhi++ if overflow
line028       jumpif 0, 3         # reloop
line029       incdec 1, 0, 0      # i++ (nop slot)
END1($J1):

line030       load   1, 0         # $reg1(C) <= MEM[ptr=3]
line031       incdec 0, 0, 0      # ptr++
line032       seed   0, 1, 1      # Seed $reg1(C) into $CR
line033       otype  rframe, 1    # REG FRAME 1
line034       otype  clear, 0     # $reg0(i) = 0
line035       otype  rframe, 0    # REG FRAME 0

              !-- Set jump(to) addresses
line036       focus  3, 2         # $J2 store &LOOP2 (line042)
line037       upper  2
line038       lower  10
line039       focus  3, 3         # $J3 store &END2 (line052)
line040       upper  3
line041       lower  4


LOOP2($J2):
line042       jumpif 3, 2         # jump to $J3(END2) if NOT i < A
line043       otype  nop          # we can actually do without this, but it's clearer with it
line044       otype  rframe, 1    # REG FRAME 1
line045       add    3, 2         # prodhi += ABhi (overflow into the abyss)
line046       otype  rframe, 0    # REG FRAME 0
line047       focus  1, 3         # focus on prodhi
line048       add    3, 2         # prodlo += ABlo (overflow into prodhi)
line049       otype  ovrflw, 0    # prodhi++ if overflow
line050       jumpif 2, 3         # reloop
line051       incdec 1, 0, 0      # i++ (nop slot)
END2($J3):

line052       otype  rframe, 1    # REG FRAME 1
line053       focus  1, 0         # ptr = 4
```

```
line054       lower  4
line055       upper  0
line056       store  0, 3         # MEM[ptr=4] <= $reg1(prodhi)

line057       otype  rframe, 0    # REG FRAME 0
line058       focus  0, 0         # ptr = 5
line059       lower  5
line060       upper  0
line061       store  0, 3         # MEM[ptr=5] <= $reg1(prodlo)

line062       otype  halt         # party’s over, everyone go home
```

**What is the dynamic instruction count of this program if the value in location 1 is 53 and location 2 is 17 and location 3 is 42?**

Dynamic Instruction Count for 53 * 17 * 42:
static23 loop6 static12 loop10 static11
<u>Dynamic Instruction Count:</u> 788 = 23 + (53*6 + 2) + 12 + (42*10 + 2) + 11
<u>Static Instruction Count:</u> 62
This count is very dependent on the parameters, and what order the parameters appear in. If we were terribly worried about making this value as small as possible, we would have a preliminary sorter to place the smallest number third, the largest number second, and the remaining number first. For 42 * 53 * 17, the count would be
23 + (42*6 + 2) + 12 + (17*10 + 2) + 11 = 472

# 18. PROGRAM: STRING MATCH

Example:
```
array: 1111 1010 1111
try to find: 1010
find:               0000 1010
loop#  arr extract  shifted
1      1111 1010    0000 1111
2      1111 0101    0000 1111
3      1110 1011    0000 1110
4      1101 0111    0000 1101
5      1010 1111    0000 1010 MATCH
6      0101 111X    0000 0101
7      1011 11XX    0000 1011
8      0111 1XXX    0000 0111
9      1111 XXXX    0000 1111
```

Register Assigments:
```
-----------------------------------------------------------------------------------------------
frame  0            0            0            0            1            1            1
reg    0            1            2            3            0            1            2
-----------------------------------------------------------------------------------------------
var    ptr          pattern byte       i            length       hits         r_ptr
-----------------------------------------------------------------------------------------------
```

Pseudo Code:
```
ptr = 0
$CR = 0000 1010
loop (i = arr.size - 3) {
       $reg0 <= mem[ptr]
       $CR <= $reg0>>4
       compare $CL and $CR
              matches++ if equal
       ptr++
}
```

FMAT Assembly:
```
       INIT:
line001       otype  rframe, 0      # $F= 0
line002       otype clear   0      # $reg0 = 0 (ptr)
line003       move   1, 0           # $reg1 = 0 (pattern)
line004       move   2, 0           # $reg2 = 0 (byte)
line005       move   3, 0           # $reg3 = 0 (i)
line006       otype  rframe, 1      # $F=1
line007       otype clear, 0 # $reg4 = 0 (length)
line008       mov    1, 0           # $reg5 = 0 (hits)

                     !-- Store length of bit stream into $reg4
line009       focus  1, 0
line010       upper  3              # $reg4 = 64 - 3 ….can we just hardcode this ? pls
line011       lower  13

                     !-- Load pattern string from memory into $CL, located in M[6] (lower 4 bits)
line012       focus  0, 0
line013       lower  6             # $reg0(ptr) = 6
line014       load   1, 0           # $reg1 = M[ptr=6]
line015       focus  1, 0
line016       upper  0              # $reg1 = 0000 xxxx, get 4 lower bits of M[6]

                     !-- Initialize ptr = M[32], increment onwards
line017       otype clear   0      # Reset $reg0 (ptr)
```

```
line018        focus  0, 0
line019        upper  2             # $reg0(ptr) = 32

                      !-- Set jump(to) addresses (LABELS)
line020        focus  3, 0          # Set $F[0] = &WHILE (line 029)
line021        upper  1
line022        lower  13
line023        focus  3, 1          # Set $F[1] = &END_WHILE (line 43)
line024        upper  2
line025        lower  13
line026        focus  3, 2          # Set $F[1] = &SKIP_IF (line 38)
line027        upper  2
line028        lower  6
line029        seed   1, 0, 0# $CL = $reg4

       WHILE:
line030        jumpif 1, 0          # Break if  i =(>)  (bit_string_length-3), go to END_WHILE
line031        seed   1, 0, 0# $CL <= $reg1(pattern), nop place

                      !-- Now, get 4 bits from bit string starting from M[ptr]
line032        load   2, 0          # $reg2 <= M[ptr], $reg2 now holds first byte of bit_string
line033        focus  0, 2
line034        seed   0, 2, 1# $CR = $reg2
line035        shift  4, 1          # $reg <= $reg2>>4

                      !-- Compare $CL, $CR, increment num_hits if a match
line036        otype  nset, 1# Negate comparison
line037        jumpif 2, 0          # if $CL (!)= $CR, goto SKIP_IF
line038        incdec 1, 1, 0       # Else, we have a match! num_hit++, nop place
       SKIP_IF:
                      !-- Increment i, ptr
line039        otype  rframe, 0     # $F = 0
line040        incdec 0, 0, 0 # $reg4(i)i++
line041        incdec 0, 3, 0 # $reg0(ptr)++

                      !-- Reset i as the left comparison value
line042        jumpif 0, 3          # Reloop
line043        seed   1, 0, 0# $CL = $reg4, nop place
       END_WHILE:
line044        otype rframe, 1
line045        focus 1, 2
line046        lower 7         # Store $reg5(hits) into M[7]
line047        store 2, 1           # M[7] <= $reg5(hits)
line048        halt                 # EOP, exit
```

**What is the dynamic instruction count of this program. If it varies according to the values, assume 10 of the entries have the value 00011001, 20 entries have the value 10111101, 9 entries have the value 11100111, and the rest have 00000000. Assume the string is 0011.**

34 Non-LoopInstructions, 14 Loop Instructions if successful case, 11 Loop Instructions if successful case, ; 19 Match Cases, 44 No-Match Cases;  **(**64 - 3) Total Loop Iterations

<u>Dynamic Instruction Count</u>: 784 = 34 + 19*14 + 44*11

<u>Static Instruction Coun</u>t: 48

# 19. PROGRAM: CLOSEST PAIR

Register Assignment:

```
-------------------------------------------------------------------
frame  0             0             0             0
reg    0             1             2             3
-------------------------------------------------------------------
var    size, i A           B, diff min
-------------------------------------------------------------------
```

Psuedo Code:

```
       i = 128
       $CL = &i
       $CR = 147
START
       otype nset 1
       goto END if i !< $CR
       nop
       load $regA <= mem[i]
       i++
       load $regB <= mem[i]
       otype nset 0
       goto SWAP if $regA > $regB
       nop
       goto START
       nop
END
       i = 128
       min = 255
START2
       load $regA <= mem[i]
       i++
       load $regB <= mem[i]
       $diff <= $regB - $regA
       otype nset 0
       goto SETMIN if $diff < min
       nop
       goto START2
       nop
END2
       return min
       halt
SWAP
       store mem[i] <= $regA
       i--
       store mem[i] <= $regB
       i++
       goto START
       nop
SETMIN
       $diff = $regB - $regA
       min = $diff
       goto START2
       nop
```

FMAT Assembly:

```
       INIT:
line001       otype  rframe, 0      # REG FRAME 0
line002       otype  clear, 0       # $reg0 = 0 (i)
line003       move   1, 0           # $reg1 = 0 (A)
line004       move   2, 0           # $reg2 = 0 (B)
line005       move   3, 0           # $reg3 = 0 (min)
```

```
                    !-- Seed $CR
line006       focus  0, 0          # size = 147
line007       lower  3
line008       upper  9
line009       seed   0, 0, 1# $CR = 147

                    !-- Seed $CL, i = 128
line010       focus  0, 0          # i = 128
line011       lower  0
line012       upper  8
line013       seed   0, 0, 0# $CL = &i

                    !-- Seed $J0(START), $J1(END), $J2(SWAP)
line014       focus  3, 0          # $J0(START) line023
line015       lower  7
line016       upper  1
line017       focus  3, 0          # $J1(END) line034
line018       lower  2
line019       upper  2
line020       focus  3, 0          # $J2(SWAP) line065
line021       lower  1
line022       upper  4

START($J0):
line023       otype  nset, 1
line024       jumpif 1, 2          # goto END if $CL !< $CR
line025       otype  nop
line026       load   1, 0          # $reg1(A) <= mem[i]
line027       incdec 0, 0, 0# i++
line028       load   2, 0          # $reg2(B) <= mem[i]
line029       otype  nset, 0
line030       jumpif 2, 1          # goto SWAP if $CL > $CR
line031       otype  nop
line032       jumpif 0, 3          # goto START
line033       otype  nop
END($J1):

line034       focus  0, 0          # i = 128
line035       lower  0
line036       upper  8
line037       focus  0, 3          # min = 255
line038       lower  15
line039       upper  15

                    !-- Seed $J0(START2), $J1(END2), $J2(SETMIN)
line040       focus  3, 0          # $J0(START2) line049
line041       lower  1
line042       upper  3
line043       focus  3, 0          # $J1(END2) line060
line044       lower  12
line045       upper  3
line046       focus  3, 0          # $J2(SETMIN) line071
line047       lower  7
line048       upper  4

START2($J0):
line049       load   1, 0          # load $regA <= mem[i]
line050       indec  0, 0, 0# i++
line051       load   2, 0          # load $regB <= mem[i]
line052       sub    2, 1          # $diff <= $regB - $regA, $reg2 = diff
line053       otype  nset, 0
line054       seed   0, 2, 0# $CL = diff
```

```
line055        seed   0, 3, 1# $CR = min
line056        jumpif 2, 1          # goto $J2(SETMIN) if $diff < min
line057        otype  nop
line058        jumpif 0, 3          # goto $J0(START2)
line059        otype  nop

END2($J1):
line060        focus  0, 0          # i = 127
line061        lower  15
line062        upper  7
line063        store  0, 3          # M[127] <= min
line064        otype  halt          # Exit

SWAP($J2):
line065        store  0, 1          # store mem[i] <= $regA
line066        incdec 0, 2, 1# i--
line067        store  0, 2          # store mem[i] <= $regB
line068        incdec 0, 0, 0# i++
line069        jumpif 0, 3          # goto START
line070        otype nop


SETMIN:
line071        sub    2, 1          # $diff = $regB - $regA
line072        move   3, 2          # min <= $diff
line073        jumpif 0, 3          # goto START2
line074        otype  nop
```

**What is the dynamic instruction count of this program? Assume the array contains the sequence: 34, -12, 18, 61, 0, 100, 49, -51, -22, 3, 41, 88, -100,14, 39, 10, 90, -90, -80, 75. (The correct answer should be 2)**

Static Instruction Count: **74**

Dynamic Instruction Count: **1006**

static22 loop11 static15 loop11 static5, swap6, setmin4

**22 + (11*19 + 2) + 15 + (11*19 + 2) + 5 + 6*(89swaps) + 4*(2setmins) = 1006**

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
 -12   18   61    0  100   49  -51  -22    3   41   88 -100   14   39   10   90  -90  -80   75
 -12   18    0   61  100   49  -51  -22    3   41   88 -100   14   39   10   90  -90  -80   75
 -12   18    0   61   49  100  -51  -22    3   41   88 -100   14   39   10   90  -90  -80   75
 -12   18    0   61   49  -51  100  -22    3   41   88 -100   14   39   10   90  -90  -80   75
 -12   18    0   61   49  -51  -22  100    3   41   88 -100   14   39   10   90  -90  -80   75
 -12   18    0   61   49  -51  -22    3  100   41   88 -100   14   39   10   90  -90  -80   75
 -12   18    0   61   49  -51  -22    3   41  100   88 -100   14   39   10   90  -90  -80   75
 -12   18    0   61   49  -51  -22    3   41   88  100 -100   14   39   10   90  -90  -80   75
 -12   18    0   61   49  -51  -22    3   41   88 -100  100   14   39   10   90  -90  -80   75
 -12   18    0   61   49  -51  -22    3   41   88 -100   14  100   39   10   90  -90  -80   75
 -12   18    0   61   49  -51  -22    3   41   88 -100   14   39  100   10   90  -90  -80   75
 -12   18    0   61   49  -51  -22    3   41   88 -100   14   39   10  100   90  -90  -80   75
 -12   18    0   61   49  -51  -22    3   41   88 -100   14   39   10   90  100  -90  -80   75
 -12   18    0   61   49  -51  -22    3   41   88 -100   14   39   10   90  -90  100  -80   75
 -12   18    0   61   49  -51  -22    3   41   88 -100   14   39   10   90  -90  -80  100   75
 -12   18    0   61   49  -51  -22    3   41   88 -100   14   39   10   90  -90  -80   75  100
 -12   18    0   49   61  -51  -22    3   41   88 -100   14   39   10   90  -90  -80   75  100
 -12   18    0   49  -51   61  -22    3   41   88 -100   14   39   10   90  -90  -80   75  100
 -12   18    0   49  -51  -22   61    3   41   88 -100   14   39   10   90  -90  -80   75  100
 -12   18    0   49  -51  -22    3   61   41   88 -100   14   39   10   90  -90  -80   75  100
 -12   18    0   49  -51  -22    3   41   61   88 -100   14   39   10   90  -90  -80   75  100
 -12   18    0   49  -51  -22    3   41   61 -100   88   14   39   10   90  -90  -80   75  100
 -12   18    0   49  -51  -22    3   41   61 -100   14   88   39   10   90  -90  -80   75  100
 -12   18    0   49  -51  -22    3   41   61 -100   14   39   88   10   90  -90  -80   75  100
 -12   18    0   49  -51  -22    3   41   61 -100   14   39   10   88   90  -90  -80   75  100
 -12   18    0   49  -51  -22    3   41   61 -100   14   39   10   88  -90   90  -80   75  100
 -12   18    0   49  -51  -22    3   41   61 -100   14   39   10   88  -90  -80   90   75  100
 -12   18    0   49  -51  -22    3   41   61 -100   14   39   10   88  -90  -80   75   90  100
 -12   18    0  -51   49  -22    3   41   61 -100   14   39   10   88  -90  -80   75   90  100
 -12   18    0  -51  -22   49    3   41   61 -100   14   39   10   88  -90  -80   75   90  100
 -12   18    0  -51  -22    3   49   41   61 -100   14   39   10   88  -90  -80   75   90  100
 -12   18    0  -51  -22    3   41   49   61 -100   14   39   10   88  -90  -80   75   90  100
 -12   18    0  -51  -22    3   41   49 -100   61   14   39   10   88  -90  -80   75   90  100
 -12   18    0  -51  -22    3   41   49 -100   14   61   39   10   88  -90  -80   75   90  100
 -12   18    0  -51  -22    3   41   49 -100   14   39   61   10   88  -90  -80   75   90  100
 -12   18    0  -51  -22    3   41   49 -100   14   39   10   61   88  -90  -80   75   90  100
```

```
  -12   18    0  -51  -22    3   41   49 -100   14   39   10   61  -90   88  -80   75   90  100
  -12   18    0  -51  -22    3   41   49 -100   14   39   10   61  -90  -80   88   75   90  100
  -12   18    0  -51  -22    3   41   49 -100   14   39   10   61  -90  -80   75   88   90  100
  -12   18  -51    0  -22    3   41   49 -100   14   39   10   61  -90  -80   75   88   90  100
  -12   18  -51  -22    0    3   41   49 -100   14   39   10   61  -90  -80   75   88   90  100
  -12   18  -51  -22    0    3   41 -100   49   14   39   10   61  -90  -80   75   88   90  100
  -12   18  -51  -22    0    3   41 -100   14   49   39   10   61  -90  -80   75   88   90  100
  -12   18  -51  -22    0    3   41 -100   14   39   49   10   61  -90  -80   75   88   90  100
  -12   18  -51  -22    0    3   41 -100   14   39   10   49   61  -90  -80   75   88   90  100
  -12   18  -51  -22    0    3   41 -100   14   39   10   49  -90   61  -80   75   88   90  100
  -12   18  -51  -22    0    3   41 -100   14   39   10   49  -90  -80   61   75   88   90  100
  -12  -51   18  -22    0    3   41 -100   14   39   10   49  -90  -80   61   75   88   90  100
  -12  -51  -22   18    0    3   41 -100   14   39   10   49  -90  -80   61   75   88   90  100
  -12  -51  -22    0   18    3   41 -100   14   39   10   49  -90  -80   61   75   88   90  100
  -12  -51  -22    0    3   18   41 -100   14   39   10   49  -90  -80   61   75   88   90  100
  -12  -51  -22    0    3   18 -100   41   14   39   10   49  -90  -80   61   75   88   90  100
  -12  -51  -22    0    3   18 -100   14   41   39   10   49  -90  -80   61   75   88   90  100
  -12  -51  -22    0    3   18 -100   14   39   41   10   49  -90  -80   61   75   88   90  100
  -12  -51  -22    0    3   18 -100   14   39   10   41   49  -90  -80   61   75   88   90  100
  -12  -51  -22    0    3   18 -100   14   39   10   41  -90   49  -80   61   75   88   90  100
  -12  -51  -22    0    3   18 -100   14   39   10   41  -90  -80   49   61   75   88   90  100
  -51  -12  -22    0    3   18 -100   14   39   10   41  -90  -80   49   61   75   88   90  100
  -51  -22  -12    0    3   18 -100   14   39   10   41  -90  -80   49   61   75   88   90  100
  -51  -22  -12    0    3 -100   18   14   39   10   41  -90  -80   49   61   75   88   90  100
  -51  -22  -12    0    3 -100   14   18   39   10   41  -90  -80   49   61   75   88   90  100
  -51  -22  -12    0    3 -100   14   18   10   39   41  -90  -80   49   61   75   88   90  100
  -51  -22  -12    0    3 -100   14   18   10   39  -90   41  -80   49   61   75   88   90  100
  -51  -22  -12    0    3 -100   14   18   10   39  -90  -80   41   49   61   75   88   90  100
  -51  -22  -12    0 -100    3   14   18   10   39  -90  -80   41   49   61   75   88   90  100
  -51  -22  -12    0 -100    3   14   10   18   39  -90  -80   41   49   61   75   88   90  100
  -51  -22  -12    0 -100    3   14   10   18  -90   39  -80   41   49   61   75   88   90  100
  -51  -22  -12    0 -100    3   14   10   18  -90  -80   39   41   49   61   75   88   90  100
  -51  -22  -12 -100    0    3   14   10   18  -90  -80   39   41   49   61   75   88   90  100
  -51  -22  -12 -100    0    3   14   10  -90   18  -80   39   41   49   61   75   88   90  100
  -51  -22  -12 -100    0    3   14   10  -90  -80   18   39   41   49   61   75   88   90  100
  -51  -22 -100  -12    0    3   14   10  -90  -80   18   39   41   49   61   75   88   90  100
  -51  -22 -100  -12    0    3   14  -90   10  -80   18   39   41   49   61   75   88   90  100
  -51  -22 -100  -12    0    3   14  -90  -80   10   18   39   41   49   61   75   88   90  100
  -51 -100  -22  -12    0    3   14  -90  -80   10   18   39   41   49   61   75   88   90  100
  -51 -100  -22  -12    0    3  -90   14  -80   10   18   39   41   49   61   75   88   90  100
  -51 -100  -22  -12    0    3  -90  -80   14   10   18   39   41   49   61   75   88   90  100
  -51 -100  -22  -12    0    3  -90  -80   10   14   18   39   41   49   61   75   88   90  100
 -100  -51  -22  -12    0    3  -90  -80   10   14   18   39   41   49   61   75   88   90  100
 -100  -51  -22  -12    0  -90    3  -80   10   14   18   39   41   49   61   75   88   90  100
 -100  -51  -22  -12    0  -90  -80    3   10   14   18   39   41   49   61   75   88   90  100
 -100  -51  -22  -12  -90    0  -80    3   10   14   18   39   41   49   61   75   88   90  100
 -100  -51  -22  -12  -90  -80    0    3   10   14   18   39   41   49   61   75   88   90  100
 -100  -51  -22  -90  -12  -80    0    3   10   14   18   39   41   49   61   75   88   90  100
 -100  -51  -22  -90  -80  -12    0    3   10   14   18   39   41   49   61   75   88   90  100
 -100  -51  -90  -22  -80  -12    0    3   10   14   18   39   41   49   61   75   88   90  100
 -100  -51  -90  -80  -22  -12    0    3   10   14   18   39   41   49   61   75   88   90  100
 -100  -90  -51  -80  -22  -12    0    3   10   14   18   39   41   49   61   75   88   90  100
 -100  -90  -80  -51  -22  -12    0    3   10   14   18   39   41   49   61   75   88   90  100
```

```
mins:
-10, 2: total of 2 guesses, the second guess was correct
```