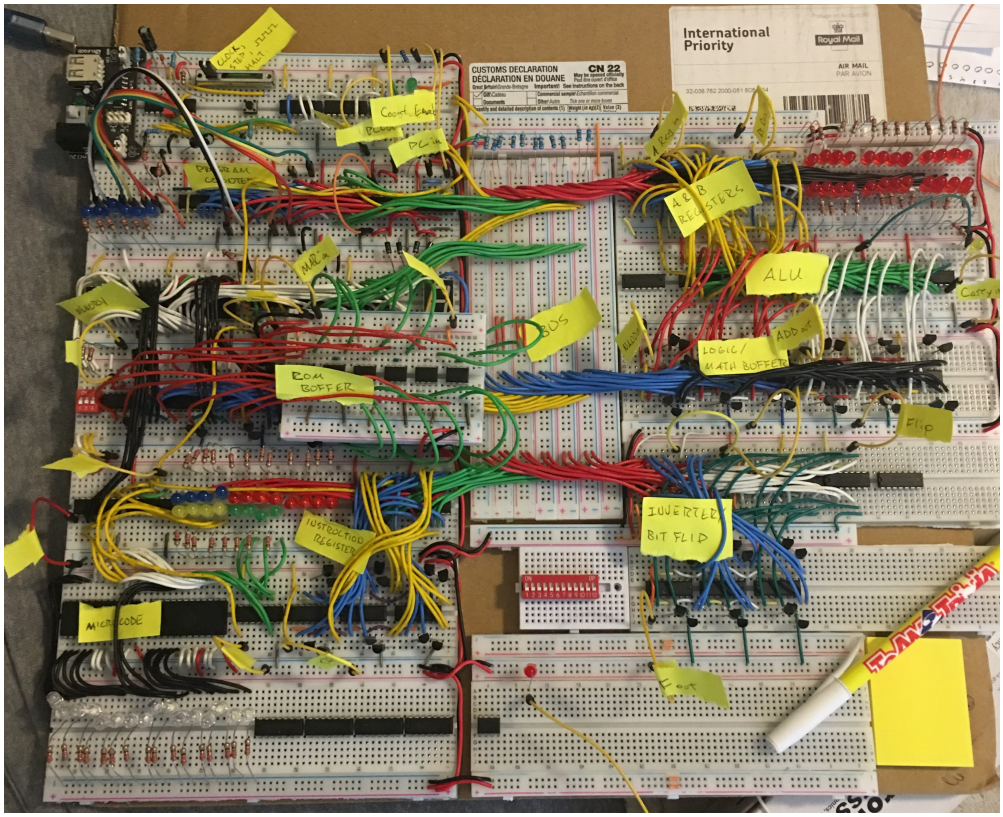
K.E.N.N.Y. v5.0 —— User Guide

"Because sometimes you gotta die to get better..."

G. Riggs — 2020



Contents

1 Who is K.E.N.N.Y.?

The "Kerneled Enumerable Natural Number Yammer" is a 12-bit breadboard computer design. In the absence of true RAM, memory is facilitated by 2x2KB EEPROMs, mapped in parallel to 64 program lines of 16-bit instructions, and 192 addresses (12 bits each); this leaves 3 bits free, partitioning the memory into 8 program blocks. The instructions are encoded, where 16 bits manipulate 21 control signals with the assistance of 3 multiplexers. Computation is provided by an arithmetic logic unit (ALU) capable of addition and NAND, operating on two general purpose 12-bit registers ("A" and "B"). A unique bit-flipper/rotator which allows subtraction and modulo-2 division (in addition to caching) is also implemented.

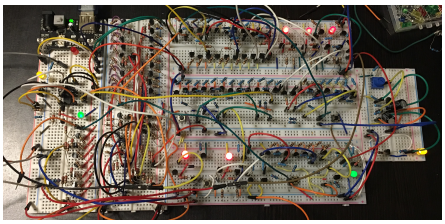
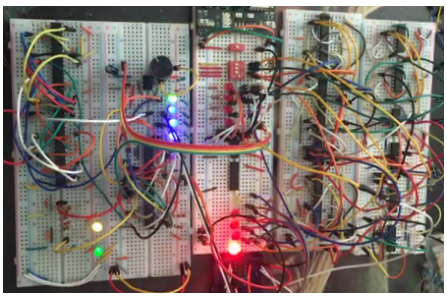
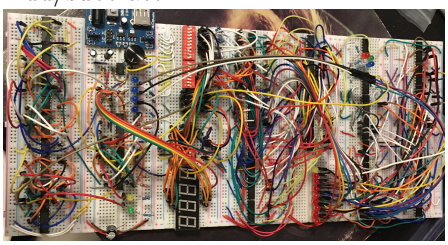
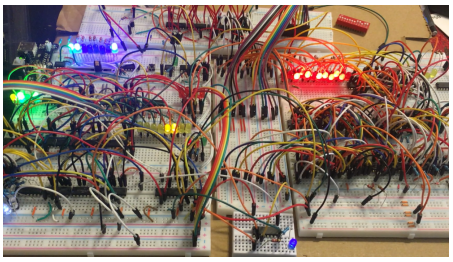
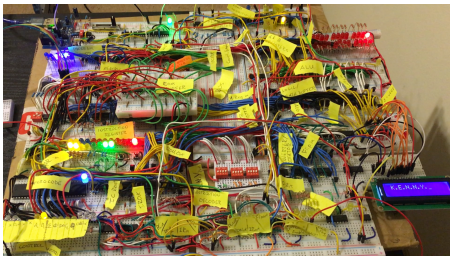
With no RAM in the classical sense, the memory map is significantly simplified, albeit at the expense of write time: programs may be written directly into the EEPROM with microcode installed on the machine in the exact way arithmetic values are stored. At present, error-free reads from EEPROM can be completed much faster than writes ($\sim$2 orders of magnitude). This is a key design restriction; future versions promise a true RAM to handle fast writes to memory, assuring prompt handling of intermediate values. On the upside, one benefit of this "slow" approach is non-volatility – all data is retained by the machine after shut-down.

As K.E.N.N.Y. has no stack paradigm, subroutines must be embedded into the program with specialized jump conditions. This is generally not a problem (as the machine remains Turing-complete), but it does pose certain challenges for purely calculative endeavors which benefit greatly from recursion. [For context, see Section 7, which outlines an obtuse implementation of the factorial function.] Nevertheless, a wide selection of machine code instructions allows many creative avenues of code design for the dedicated breadboard computer scientist.

The microcode comprising the kernel of K.E.N.N.Y.'s control unit is easily adjustable with nothing but tweezers and patience. This allows users to design application-specific instructions to suit their individual needs or desires. In the event of a particularly involved computation, it is relatively painless (though not straightforward) to implement a finite-state machine. Additionally, a 16x2 LCD display and two-button controller provides the hardware to run a few simple video games [see 8], assuming the coder refrains from writing to EEPROM (e.g., by using the flipper/rotator as a cache, or reading from the LCD). Development of these routines has proved both a challenging and rewarding experience for the author.

In essence, K.E.N.N.Y. is a flashing tangle of prototype board, wire, and ICs which costs less than a hamburger and three craft beers at your favorite hipster restaurant. He runs at least a million times slower than the computer used to typeset this document, and has less memory than an average microwave. And though he does not reheat burritos, he will happily replace his heart with a baked potato, if only you translate the ascii to his machine code.

2 Genealogy

K.E.N.N.Y. v1.0	2-bit parallel, finite-state No ICs	 Add
K.E.N.N.Y. v2.0	4-bit serial, finite-state 555, NANDs, ANDs, Shift register	 Add/subtract
K.E.N.N.Y. v3.0	8-bit serial, finite-state 555, NANDs, ANDs, ORs, XORs, Shift register	 Add/subtract/Fibonacci/Collatz
K.E.N.N.Y. v4.0	12-bit parallel, finite-state 555s, NANDs, ANDs, ORs, XORs, Adders, Counters, Inverters	 Add/subtract/Fibonacci/Collatz
K.E.N.N.Y. v5.0	12-bit parallel, Turing-complete 555s, NANDs, ANDs, ORs, XORs, EEPROMs, Adders, Counters, Inverters, Op Amps	 Programmable

3 Block Diagram

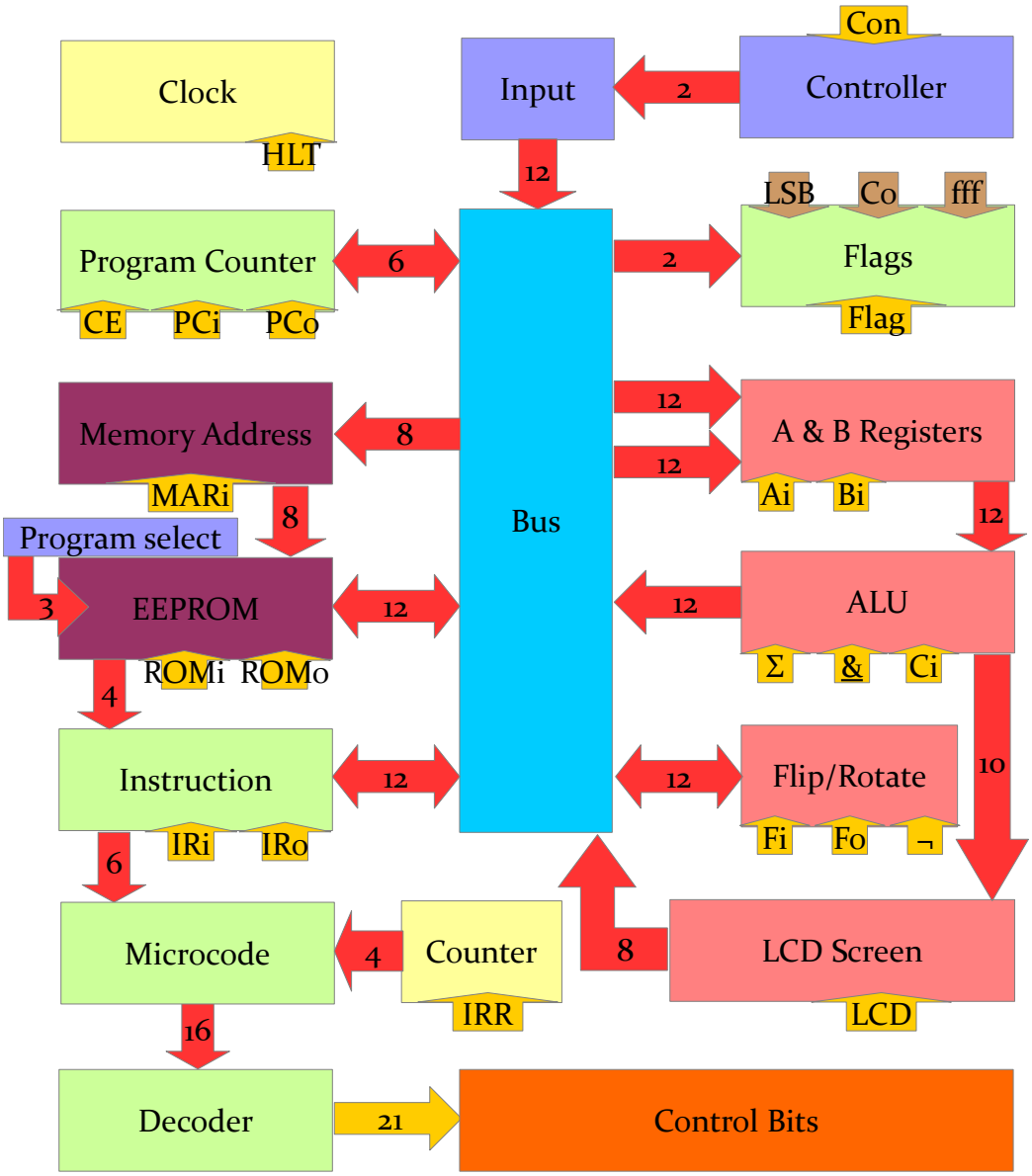


Figure 1: Schematic of K.E.N.N.Y. computer architecture. Connections in red are labeled by how many bits they transfer, and which direction these data are moved. Orange arrows represent respective control signals. The three brown arrows into the flags register show the conditions under which a program may branch.

4 Control Scheme

Fundamentally, a computer might be imagined as a "one-button" machine, where the instructions are set before each clock pulse. Upon receipt of a pulse, the machine executes the chosen instructions, increments some form of counter, and the process repeats *ad nauseum*. In K.E.N.N.Y.'s case, the control word is given by

$$\text{In x x Out x x Ai Bi } \Sigma \text{ } \overline{\&} \text{ Inv Ci Fi Clk x x } .$$

By multiplexing, we can manipulate more than 16 control bits. For clarity, we provide the following tables:

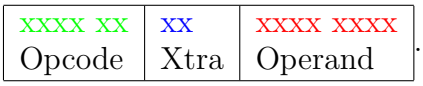
Selection	xx	Result	Selection	xx	Result
In = 1	00	PC in	Clk = 1	00	IR Reset
	01	ROM in		01	Count enable
	10	IR in		10	Halt
	11	MAR in		11	Latch flags
Out = 1	00	PC out	Clk = 0	01	LCD enable
	01	ROM out		10	Controller
	10	IR out			
	11	Flip out			

Control bit	Action on clock pulse	Microcode
PC in	Latch bus to program counter*	`pci`
IR in	Latch bus to instruction register	`iri`
ROM in	Latch bus to EEPROM data	`romi`
MAR in	Latch bus to EEPROM address	`mari`
Ai	Latch bus to register A	`ai`
Bi	Latch bus to register B	`bi`
Fi	Latch bus to flipper/rotator	`fi`
PC out	Send program counter value to bus	`pco`
IR out	Send instruction register value to bus	`iro`
ROM out	Send EEPROM value to bus	`romo`
Flip out	Send flipper/rotator value to bus	`fo`
Σ	Send adder value to bus	`sum`
$\overline{\&}$	Send NAND value to bus	`nand`
Inv	Invert bits in flipper/rotator	`inv`
Ci	Set adder's carry input to 1	`ci`
IR Reset	Reset instruction register's counter	`irr`
Count enable	Increment program counter	`ce`
Halt	Stop clock from running	`halt`
Latch flags	Latch flags register	`flag`
LCD enable	Send adder value to display	`lcd`
Controller	Enable input from controller	`con`

*Assuming the jump condition is satisfied.

5 Instruction Set

All machine code instructions to K.E.N.N.Y. are two bytes, and may be interpreted as



The "xtra" bits are for encoding jump conditions, adding LCD functionality, enabling game controllers, and anything else a programmer can imagine! Thus, if we want K.E.N.N.Y. to add the contents of registers A and B, then store the result to memory address 0x80, we would use the instruction "0001 0000 1000 0000" or equivalently, "0x1080". The following list elucidates which control signals are brought high by the microcode EEPROMs for each value of the counter. Unless otherwise noted, all instructions involve a fetch cycle on the first two clock ticks:

```
0000 pco mari                          11110000 00000000
0001 romo iri ce                       11010100 00000101
```

hence we will ignore these redundant cycles in the spirit of brevity.

5.1 0000 xx BOOT

```
Intializes the system.  Clears registers, program counter, and memory address.


0000                                   00000000 00000000
.                                      .
.                                      .
.                                      .
1100 flag                              00000000 00000111
1101 mari                              11100000 00000000
1110 pci ai bi fi                      10000011 00001000
1111 romo iri                          11010100 00000000
```

5.2 0001 00 ADD

```
Adds registers A and B, saves result to operand's memory address.
Latches flags register.

0010 iro mari                          11111000 00000000
0011 sum romi flag                     10100000 10000111
0100 irr                               00000000 00000100
```

5.3 0010 00 AWC

Adds registers A and B, adds one, saves result to operand's memory address.
Latches flags register!

```
0010 iro mari                    11111000 00000000
0011 sum romi ci flag            10100000 10010111
0100 irr                         00000000 00000100
```

5.4 0011 00 NAN

NANDs registers A and B, saves result to operand's memory address.
Latches flags register!

```
0010 iro mari                    11111000 00000000
0011 nand romi flag              10100000 01000111
0100 irr                         00000000 00000100
```

5.5 0100 00 FLP

Flip bits of value at operand's memory address.
Latches flags register!

```
0010 iro mari                    11111000 00000000
0011 romo fi                     10100000 00001000
0100 fo inv fi                   00011100 00101000
0101 fo romi flag                10111100 00000111
0110 irr                         00000000 00000100
```

5.6 0101 00 ROT

Rotate bits of value at operand's memory address.
Latches flags register!

```
0010 iro mari                    11111000 00000000
0011 romo fi                     00010100 00001000
0100 fo romi flag                10111100 00100111
0101 irr                         00000000 00000100
```

5.7 0110 00 INC

Increment value at operand's memory address.

```
0010 iro mari                    11111000 00000000
0011 romo ai                     00010110 00000000
0100 bi                          00000001 00000000
0101 sum romi ci                 10100000 10010000
0110 irr                         00000000 00000100
```

5.8 0111 00 [xx] JMP

```
Loads program counter with operand's value if jump condition is satisfied.
00:  unconditional
01:  LSB odd
10:  carry out
11:  ALU = 0xfff
```

```
0010 iro pci                        10011000 00000000
0011 irr                            00000000 00000100
```

5.9 1000 00 LDA

```
Load value at operand's memory address into register A.
```

```
0010 iro mari                       11111000 00000000
0011 romo ai                        00010110 00000000
0100 irr                            00000000 00000100
```

5.10 1001 00 LDB

```
Load value at operand's memory address into register B.
```

```
0010 iro mari                       11111000 00000000
0011 romo bi                        00010101 00000000
0100 irr                            00000000 00000100
```

5.11 1010 00 MXX

```
Multiply value at operand's memory address by 2 (shift left).
Latches flags register!
```

```
0010 iro mari                       11111000 00000000
0011 romo ai bi                     00010111 00000000
0100 sum romi flag                  10100000 10000111
0101 irr                            00000000 00000100
```

5.12 1011 00 DXX

```
Divide value at operand's memory address by 2 (shift right).
Latches flags register!
```

```
0010 iro mari                       11111000 00000000
0011 romo fi                        00010100 00001000
0100 fo ai bi                       00011111 00000000
0101 sum fi                         00000000 10001000
0110 fi romi flag                   10100000 00001111
0111 irr                            00000000 00000100
```

5.13 1100 00 [xx] LCD

```
Perform operation on LCD display; data (if read) sent to B register.
00:  instruction
01:  read address
10:  character
11:  read data @ address
```

```
0010 iro ai                          00011010 00000000
0011 bi                              00000001 00000000
0100 lcd bi                          00000001 00000001
0101 irr                             00000000 00000100
```

5.14 1101 00 INP

```
Load bus value into operand's memory address.  (Finicky!)
```

```
0010 iro mari                        11111000 00000000
0011 romi halt                       10100000 00000110
0100 halt                            00000000 00000110
0101 irr                             00000000 00000100
```

5.15 1110 00 WRI

```
User chooses memory location then writes data there.  (How to program.)
```

```
0000 mari ai halt                    11100010 00000110
0001 romi bi halt                    10100001 00000110
0010 irr                             00000000 00000100
```

5.16 1111 00 NOP

```
No operation; moves to next instruction.
```

```
0010 irr                             00000000 00000100
```

5.17 0001 01 SVP

```
Saves operand as a pointer at memory location 0xff.
```

```
0010 fi                              00000000 00001000
0011 fo inv mari                     11111100 00100000
0100 iro romi                        10111000 00000000
0101 irr                             00000000 00000100
```

5.18 0010 01 SWP

Swaps value at operand's memory location with pointer's value.

```
0010 ai bi fi                        00000011 00001000
0011 fo inv mari                     11111100 00100000
0100 romo mari                       11110100 00000000
0101 romo fi                         00010100 00001000
0110 iro mari                        11111000 00000000
0111 romo ai                         00010110 00000000
1000 fo fi                           00011100 00001000
1001 fo romi                         10111100 00000000
1010 fi                              00000000 00001000
1011 fo inv mari                     11111100 00100000
1100 romo mari                       11110100 00000000
1101 sum romi                        10100000 10000000
1110 irr                             00000000 00000100
```

5.19 0011 01 FAN

ANDs value at operand's memory location with flip register.

```
0010 iro mari                        11111000 00000000
0011 romo ai                         00010110 00000000
0100 fo bi                           00011101 00000000
0101 nand fi                         00000000 01001000
0110 fo inv fi                       00011100 00101000
0111 irr                             00000000 00000100
```

5.20 0100 01 FAD

Adds value at operand's memory location to flip register.
Latches flags register!

```
0010 iro mari                        11111000 00000000
0011 romo bi                         00010101 00000000
0100 fo ai                           00011110 00000000
0101 sum fi flag                     00000000 10001111
0110 fo fi                           00011100 00001000
0111 irr                             00000000 00000100
```

5.21 0101 01 FSU

Subtracts value of operand's memory location from flip register.
Latches flags register!

```
0010 iro mari                          11111000 00000000
0011 romo ai bi                        00010111 00000000
0100 nand bi                           00000001 01000000
0101 fo ai                             00011110 00000000
0110 sum ci fi flag                    00000000 10011111
0111 fo fi                             00011100 00001000
1000 irr                               00000000 00000100
```

5.22 0110 01 CWF

Compares flip register with value at operand's memory address.
n/a: op* < F
car: op* > F
fff: op* == F

```
0010 iro mari                          11111000 00000000
0011 romo ai                           00010110 00000000
0100 fo inv bi                         00011101 00100000
0101 flag                              00000000 00000111
0110 irr                               00000000 00000100
```

5.23 0111 01 [xx] JIN

Jumps if controller button depressed. Use for games!
01: right button
10: left button

```
0010 ai bi                             00000011 00000000
0011 nand ai                           00000010 01000000
0100 flag                              00000000 00000111
0101 romo iri con                      11010100 00000010
0110 iro pci                           10011000 00000000
0111 irr                               00000000 00000100
```

5.24 1000 01 LDF

Load value at operand's memory address into flip register.

```
0010 iro mari                          11111000 00000000
0011 romo fi                           00010100 00001000
0100 fo fi                             00011100 00001000
0101 irr                               00000000 00000100
```

5.25 1001 01 SAF

Saves value of flip register at operand's memory address.

```
0010 iro mari                          11111000 00000000
0011 fo romi                           10111100 00001000
0100 irr                               00000000 00000100
```

5.26 1010 01 FMX

```
Multiplies flip register value by 2 (shift left).
Latches flags register!
```

```
0010 fo ai bi                          00011111 00000000
0011 sum fi flag                       00000000 10001111
0100 fo fi                             00011100 00001000
0101 irr                               00000000 00000100
```

5.27 1011 01 FDX

```
Divides flip register value by 2 (shift right).
Latches flags register!
```

```
0010 fo fi                             00011100 00001000
0011 fo ai bi                          00011111 00000000
0100 sum fi flag                       00000000 10001111
0101 irr                               00000000 00000100
```

5.28 1100 01 LCF

```
Performs LCD operation in flip register.
```

```
0010 fo ai                             00011110 00000000
0011 bi                                00000001 00000000
0100 lcd bi                            00000001 00000001
0101 irr                               00000000 00000100
```

5.29 1101 01 INF

```
Load bus value into flip register.
```

```
0010 fi halt                           00000000 00001110
0011 halt                              00000000 00000110
0100 fo fi                             00011100 00001000
0101 irr                               00000000 00000100
```

5.30 1111 01 CLR

```
Clears all registers.
```

```
0010 ai bi fi                          00000011 00001000
0011 irr                               00000000 00000100
```

5.31 0001 10 PXA

Adds value at operand's memory location to pointer's value (@0xfff).

```
0010 iro mari                        11111000 00000000
0011 romo bi                         00010101 00000000
0100 fi                              00000000 00001000
0101 fo inv mari                     11111100 00100000
0110 romo mari                       11110100 00000000
0111 romo ai                         00010110 00000000
1000 sum romi                        10100000 10000000
1001 irr                             00000000 00000100
```

5.32 0010 10 PXS

Subtracts value at operand's memory location from pointer's value.

```
0010 iro mari                        11111000 00000000
0011 romo ai bi                      00010111 00000000
0100 nand bi                         00000001 01000000
0101 fi                              00000000 00001000
0110 fo inv mari                     11111100 00100000
0111 romo mari                       11110100 00000000
1000 romo ai                         00010110 00000000
1001 sum ci romi                     10100000 10010000
1010 irr                             00000000 00000100
```

5.33 0011 10 FOR

ORs value at operand's memory address with flip register.

```
0010 iro mari                        11111000 00000000
0011 romo ai bi                      00010111 00000000
0100 nand ai                         00000010 01000000
0101 fo inv bi                       00011101 00100000
0110 nand fi                         00000000 01001000
0111 fo fi                           00011100 00001000
1000 irr                             00000000 00000100
```

5.34 0100 10 FIN

Increments value in flip register.

```
0010 fo ai                           00011110 00000000
0011 bi                              00000001 00000000
0100 sum ci fi                       00000000 10011000
0101 fo fi                           00011100 00001000
0110 irr                             00000000 00000100
```

5.35 0101 10 FDE

Decrements value in flip register.

```
0010 ai bi                          00000011 00000000
0011 nand bi                        00000001 01000000
0100 fo ai                          00011110 00000000
0101 sum fi                         00000000 10001000
0110 fo fi                          00011100 00001000
0111 irr                            00000000 00000100
```

5.36 0110 10 CWB

Compares B register with value at operand's memory address.

```
0010 ai                             00000010 00000000
0011 sum ai                         00000010 10000000
0100 nand bi                        00000001 01000000
0101 iro mari                       11111000 00000000
0110 romo ai                        00010110 00000000
0111 flag                           00000000 00000111
1000 ai                             00000010 00000000
1001 sum ai                         00000010 10000000
1010 nand bi                        00000001 01000000
1011 irr                            00000000 00000100
```

5.37 0111 10 [01] JFO

Jumps if flip register LSB == 1.

```
0010 fo flag                        00011100 00000111
0011 iro pci                        10011000 00000000
0100 irr                            00000000 00000100
```

5.38 1000 10 AFF

Adds A and F registers, places value in F register.

```
0010 fo bi                          00011101 00000000
0011 sum fi                         00000000 10001000
0100 fo fi                          00011100 00001000
0101 irr                            00000000 00000100
```

5.39 1001 10 BFF

Adds B and F registers, places value in F register.

```
0010 fo ai                          00011110 00000000
0011 sum fi                         00000000 10001000
0100 fo fi                          00011100 00001000
0101 irr                            00000000 00000100
```

5.40 1010 10 FMT

```
Multiplies flip register value by 3.
Latches flags register!
```

```
0010 fo ai bi                       00011111 00000000
0011 sum ai                         00000010 10000000
0100 fo bi                          00011101 00000000
0101 sum fi flag                    00000000 10001111
0110 fo fi                          00011100 00001000
0111 irr                            00000000 00000100
```

5.41 1011 10 FMV

```
Multiplies flip register value by 5.
Latches flags register!
```

```
0010 fo ai bi                       00011111 00000000
0011 sum ai bi                      00000011 10000000
0100 sum ai                         00000010 10000000
0101 fo bi                          00011101 00000000
0110 sum fi flag                    00000000 10001111
0111 fo fi                          00011100 00001000
1000 irr                            00000000 00000100
```

5.42 1100 10 LFP

```
Uses flip register as pointer to address for LCD char/op.
```

```
0010 fo mari                        11111100 00000000
0011 romo ai                        00010110 00000000
0100 bi                             00000001 00000000
0101 lcd bi                         00000001 00000001
0110 irr                            00000000 00000100
```

5.43 0001 11 PPA

```
Adds value at operand's memory location to pointer.
```

```
0010 iro mari                       11111000 00000000
0011 romo bi                        00010101 00000000
0100 fi                             00000000 00001000
0101 fo inv mari                    11111100 00100000
0110 romo ai                        00010110 00000000
0111 sum romi                       10100000 10000000
1000 irr                            00000000 00000100
```

5.44 0010 11 PPS

Subtracts value of operand's memory location from pointer.

```
0010 iro mari                        11111000 00000000
0011 romo ai bi                      00010111 00000000
0100 nand bi                         00000001 01000000
0101 fi                              00000000 00001000
0110 fo inv mari                     11111100 00100000
0111 romo ai                         00010110 00000000
1000 sum ci romi                     10100000 10010000
1001 irr                             00000000 00000100
```

5.45 0011 11 FXO

XORs value at operand's memory address with flip register.

```
0010 iro mari                        11111000 00000000
0011 romo ai                         00010110 00000000
0100 fo bi                           00011101 00000000
0101 nand bi                         00000001 01000000
0110 fo ai                           00011110 00000000
0111 nand fi                         00000000 01001000
1000 romo ai                         00010110 00000000
1001 nand ai                         00000010 01000000
1010 fo fi                           00011100 00001000
1011 fo bi                           00011101 00000000
1100 nand fi                         00000000 01001000
1101 fo fi                           00011100 00001000
1110 irr                             00000000 00000100
```

5.46 0100 11 AIF

```
Increments A register, adds to flip register.
Sets fff flag if A+1 == op*.
Use to sum like young Gauss!  (for squaring...)
```
THIS NEEDS FIXED!

```
0010 bi                                 00000001 00000000
0011 sum ci ai                          00000010 10010000
0100 fo bi                              00011101 00000000
0101 sum fi                             00000000 10001000
0110 fo fi                              00011100 00001000
0111 sum ai bi                          00000010 10000000
1000 nand ai                            00000010 01000000
1001 romo bi                            00010101 00000000
1010 bi flag                            00000001 00000111
1011 sum bi                             00000001 10000000
1100 nand ai                            00000010 01000000
1101 irr                                00000000 00000100
```

5.47 0101 11 NGF

Negates flip register (twos-complement).

```
0010 ai                                 00000010 00000000
0011 fo inv bi                          00011101 00100000
0100 sum ci fi                          00000000 10011000
0101 fo fi                              00011100 00001000
0110 irr                                00000000 00000100
```

5.48 0110 11 [11] JFM

Jumps PC to op if !(F&op*)+op* == fff.
Useful to check a single bit in flip register.

```
0010 iro mari                           11111000 00000000
0011 romo ai                            00010110 00000000
0100 fo bi                              00011101 00000000
0101 nand bi                            00000001 01000000
0110 romo ai                            00010110 00000000
0111 flag                               00000000 00000111
1000 iro pci                            10011000 00000000
1001 irr                                00000000 00000100
```

5.49 0111 11 [01] JFC

Jumps if flip register MSB == 1.

```
0010 fo fi                              00011100 00001000
0011 fo flag                            00011100 00000111
0100 fo fi                              00011100 00001000
0101 iro pci                            10011000 00000000
0110 irr                                00000000 00000100
```

5.50 1000 11 SRI

```
Sets value at operand's memory location to PC + 1.
Necessary for subroutine calls.

0010 iro mari                          11111000 00000000
0011 pco ai                            00010010 00000000
0100 bi                                00000001 00000000
0101 sum ci romi                       10100000 10010000
0110 irr                               00000000 00000100
```

5.51 1001 11 SRO

```
Loads value at operand's memory location to PC.
Necessary for subroutine calls.

0010 iro mari                          11111000 00000000
0011 romo pci                          10010100 00000000
0100 irr                               00000000 00000100
```

5.52 1010 11 FSS

```
Adds flip to A register (saves in IR), multiplies flip by 2.
Can be used to multiply nibbles...

0010 fo bi                             00011101 00000000
0011 sum romo iri                      11010100 10000000
0100 fo ai bi                          00011111 00000000
0101 sum fi                            00000000 10001000
0110 fo fi                             00011100 00001000
0111 fo inv ai                         00011110 00100000
1000 bi                                00000001 00000000
1001 romo ai bi flag                   00010111 00000111
1010 nand bi                           00000001 01000000
1011 iro ai                            00011010 00000000
1100 nand ai bi                        00000011 01000000
1101 nand ai                           00000010 01000000
1110 irr                               00000000 00000100
```

5.53 1011 11 FSX

```
Saves A register in IR, multiplies flip by 2.
Can be used to multiply nibbles...
```

```
0010 bi                          00000001 00000000
0011 sum romo iri                11010100 10000000
0100 fo ai bi                    00011111 00000000
0101 sum fi                      00000000 10001000
0110 fo fi                       00011100 00001000
0111 fo inv ai                   00011110 00100000
1000 bi                          00000001 00000000
1001 romo ai bi flag             00010111 00000111
1010 nand bi                     00000001 01000000
1011 iro ai                      00011010 00000000
1100 nand ai bi                  00000011 01000000
1101 nand ai                     00000010 01000000
1110 irr                         00000000 00000100
```

5.54 1111 11 HLT

Halts operation.

```
0010 halt                        00000000 00000110
```

6 Function Table

high\low	00	01	10	11
0000	BOOT	BOOT	BOOT	BOOT
0001	ADD*†	SVP$^{\dagger}_{f}$	PXA$^{\dagger}_{abf}$	PPA$^{\dagger}_{abf}$
0010	AWC*†	SWP$^{\dagger}_{abf}$	PXS$^{\dagger}_{abf}$	PPS$^{\dagger}_{abf}$
0011	NAN*†	FAN$_{ab}$	FOR$_{ab}$	FXO$_{ab}$
0100	FLP$^{*\dagger}_{f}$	FAD$^{*}_{ab}$	FIN$^{\ddagger}_{ab}$	AIF$^{*}_{b}$
0101	ROT$^{*\dagger}_{f}$	FSU$^{*}_{ab}$	FDE$^{\ddagger}_{ab}$	NGF‡
0110	INC$^{\dagger}_{ab}$	CWF$^{*}_{ab}$	CWB$^{*}_{a}$	JFM$^{*}_{ab}$
0111	JMP	JIN$^{*}_{ab}$	JFO*	JFC*
1000	LDA$_{a}$	LDF$_{f}$	AFF$^{\ddagger}_{b}$	SRI$^{\dagger}_{ab}$
1001	LDB$_{b}$	SAF†	BFF$^{\ddagger}_{a}$	SRO
1010	MXX$^{*\dagger}_{abf}$	FMX$^{*\ddagger}_{ab}$	FMT$^{*\ddagger}_{ab}$	FSS$^{*\ddagger}_{b}$
1011	DXX$^{*\dagger}_{abf}$	FDX$^{*\ddagger}_{ab}$	FMV$^{*\ddagger}_{ab}$	FSX$^{*\ddagger}_{b}$
1100	LCD$_{ab}$	LCF$^{\ddagger}_{ab}$	LFP$^{\ddagger}_{ab}$	LFM$_{ab}$
1101	INP†	INF$^{\ddagger}_{f}$		
1110	WRI$^{\dagger\circ}$	RDM$^{\circ}$	WRL$^{\dagger\circ}$	RDL$^{\circ}$
1111	NOP‡	CLR$^{\ddagger}_{abf}$		HLT‡

*Changes flags register

†Writes to EEPROM

‡Instruction has no operand

$^{\circ}$No fetch cycle; use with caution!

$_{x}$Destroys information in register x

7 Example Code

With the exception of 7.5 and 7.7, the following examples use the flip register a cache. It may be an illuminating exercise to reconstruct their action using only the first column of the function table (warning: 7.6 cannot be done!). It is important to note that, by convention, each of these programs uses a `NOP` operation for line 0 (memory address 0x00). For clarity, we provide the corresponding machine code on the right-hand side of the page.

7.1 Increment

Standard debugging program. Should repeat after 4096 iterations. `=^]`

```
-3 increment.k                                  .
-2 ============                                 .
-1 "Counts up, and up, and..."                  .
 0 ------------                                 .
 1 //A/FIN/x00 % F->F+1                         0x4800
 2     JMP[]/A % loop                           0x7001
```

7.2 Fibonacci

A classic. Can be used to generate a repeatable sequence of pseudo-random numbers.

```
-3 fibonacci.k                                  .
-2 ============                                 .
-1 "To a golden ratio..."                       .
 0 ------------                                 .
 1 FIN/x00 % F->F+1                             0x4800
 2 //A/AFF/x00 % A+F->F                         0x8800
 3     BFF/x00 % B+F->F                         0x9800
 4     JMP[]/A % loop                           0x7002
```

7.3 Bit shift

This looks interesting because division is accomplished by multiplying reversed numbers, leading to a "swooshing" effect.

```
-3 shift.k                                      .
-2 ============                                 .
-1 "Fun register visual"                        .
 0 ------------                                 .
 1 //B/FMX/x00 % F->F*2                         0xa400
 2     FIN/x00 % F->F+1                         0x4800
 3     JMP[car]/A % jump on carry               0x7205
 4     JMP[]/B % else loop                      0x7001
 5 //A/FDX/x00 % F->F/2                         0xb400
 6     JMP[car]/B % restart if done             0x7201
```

7.4 Collatz conjecture

Probably the most beautiful thing K.E.N.N.Y. can do, and the reason I took the time to build him. For the unfamiliar, given some $n_0 \in \mathbb{N}$,

$$n_{k+1} = \begin{cases} n_k/2, & n_k \text{ even} \\ 3n_k + 1, & n_k \text{ odd} \end{cases}$$

The conjecture is that all n_0 will eventually fall into the $1 \to 4 \to 2$ cycle. As this is a modulo-2^{12} enumeration, roughly 1/3 of all numbers will never halt.

```
-3 collatz.k                                          .
-2 ============                                        .
-1 "Please prove this!"                               .
 0 ------------                                       .
 1 //C/INF/x00 % input to flip                        0xd400
 2 //B/JFO[01]/A % jump if odd                        0x7a05
 3     FDX/x00 % F->value/2 if even                   0xb400
 4     JMP[]/B                                        0x7002
 5 //A/CWF/x81(=x001) % check if value=1              0x6481
 6     JMP[fff]/C % restart if so                     0x7301
 7     FMX/x00 % F->2*value                           0xa400
 8     AFF/x00 % F->3*value                           0x8800
 9     FIN/x00 % F->3*value+1                         0x4800
10     JMP[]/B                                        0x7002
```

7.5 Display name

Essentially a standard "Hello world!" routine. Nice for debugging LCDs, and display.

```
-3 kenny.k                                            .
-2 ============                                        .
-1 "I can spell!"                                     .
 0 ------------                                       .
 1 LCD/?x038 % initializes lcd                        0xc038
 2 LCD/?x00f % turn on cursor                         0xc00f
 3 //A/LCD/?x082 % puts cursor to (0,3)               0xc082
 4     LCD/"x4b" % K                                  0xc24b
 5     LCD/"x2e" % .                                  0xc22e
 6     LCD/"x45" % E                                  0xc245
 7     LCD/"x2e" % .                                  0xc22e
 8     LCD/"x4e" % N                                  0xc24e
 9     LCD/"x2e" % .                                  0xc22e
10     LCD/"x4e" % N                                  0xc24e
11     LCD/"x2e" % .                                  0xc22e
12     LCD/"x59" % Y                                  0xc259
13     LCD/"x2e" % .                                  0xc22e
14     LCD/?x001 % clear screen                       0xc001
15     JMP[]/A % loop                                 0x7003
```

7.6 Shooter

This was an incredibly fun time to write. Besides pushing the current version of K.E.N.N.Y. to his limits, it is just a generally good time (despite the lack of enemies, levels, score, or objective). Running 7.5 or something similar before playing allows you the joy of clearing the screen with your missile...

```
-9 shooter.k                                                    .
-8 ============                                                 .
-7 "Super basic shooter game"                                   .
-6 ------------                                                 .
-5 ?x010 -> Z + x80 % jump for shot value=16                    .
-4 ?x080 -> H + x80 % jump for shot on                          .
-3 ?msb = x40(=800) % msb toggle                                .
-2 ?top = x41(=082) % top shot                                  .
-1 ?bot = x42(=8c2) % bottom shot                               .
 0 ?onof = x44(=080) % shot on/off                              .
 1 LCD/?x038 % initializes lcd                                  0xc038
 2 LCD/?x00c % turn off cursor                                  0xc00c
 3 LCD/?x081 % puts cursor to (0,2)                             0xc081
 4 LCD/"x3e" % draws ship                                       0xc23e
 5 //D/JIN[01]/A % check up/down                               0x7511
 6 //G/JIN[10]/B % check fire                                  0x761c
 7     JFM[fff]/H + 0x80 % check shot on                        0x6f89
 8     JMP[]/D % do nothing                                     0x7005
 9 //H/LCF/x00 % place cursor                                   0xc100
10     LCD/"x20" % draw blank                                   0xc220
11     FIN/x00 % increment flip                                 0x4800
12     JFM[fff]/Z + 0x80 % exit if shot off screen              0x6f90
13     LCD/"xa5" % draw exhaust                                 0xc2a5
14     LCD/"x7e" % draw missile                                 0xc27e
15     JMP[]/D                                                  0x7005
16 //Z/FXO/?onof % turn shot off *forgot:17 jmp[]/D*           0x3c44
17     //A/FAD/?msb % toggle up/down                            0x4400
18         JMP[car]/E                                           0x7217
19         LCD/?x081 % up                                       0xc081
20         LCD/"x20" % draw blank                               0xc220
21         LCD/?x0c1 % down                                     0xc0c1
22         JMP[]/F                                              0x701a
23     //E/LCD/?x0c1 % down                                     0xc0c1
24         LCD/"x20" % draw blank                               0xc220
25         LCD/?x081 % up                                       0xc081
26     //F/LCD/"x3e" % draw ship                                0xc23e
27         JMP[]/G                                              0x7006
28     //B/JFM[fff]/H + 0x80 % check shot on                    0x6f89
29         JFC[01]/J % check up/down                            0x7d20
30         LDF/?top % if up, set pos.                           0x8441
31         JMP[]/H                                              0x7009
32     //J/LDF/?bot % same if down                              0x8442
33         JMP[]/H                                              0x7009
```

7.7 Factorial

Naïve implementation which demonstrates most of the standard (1^{st} column) functions.

```
-13 factorial.k                                                    .
-12 ============                                                   .
-11 "Computes n*(n-1)*...*2*1"                                     .
-10 ------------                                                   .
 -9 ?null = x40 (=000)                                            .
 -8 ?fff = x4f (=fff)                                             .
 -7 ?nopp = x44                                                   .
 -6 ?n = x80 (=value)                                             .
 -5 ?q = x81                                                      .
 -4 ?k = x8f                                                      .
 -3 ?a = x88                                                      .
 -2 ?b = x8c                                                      .
 -1 ?c = x8e                                                      .
  0 AWC/?c     % c->1                                             0x208e
  1 ADD/?k     % k->0                                             0x108f
  2 LDA/?n     % regA = n                                         0x8080
  3 ADD/?q     % q->n                                             0x1081
  4 FLP/?q     % q->!q (=!n)                                      0x4081
  5 //C/LDA/?k       % regA = k                                   0x808f
  6     LDB/?q       % regB = q                                   0x9081
  7     ADD/?nopp     % set flag                                  0x1044
  8     JMP[fff]/A   % jump if k=n                               0x7321
  9     INC/?k       % k->k+1                                    0x608f
 10     LDA/?null    % regA = 0                                  0x8040
 11     LDB/?k       % regB = k                                  0x908f
 12     ADD/?a       % a->k                                      0x1088
 13     LDB/?c       % regB = c                                  0x908e
 14     ADD/?b       % b->c                                      0x108c
 15     LDB/?null    % regB = 0                                  0x9040
 16     ADD/?c       % c->0                                      0x108e
 17     LDB/?b       % regB = b                                  0x908c
 18     ADD/?b       % b = b (need to set flag)                  0x108c
 19     //D/JMP[odd]/B  % jump if b odd                           0x721e
 20         LDA/?fff     % regA = fff                            0x804f
 21         LDB/?b       % regB = b                              0x908c
 22         ADD/?NOP     % set flag                              0x108e
 23         JMP[fff]/C   % jump if b=0                          0x7306
 24         //E/MXX/?a   % a->2*a                                0xa088
 25             DXX/?b   % b->b/2                                0xb08c
 26             LDA/?c   % regA = c                              0x808e
 27             JMP[]/D  % jump                                  0x7014
 28         //B/LDA/?c   % regA = c                              0x808e
 29             LDB/?a   % regB = a                              0x9088
 30             ADD/?c   % c->a+c                                0x108e
 31             JMP[]/E                                          0x7019
 32 //A/LDB/?null % clear regB                                   0x9040
 33 //F/LDA/?c % show answer                                     0x808e
 34     JMP[]/F % loop forever                                   0x7022
```

8 Game Development

By allowing *reads* from the LCD's internal RAM, we can easily implement collision detection, or other more advanced behavior. This section briefly outlines some rough ideas for entertaining/interactive programs. Most involve using the flip register as a pointer to memory locations beyond 0x3f, which allows custom characters, lengthy setup, &c. to occur without distressing the program's run-time memory.

8.1 Frog vs. Flies

After the proof of principle that is 7.6, we now use side scroll, enemies, and collision detection.

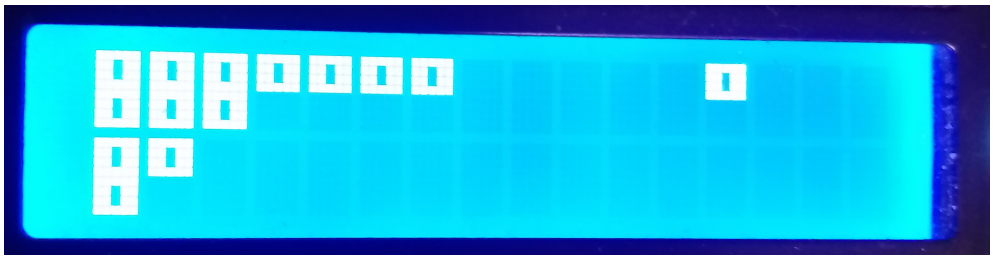
```
?cus1  = x40(x213)
       = x41(x215)
       = x42(x20c)
       = x43(x21f)
       = x44(x20c)
       = x45(x215)
       = x46(x213)
       = x47(x200)
?cus2  = x48(x201)
       = x49(x202)
       = x4a(x20c)
       = x4b(x21e)
       = x4c(x20c)
       = x4d(x202)
       = x4e(x201)
       = x4f(x200)
?loc1  = x50(x08c) % enemy location 1 -- 12up
?guy1  = x51(x201) % enemy 1
?loc2  = x52(x0cf) % enemy location 2 -- 15down
?guy2  = x53(x201) % enemy 2
?loc3  = x54(x092) % enemy location 3 -- 18up
?guy3  = x55(x201) % enemy 3
?loc4  = x56(x0d3) % enemy location 4 -- 19down
?guy4  = x57(x201) % enemy 4
?loc5  = x58(x094) % enemy location 5 -- 20up
?guy5  = x59(x201) % enemy 5
?loc6  = x5a(x0dd) % enemy location 6 -- 29down
?guy6  = x5b(x201) % enemy 6
```

```
?loc7 = x5c(x09f) % enemy location 7 -- 31up
?guy7 = x5d(x201) % enemy 7
?loc8 = x5e(x0a5) % enemy location 8 -- 37up
?guy8 = x5f(x201) % enemy 8
?loc6 = x60(x0e5) % enemy location 9 -- 37down
?guy6 = x61(x201) % enemy 9
```

```
LCD/?x038 % initialize
LCD/?x00c % cursor off
LCD/?x001 % clear
NOP/x00 % END INITIALIZATION
LCD/?x040 % start custom chars
LDF/x80(x040)    % load pointer
//DRAWS/LCP/x00 % perform lcd command @ pointer value
        FIN/x00 % increment flip
        CWF/x81(x062)  % compare with limit
        JMP[fff]/START % jump if done
        JMP[]/DRAWS
//START/LCD/?x081 % goto start
        LCD/"200" % draw ship
        LDF/x70(x081) % set initial position
        NOP/x00 % END DRAW SCREEN
//LOOPA/JIN[01]/UPDOW % check up/down
//LOOPB/JIN[10]/SHOOT % check shot
        JMP[]/SHIFT    % shift screen
        //UPDOW/LCF/x00     % goto ship pos
                LCD/"x220" % draw blank
                FXO/xc0(x040)  % xor position
                LCF/x00     % goto new pos
                LCD/"x200" % draw ship
                JMP[]/LOOPB
                NOP/x00 % END UP/DOWN
        //SHOOT/LCF/x00     % goto pos
                LCD/?x014  % move right
                LCD/"x23d" % draw laser
                LCD/"x23d" % draw laser
                LCD/"x22a" % draw tip
                LCD/?x010  % move left
                LCD/?x004  % change input to left
                LCD/"x220" % draw blank
                LCD/"x220" % draw blank
                LCD/"x220" % draw blank
                LCD/?x006  % change input to right
                JMP[]/SHIFT
                NOP/x00 % END SHOOT
        //SHIFT/LCF/x00  % goto ship pos
                LCD/"x220" % draw blank
                FIN/x00  % increment pos
                LCF/x00  % must set ram address for read apparently
                LCD/?x300 % send char data to regB
                CWB/xc1(x001) % set fff flag if space has enemy
                JMP[fff]/LOSER % jump to end game
```

```
                LCD/?x010 % mem check moves cursor, must reverse!
                LCD/"x200"  % draw ship
                LCD/?x018  % shift left
                JMP[]/LOOPA
                NOP/x00 % END SHIFT
//LOSER/LCD/?x001
        LCD/?x083  %
        LCD/"x247" % G
        LCD/"x261" % a
        LCD/"x26d" % m
        LCD/"x265" % e
        LCD/"x220" %
        LCD/"x24f" % O
        LCD/"x276" % v
        LCD/"x265" % e
        LCD/"x272" % r
        LCD/"x221" % !
        HLT/x00
```

8.2 Stack-a-Block



An incredibly basic game in the spirit of Tetris, albeit lacking rotation and line-clearing. A game over is possible, however.

```
?gogo  = xbb(x038) % no space to intialize in main program!
       = xbc(x00c) % so do it here!
       = xbd(x001)
       = xbe(x006)
       = xbf(x040)
?cus0  = xc0(x21f) % "top" block
       = xc1(x21b)
       = xc2(x21b)
       = xc3(x21f)
       = xc4(x200)
       = xc5(x200)
       = xc6(x200)
       = xc7(x200)
?cus1  = xc8(x200) % "bottom" block
       = xc9(x200)
       = xca(x200)
       = xcb(x200)
       = xcc(x21f)
```

```
       = xcd(x21b)
       = xce(x21b)
       = xcf(x21f)
?cus2  = xd0(x21f) % "full" block
       = xd1(x21b)
       = xd2(x21b)
       = xd3(x21f)
       = xd4(x21f)
       = xd5(x21b)
       = xd6(x21b)
       = xd7(x21f)
?start= xd8(x080) % draw blocks @ left end of screen
       = xd9(x202)
       = xda(x0c0)
       = xdb(x202)
```

```
LDF/x80(x0bb) % load pointer
//BLOCK/LCP/x00 % do stuff!
        FIN/x00 % increment
        CWF/x81(x0dc) % check limit
        JMP[fff]/NEWWW % exit if done
        JMP[]/BLOCK % else loop
//NEWWW/LDF/x82(x08f) % load start pos
        LCF/x00   % goto pos
        LCD/"x200" % draw up
        NOP/x00  % END INITIALIZATION
//CHANG/JIN[01]/LEFFT
        JIN[10]/RIGHT
        JMP[]/LOOOP
//LEFFT/JFC[odd]/DERPA % if down, switch top/bottom
        JMP[]/MYGOD % else don't worry
        //RIGHT/JFC[odd]/MYGOD % if up, don't worry
        //DERPA/LCF/x00 %
                LCD/"x220" % draw blank
                FXO/x70(x040) % xor shit -> switch t/b
        //MYGOD/jad/x71(x800) % switch up/down
//LOOOP/FDE/x00    % decrement flip
        LCF/x00    % goto left of pos
        LCD/?x300  % read data
        CWB/x61(x002) % see if full
        JMP[fff]/STOOP
        CWB/x62(x001) % see if down
        JMP[fff]/CHEKD %
        CWB/x63(x000) % see if up
        JMP[fff]/CHEKU
        LCF/x00 % go back to left of pos
        JFC[odd]/PRNTD % must be blank! check current block
        LCD/"x200" % if up, print that
        JMP[]/REDRW
        //PRNTD/LCD/"x201" % print down
                JMP[]/REDRW
                NOP/x00
```

```
        //CHEKU/JFC[odd]/PRNTF % up on down -> whole
                JMP[]/STOOP % up on up, do nothing
        //CHEKD/JFC[odd]/STOOP % down on down, do nothing
        //PRNTF/LCF/x00 % go back to left of pos
                LCD/"x202" % draw whole block
        //REDRW/FIN/x00     % increment to get to pos
                LCF/x00     % goto pos
                LCD/?x300   % read data
                CWB/x61(x002) % see if full
                LCF/x00 % go back to pos
                JMP[fff]/RUDWN % if full, check if block is up
                LCD/"x220" % else can print space
                JMP[]/MOVIN
        //RUDWN/JFC[odd]/OHYEA % jump if current is down
                LCD/"x201" % draw down
                JMP[]/MOVIN
        //OHYEA/LCD/"x200" % current is down, draw up
                NOP/x00
        //MOVIN/FDE/x00 % set new position
                LCF/x00 % goto pointer
                JMP[]/CHANG
        //STOOP/JFM[fff]/{x80+LOSER}(x00e) % check for game over
                JMP[]/NEWWW % else do new block
//LOSER/LCD/?x001
        LCD/?x087
        LCD/"x221"
        HLT/x00
```

8.3 K.E.N.N.Y. Pong

An attempt at cloning the classic Atari hit, but essentially a one-sided, massive failure. Fun though!

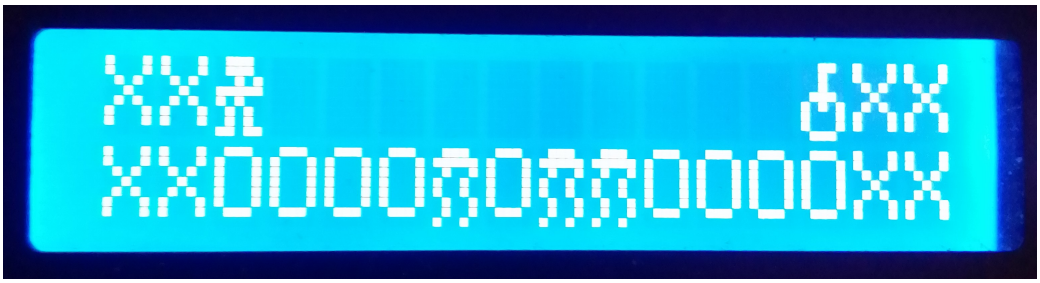
```
?gogo  = xbb(x038) % intialization
       = xbc(x00c)
       = xbd(x001)
       = xbe(x006)
       = xbf(x040)
?cus0  = xc0(x20e) % "top" ball
       = xc1(x21f)
       = xc2(x21f)
       = xc3(x20e)
```

```
       = xc4(x200)
       = xc5(x200)
       = xc6(x200)
       = xc7(x200)
?cus1  = xc8(x200) % "bottom" ball
       = xc9(x200)
       = xca(x200)
       = xcb(x200)
       = xcc(x20e)
       = xcd(x21f)
       = xce(x21f)
       = xcf(x20e)
?start= xd0(x080) % draw paddle
       = xd1(x27c)
       = xd2(x0c0)
       = xd3(x220)
       = xd4(x08e)
       = xd5(x250) % P
       = xd6(x24f) % O
       = xd7(x0ce)
       = xd8(x24e) % N
       = xd9(x247) % G
```

```
LDF/x80(x0bb) % load pointer
//BLOCK/LCP/x00 % do stuff!
        FIN/x00 % increment
        CWF/x81(x0da) % check limit
        JMP[fff]/NEWWW % exit if done
        JMP[]/BLOCK % else loop
//NEWWW/LDF/x82(x088) % load starting ball
        NOP/x00 % END INITIALIZATION
//BALLL/LCF/x00 % goto pos
        LCD/"x220" % clear ball
        LCD/?x300 % get data -- holy shit this works!
        CWB/xa0(x000) % see if top
        JMP[fff]/CEILY % jump if top
        JFM[fff]/{x40 + HITTT}(x040) % must be bottom... so check floor
        //NOHIT/FXO/xa1(x080) % no collision? xor to set bot->top
        //WHCHW/JFM[fff]/{x40 + MOVED}(x400) % jump if move down
                JFM[fff]/{x40 + FLIPP}(x040) % must be moving up... jump if down
                JMP[]/MOVIN
        //CEILY/JFM[fff]/{x40 + WHCHW}(x040) % check if down
                FXO/xa1(x080) % xor top/bot
        //HITTT/FXO/xa1(x080) % xor top/bot
                FXO/xa2(x400) % xor move up/down
        //MOVIN/JFC[odd]/RIGHT % jump if move right
        //LEFFT/FAD/xa3(x00e) % add 14 to flip
                JFM[fff]/{x40 + LCHEK}(x00f) % check flip = 1
                FSU/xa4(x00f) % guess not! so subtract 15...
        //PRINT/JFM[fff]/{x40 + PRNTB}(x080) % see if bot
                FOR/xa1(x080) % if not, must or w/ x080
                LCF/x00 % goto pos
```

```
        LCD/"x200"  % print top
        JMP[]/INPUT % jump out
//PRNTB/LCF/x00  % goto pos
        LCD/"x201"  % print bot
        NOP/x00
//INPUT/JIN[01]/LPADD % check left button
        JIN[10]/RPADD % check right button
        JMP[]/BALLL
//LPADD/LCD/?x080 % goto left pos
        LCD/"x27c" % print paddle
        LCD/?x0c0 %
        LCD/"220" % print blank
        JMP[]/BALLL
//RPADD/LCD/?x080
        LCD/"x220" % print blank
        LCD/?x0c0 % goto right pos
        LCD/"27c" % print paddle
        JMP[]/BALLL
        NOP/x00
//MOVED/JFM[fff]/{x40 + MOVIN}(x040) % jump to move if down
//FLIPP/FXO/xa6(x040) % xor up/down
        JMP[]/MOVIN
//RIGHT/JFM[fff]/{x40 + SLAPP}(x00d) % check flip = 13
        FIN/x00 % no? okay to increment!
        JMP[]/PRINT % else print
//LCHEK/FDE/xa7(x00f) % subtract 15
        FOR/xa1(x080) % or w/ x080
        LCF/x00 % goto pos
        LCD/?x300 % get data
        CWB/xa5(x020) % see if empty
        JMP[fff]/NEWWW % if so, oops! new ball
        FIN/x00 % have to increment now!
//SLAPP/FAD/xa8(x800) % change move l/r
        JMP[]/PRINT % now print what was there!
```

8.4 Jellyfish Bridge



An homage to the well-received "Purple Turtles" for the Commodore 64. Pretty sweet, but hard as shit!

```
?gogo = x90(x038)
```

```
       = x91(x00c) % use x00f initially!
       = x92(x001) % clear screen
       = x93(x006) % cursor moves right
       = x94(x083) % goto 0,3
       = x95(x24c) % L
       = x96(x26f) % o
       = x97(x261) % a
       = x98(x264) % d
       = x99(x269) % i
       = x9a(x26e) % n
       = x9b(x267) % g
       = x9c(x22e) % .
       = x9d(x22e) % .
       = x9e(x22e) % .
       = x9f(x040) % start custom chars
?cus0  = xa0(x20e) % walking dude
       = xa1(x20e)
       = xa2(x205)
       = xa3(x21f)
       = xa4(x214)
       = xa5(x20a)
       = xa6(x20a)
       = xa7(x21b)
?cus1  = xa8(x20e) % turtle up
       = xa9(x21f)
       = xaa(x209)
       = xab(x212)
       = xac(x212)
       = xad(x209)
       = xae(x209)
       = xaf(x212)
?cus2  = xb0(x200) % turtle down
       = xb1(x200)
       = xb2(x20e)
       = xb3(x21f)
       = xb4(x209)
       = xb5(x212)
       = xb6(x209)
       = xb7(x212)
?cus3  = xb8(x206) % cherry
       = xb9(x204)
       = xba(x204)
       = xbb(x20e)
       = xbc(x215)
       = xbd(x211)
       = xbe(x211)
       = xbf(x20e)
?start= xc0(x080) % goto 0,0
       = xc1(x258) % X
       = xc2(x258) % X
       = xc3(x200) % dude
       = xc4(x220)
```

```
       = xc5(x220)
       = xc6(x220)
       = xc7(x220)
       = xc8(x220)
       = xc9(x220)
       = xca(x220)
       = xcb(x220)
       = xcc(x220)
       = xcd(x220)
       = xce(x203) % cherry
       = xcf(x258) % X
       = xd0(x258) % X
       = xd1(x0c0) % goto 1,0
       = xd2(x258) % X
       = xd3(x258) % X
       = xd4(x24f) % O
       = xd5(x24f) % O
       = xd6(x24f) % O
       = xd7(x24f) % O
       = xd8(x201) % turtle 1
       = xd9(x24f) % O
       = xda(x201) % turtle 2
       = xdb(x201) % turtle 3
       = xdc(x24f) % O
       = xdd(x24f) % O
       = xde(x24f) % O
       = xdf(x24f) % O
       = xf0(x258) % X
       = xf1(x258) % X
```

```
//NEWWW/LDF/x80(x090) % load pointer
//MAPPP/LCP/x00 % do stuff!
        FIN/x00 % increment
        CWF/x81(x0f2) % check limit
        JMP[fff]/START % exit if done
        JMP[]/MAPPP % else loop
//START/LDF/x82(x082) % load initial pos
//LOOOP/LCD/?x014 % move cursor right
        JIN[01]/LMOVE
        JIN[10]/RMOVE
        JMP[]/LOOOP
//LMOVE/JFC[odd]/GOLEF
        JMP[]/TURTS
        //GOLEF/FXO/x83(x800)
                JMP[]/TURTS
//RMOVE/FOR/x83(x800)
//TURTS/LCD/?x100
        CWB/x84(x04a)
        JMP[car]/TURT1 % jump if x4a > address
        CWB/x85(x055)
        JMP[car]/TURT2 % jump if x55 > address
        CWB/x86(x060)
```

```
23          JMP[car]/TURT3 % jump if x60 > address
24          JMP[]/CHECK
25          //TURT1/LCD/?0c4
26                  JMP[]/SWITC
27          //TURT2/LCD/?0c6
28                  JMP[]/SWITC
29          //TURT3/LCD/?0c7
30          //SWITC/LCD/?300 % get current turtle
31                  CWB/x87(x002) % see if down
32                  LCD/?x010 % must move left!
33                  JMP[fff]/TDOWN
34                  LCD/"202" % up? print down
35                  JMP[]/CHECK
36                  //TDOWN/LCD/"201" % move turtle up
37          //CHECK/LCF/x00 % goto pos
38                  LCD/"220" % draw blank
39                  JFC[odd]/RIGHT % jump if moving right
40                  CWF/x88(x082) % see if left wall
41                  JMP[fff]/WALKK
42                  FDE/x00 % decrement
43                  CWF/x89(x482) % see if moving to score
44                  JMP[fff]/SCORE
45                  JMP[]/WALKK
46          //RIGHT/CWF/x8a(x88d) % see if right wall is next
47                  JMP[fff]/WALKK
48                  FIN/x00 % increment
49                  CWF/x8b(x88d) % see if moving to cherry
50                  JMP[fff]/CHERY
51          //WALKK/LCF/x00
52                  LCD/"200" % draw dude
53                  FXO/x8c(x040) % xor to go below
54                  LCF/x00 % goto pos
55                  LCD/?x300 % get data
56                  CWB/x8d(x002) % see if down turtle
57                  JMP[fff]/NEWWW
58                  FXO/x8c(x040) % xor to come back
59                  JMP[]/LOOOP
60          //SCORE/LCD/?x08d
61                  LCD/"x203" % draw cherry!
62          //CHERY/FXO/x8e(x400) % toggle cherry in inventory
63                  JMP[]/WALKK % keep walking
```

9 Next Steps

Although K.E.N.N.Y. has come an incredible distance in five iterations, he undoubtedly has further to go. We briefly summarize the pros/cons of the current machine, and discuss what will provide v6.0 the proverbial leg-up.

What to keep

• No other hardware – EEPROMS programmable by hand, no cross-assemblers necessary.

• Reliability – non-volatile memory and low error rate when properly powered.

• Simplicity – a powerful tool for sanity and education.

• Philosophy – make a simple computer that can be enjoyed and understood.

What to remove

• Substitution of ICs – buffers were made with ANDs and diodes, (de-)multiplexers with ANDs and ORs; substantially affects space requirements.

• "Daisy chain" power supply topology – ground loops and voltage inadequacies are almost impossible to avoid.

What to add

• True RAM – obviously! Ability to read LCD is nice, but wildly inefficient.

• Stack pointers and general purpose registers – enable less "clunky" recursion, and more efficient algorithms in general.

• Move to printed circuit board (PCB) for modularity, ease of construction, and erasure of power supply problems.

• More I/O – a keyboard buffer, output register, and interrupts would open many avenues (ease of use, game development, etc.).

• Operating system – despite the challenge, it would be wonderful to create a more universal venue for software. The dream would be the ability to write assembly code.

Thanks for your time!