Introduction

The “Flyer-1” is a simulated computer system built in Logisim, based on 24-bit Von Neumann architecture that includes both data and instructions in a 24-bit address wide memory, amounting to 50.3MByte (16.7MWord x 3Byte) available for operation. The computer system communicates with peripheral systems via 8 digital ports and 4 (simulated) analog ports, controlled via input/output instructions. The ALU is capable of executing binary arithmetic (add, subtract, multiply, divide) and bitwise logic operations (AND, OR, XOR, NOT, shift, rotate), operating on both integer and fixed-point numbers.

Instruction Set Architecture

Instruction Set

The “Flyer-1” is designed according to following requirements:

1. 24-bit wide data bus (16.777.216 capable values) and address bus (16.777.216 locations or 50.331.648 bytes). This is to allow processing of wider data ranges. A binary word here is defined as this 24-bit value.
2. Minimum register count for simplicity in instruction set and control wire timings. Hence, instructions would mainly involve direct memory access and essential register only, e.g. AX (ALU accumulator register).
3. Separated data and address bus, to allow faster memory access.
4. Arithmetic operations are more hardware-based to increase computation speed, through implementations of arithmetic circuits (multiplier, divider, shifter etc.)
5. Conditional branching capability so that programmers can write functions; essentially making the computer “Turing complete”.
6. Digital and analog I/O, to enable future implementations of this computer in embedded systems that also require analog control.

The design requirements mentioned above results in a computer system with following register layout as written in Table 2.1, and are capable of executing 48 instructions, all listed in Table 2.3.

B_{23}–B_{20}	B_{19}–B_{16}	B_{15}–B_{12}	B_{11}–B_{8}	B_{7}–B_{4}	B_{3}–B_{0}	**Bit Position**
Main Register						
IR_H						Instruction Register*
AX_H						Accumulator
TX_H						Temporary*
Pointer Register						
PC_H						Program Counter
SP_H						Stack Pointer*
IA_H						Instruction Pointer*
Status Register						
				$0CAE_B$	$ZNPO_B$	Flag Register*

*Table 2.1. Flyer-1 Register Layout. *Only reserved for internal use.*

Each instruction comprises of an opcode word, often followed by 2 or 3 operand words. The opcode acts as carrier of information, regarding what instruction must be executed; this is imprinted on most significant 9 bits of the opcode word. Operand words, on the

other hand, may act as either an immediate value to load into registers, or address pointers that is then loaded into program counter. The configuration of individual bits for opcode and operand words are described by Table 2.2 below. (X = ignored)

Bit Position	B_{23}–B_{19}	B_{18}–B_{15}	B_{14}–B_0
Opcode	Inst.	Compl.	X X X X X X X X X X X X X X X
Operand 1	Address Pointer / Immediate		
Operand 2	Address Pointer		

Table 2.2. Flyer-1 Opcode and Operand Word Layout.

Below are listed all the 43 instructions valid for use within the Flyer-1 computer system.

Instruction	Description
LDA	Load to accumulator with addressed data
LDI	Load to accumulator with immediate data
STA	Store from accumulator with addressed data
STI	Store addressed immediate data
TRX	Transfer data between addressed locations
JMP	Jump (load PC) to next instruction at address
JCB	Jump if carry or borrow
JNC	Jump if not carry nor borrow
JAL	Jump if A > B
JNA	Jump if not A > B
JEQ	Jump if A = B
JNQ	Jump if not A = B
JZR	Jump if A is zero
JNZ	Jump if A is non-zero
JNG	Jump if A sign negative
JPS	Jump if A sign positive
JPE	Jump if A parity even
JPO	Jump if A parity odd
CLF	Clear all flags
DIP	Digital input addressed from port
DOP	Digital output addressed to port
AIP	Analog input addressed from port
AOP	Analog output addressed to port
PST	Push addressed data to stack
POP	Pop stack to addressed data
NOP	No operation (idle)
HLT	Halt
ADD	Add addressed data
SUB	Subtract addressed data
MUL	Multiply addressed data
DIV	Divide addressed data
ANA	Logic AND accumulator with addressed data

ORA	Logic OR accumulator with addressed data
XRA	Logic XOR accumulator with addressed data
NOT	Logic NOTaccumulator
CMP	Compare addressed data, store flags/status
SHL	Shift left addressed data
SHR	Shift right addressed data
ROL	Roll left addressed data
ROR	Roll right addressed data
CLA	Clear accumulator
ITF	Integer to fixed-point conversion
FTI	Fixed-point to integer conversion

Table 2.3. Flyer-1 Instruction Set.

The following table (2.4) lists every possible addressing modes of each instructions, along with respective word length. For further reading, refer to the chapter “Programming Guide”.

$PPPP_H$ → Addressed; $PPPP_H$ is used as an address pointer
#$PPPP_H$ → Immediate; $PPPP_H$ is used as immediate value
PORT_D → Digital Port
PORT_A → Analog Port
[XXXX] → Contents of device/location XXXX (not for registers)

Instruction	Operand	Length	Action
LDA	$\$AAAA_H$	2	AX <= [$\$AAAA_H$]
LDI	#$DDDD_H$	2	AX <= #$DDDD_H$
STA	$\$AAAA_H$	2	[$\$AAAA_H$] <= AX
STI	$\$AAAA_H$,#$DDDD_H$	3	[$\$AAAA_H$] <= #$DDDD_H$
TRX	$\$XXXX_H$,$\$YYYY_H$	3	[$\$YYYY_H$] <= [$\$XXXX_H$]
JMP	$\$AAAA_H$	2	PC = $\$AAAA_H$
JCB	$\$AAAA_H$	2	PC = $\$AAAA_H$
JNC	$\$AAAA_H$	2	PC = $\$AAAA_H$
JAL	$\$AAAA_H$	2	PC = $\$AAAA_H$
JNA	$\$AAAA_H$	2	PC = $\$AAAA_H$
JEQ	$\$AAAA_H$	2	PC = $\$AAAA_H$
JNQ	$\$AAAA_H$	2	PC = $\$AAAA_H$
JZR	$\$AAAA_H$	2	PC = $\$AAAA_H$
JNZ	$\$AAAA_H$	2	PC = $\$AAAA_H$
JNG	$\$AAAA_H$	2	PC = $\$AAAA_H$
JPS	$\$AAAA_H$	2	PC = $\$AAAA_H$
JPE	$\$AAAA_H$	2	PC = $\$AAAA_H$
JPO	$\$AAAA_H$	2	PC = $\$AAAA_H$
CLF	-	1	FLAG = $0000\ 0000_B$
DIP	$\$AAAA_H$,PORT_D	2	[$\$AAAA_H$] <= [PORT_D]
DOP	$\$AAAA_H$,PORT_D	2	[PORT_D] <= [$\$AAAA_H$]
AIP	$\$AAAA_H$,PORT_A	2	[$\$AAAA_H$] <= [PORT_A]
AOP	$\$AAAA_H$,PORT_A	2	[PORT_A] <= [$\$AAAA_H$]
PST	$\$AAAA_H$	2	[$\$FFFX_H$] <= [$\$AAAA_H$]; SP(X)++

POP	$AAAA$_H$	2	[$AAAA$_H$] <= [$FFFX$_H$]; SP(X)--
NOP	-	1	-
HLT	-	1	-
ADD	$XXXX$_H$,$YYYY$_H$	3	AX = [$XXXX$_H$] + [$YYYY$_H$]
SUB	$XXXX$_H$,$YYYY$_H$	3	AX = [$XXXX$_H$] - [$YYYY$_H$]
MUL	$XXXX$_H$,$YYYY$_H$	3	AX = [$XXXX$_H$] * [$YYYY$_H$]
DIV	$XXXX$_H$,$YYYY$_H$	3	AX = [$XXXX$_H$] / [$YYYY$_H$]
ANA	$AAAA$_H$	2	AX = AX & [$AAAA$_H$]
ORA	$AAAA$_H$	2	AX = AX | [$AAAA$_H$]
XRA	$AAAA$_H$	2	AX = AX ⊕ [$AAAA$_H$]
NOT	-	1	AX = !AX
CMP	$XXXX$_H$,$YYYY$_H$	3	AX = [$XXXX$_H$] + [$YYYY$_H$]
SHL	$XXXX$_H$,$YYYY$_H$	3	AX = [$XXXX$_H$] + [$YYYY$_H$]
SHR	$XXXX$_H$,$YYYY$_H$	3	AX = [$XXXX$_H$] + [$YYYY$_H$]
ROL	$XXXX$_H$,$YYYY$_H$	3	AX = [$XXXX$_H$] + [$YYYY$_H$]
ROR	$XXXX$_H$,$YYYY$_H$	3	AX = [$XXXX$_H$] + [$YYYY$_H$]
CLA	-	1	AX = 0
ITF	$AAAA$_H$	2	**Not yet implemented**
FTI	$AAAA$_H$	2	**Not yet implemented**

Table 2.4. Flyer-1 Instruction Addressing Modes.

Data Structures and Memory Specifications

Flyer-1 computer comes with built-in data formats of integer, fixed-point and 3-character ASCII word. Word configuration of each data formats are shown in Table 2.5. uture versions would include more data formats with variable length and functionality (incl. string, unsigned 32-bit/64-bit integer/floating-point).

B_{23}–B_0 (Whole)			Integer (int)
B_{23}–B_8 (Whole)		**B_7–B_0** (Fractional)	Fixed-Point (fix)
B_{23}–B_{16} (Char 1)	**B_{15}–B_8** (Char 2)	**B_7–B_0** (Char 3)	3-char ASCII Word (asc)

Table 2.5. Flyer-1 Built-in Data Formats.

The addressable memory size for Flyer-1 computer is 16.777.216 locations (2^24 bits). This memory is divided into segments as shown in Table 2.6, separating the locations in which instructions, static and dynamic variables are stored. This is done to provide memory safety, preventing mistaken interpretation of instructions as data, and vice versa. Memory allocation of variables and instructions are hard-coded in the assembler program, which can flexibly be adjusted according to system/programmer’s requirements.

0000_H$ $5FFF$_H$	Instruction segment; stores all instructions necessary to run the computer, incl. assembler.
6000_H$ $7FFF$_H$	Static variable segment; data stored within this segment won't change during runtime.
8000_H$ $BFFF$_H$	Dynamic variable segment; data stored within this segment can be altered flexibly.
$C000$_H$ $FFEF$_H$	Reserved/Empty segment; used to store externally-loaded programs or data.
$FFF0$_H$ $FFFF$_H$	Stack segment; only accessible via PST and POP instructions.

Table 2.6. Flyer-1 Memory Segmentation.

Control Unit

Upon fetching an instruction, as the opcode word is loaded into Instruction Register, combinational logic circuits made from arrays of AND, OR and NOT gates decodes this instruction into control words. These control words are what commands which register to output into, or input from the data/address buses; what operations must the ALU perform; whether a conditional jump can be triggered, and so on.

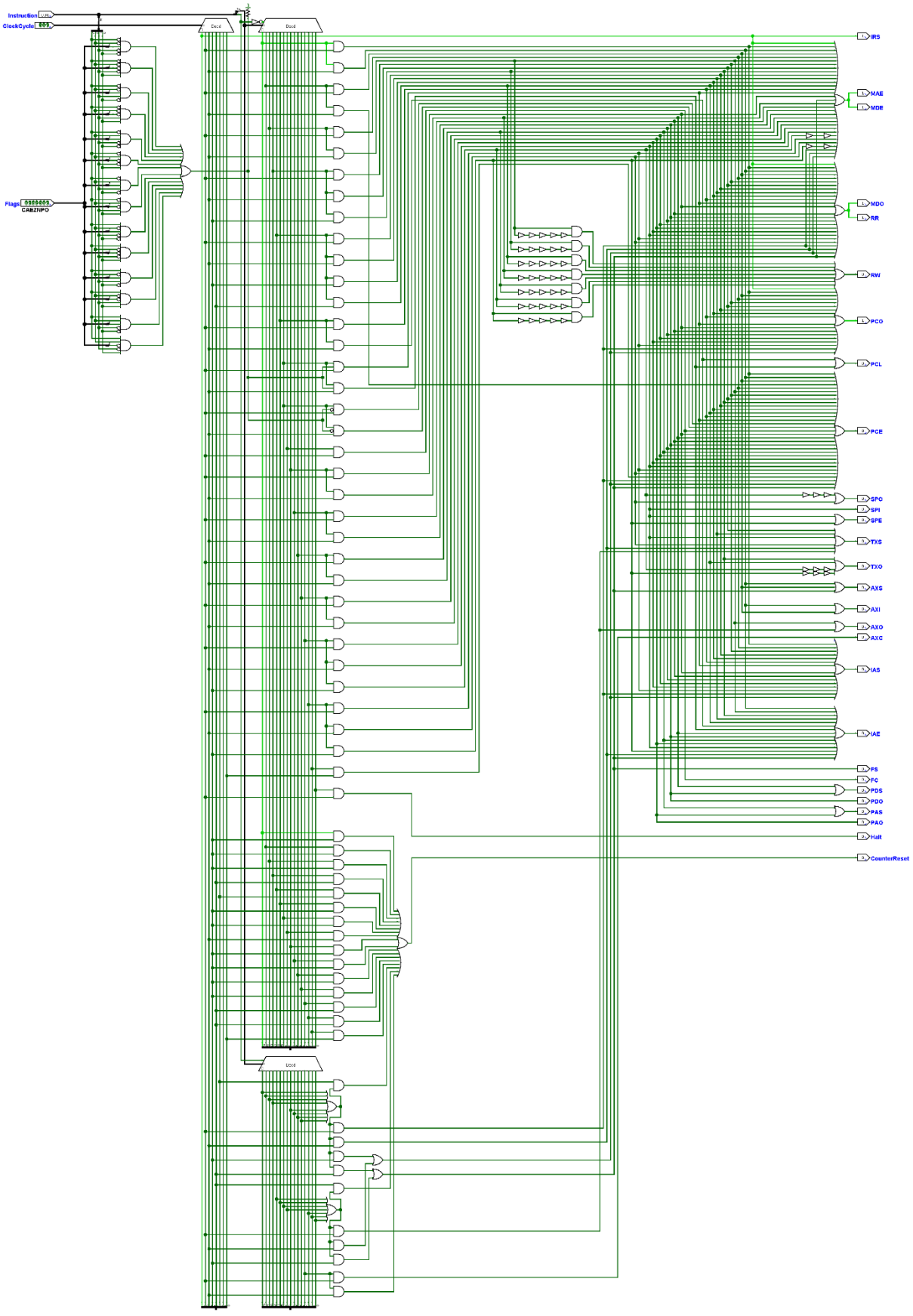

Figure 2.1. Flyer-1 Control Unit.

In execution cycle, if two register simultaneously outputs data into the same bus, there may exist conflicting signals, which prevents the computer from working properly. In Logisim, this is represented by red-colored 'Error' (E) message on the bus. To prevent this, exact timing of control words are required, hence implemented a clock divider unit.

The clock divider unit is connected to a digital oscillator, which outputs square wave at predictable and constant intervals. One clock cycle equals to one sequence of high-low signal. By connecting this into 3-bit binary counters, we can extract how many clock cycles have occurred. By connecting this unit into the combinational logic circuit via decoders, the computer can accurately time signals to control registers / counters / buffers.

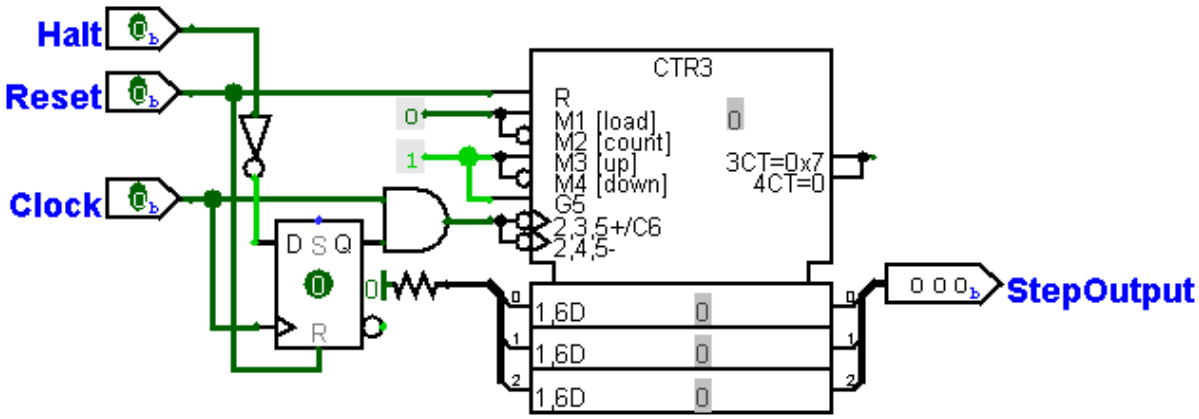

Figure 2.2. Flyer-1 Clock Divider Unit.

Upon executing conditional jump instructions, the computer needs to know whether the jump conditions specified by the opcode word are identical to the computer's current status or flags. For this, a flag comparator unit is required, which A

The resulting control words are 26-bit wide, with each bits' functions written on Table 2.7 below.

Bit Position	Control Bit	Description
C_{25}	IRS	Instruction Register Set
C_{24}	MAE	Memory Address Buffer Enable
C_{23}	MDE	Memory Data Buffer Enable
C_{22}	MDO	Memory Data Buffer Output (ito data bus)
C_{21}	RR	RAM Read
C_{20}	RW	RAM Write
C_{19}	PCO	Program Counter Output (to address bus)

C_{18}	PCL	Program Counter Load New Address
C_{17}	PCE	Program Counter Enable Count
C_{16}	SPO	Stack Pointer Output (to address bus)
C_{15}	SPI	Stack Pointer Increment Count
C_{14}	SPE	Stack Pointer Enable Count
C_{13}	TXS	Temporary Set
C_{12}	TXO	Temporary Output (to data bus)
C_{11}	AXS	Accumulator Set
C_{10}	AXI	Accumulator Input (from data bus)
C_{9}	AXO	Accumulator Output (to data bus)
C_{8}	AXC	Accumulator Clear
C_{7}	IAS	Instruction Pointer Set
C_{6}	IAE	Instruction Pointer Output (to address bus)
C_{5}	FS	Flag Set
C_{4}	FC	Flag Clear
C_{3}	PDS	Port Digital Select
C_{2}	PDO	Port Digital Output Direction
C_{1}	PAS	Port Analog Select
C_{0}	PAO	Port Analog Output Direction

Table 2.7. Flyer-1 Control Word Configuration.

Instruction Execution Table

The following Table 2.8 includes all details regarding clock cycle count for each instruction's execution, control word timing, and actions performed. Additional notes to aid with reading the table are as follows:

X Flag bits would always follow the ALU's output flags
- Flag bits aren't affected / won't change
REG Register REG directly referred for its contents
[REG] Register REG referred as pointer to memory address, e.g. "Contents of memory address $REG"

Mne.	N_{CC}	Operation	W_{CTRL}	W_{OPC}	WL	Flags							Description
						C	A	E	Z	N	P	O	
Fetch	*0*	IR <= [PC]	3E80000	*N/A*	*N/A*	-	-	-	-	-	-	-	Fetches instruction to IR
LDA	1	IA <= [PC]; PC++	1EA0080	000000	2	-	-	-	-	-	-	-	Address pointer loaded into IA
	2	AX <= [IA]; PC++	1E20C40			-	-	-	-	-	-	-	Contents at address $IA loaded into AX
LDI	1	AX <= [PC]; PC++	1EA0080	080000	2	-	-	-	-	-	-	-	Immediate value into AX
	2	PC++	0020000			-	-	-	-	-	-	-	PC points to next instr.
STA	1	IA <= [PC]; PC++	1EA0080	100000	2	-	-	-	-	-	-	-	Address pointer loaded into IA
	2	[IA] <= AX; PC++	1920240			-	-	-	-	-	-	-	Contents at AX stored to address $IA
STI	1	IA <= [PC]; PC++	1EA0080	180000	3	-	-	-	-	-	-	-	Address pointer loaded into IA
	2	TX <= [PC]; PC++	1EA2000			-	-	-	-	-	-	-	Immediate value loaded into TX
	3	[IA] <= TX; PC++	1921040			-	-	-	-	-	-	-	Contents at TX stored to address $IA
TRX	1	IA <= [PC]; PC++	1EA0080	200000	3	-	-	-	-	-	-	-	Loads origin address to IA
	2	TX <= [IA]	1E02040			-	-	-	-	-	-	-	Contents at $IA loaded into TX
	3	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	Loads destination address to IA
	4	[IA] <= TX; PC++	1921040			-	-	-	-	-	-	-	Contens at TX stored to address $IA
JMP	1	IA <= [PC]; PC++	1EA0080	280000	2	-	-	-	-	-	-	-	Address pointer loaded into IA
	2	PC <= IA	0040040			-	-	-	-	-	-	-	IA loaded into PC, eff. points to next
JCB	1	IA <= [PC]; PC++	1EA0080	300000	2	-	-	-	-	-	-	-	Only happens if Carry or Borrow is 1
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
JNC	1	IA <= [PC]; PC++	1EA0080	308000	2	-	-	-	-	-	-	-	Only happens if no Carry or Borrow
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
JAL	1	IA <= [PC]; PC++	1EA0080	310000	2	-	-	-	-	-	-	-	Only happens if A>B is True
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
JNA	1	IA <= [PC]; PC++	1EA0080	318000	2	-	-	-	-	-	-	-	Only happens if A>B is False
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
JEQ	1	IA <= [PC]; PC++	1EA0080	320000	2	-	-	-	-	-	-	-	Only happens if A=B is True
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
JNQ	1	IA <= [PC]; PC++	1EA0080	328000	2	-	-	-	-	-	-	-	Only happens if A=B is False
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.

JZR	1	IA <= [PC]; PC++	1EA0080	330000	2	-	-	-	-	-	-	-	Only happens if A=0 is True
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
JNZ	1	IA <= [PC]; PC++	1EA0080	338000	2	-	-	-	-	-	-	-	Only happens if A=0 is False
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
JNG	1	IA <= [PC]; PC++	1EA0080	340000	2	-	-	-	-	-	-	-	Only happens if A sign is Negative
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
JPS	1	IA <= [PC]; PC++	1EA0080	348000	2	-	-	-	-	-	-	-	Only happens if A sign is Positive
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
JPE	1	IA <= [PC]; PC++	1EA0080	350000	2	-	-	-	-	-	-	-	Only happens if A parity is even (0)
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
JPO	1	IA <= [PC]; PC++	1EA0080	358000	2	-	-	-	-	-	-	-	Only happens if A parity is odd (1)
	2	PC <= IA	0040040			-	-	-	-	-	-	-	Else, PC increments to next instr.
CLF	1	FLAG = 00_H; PC++	0020010	380000	1	0	0	0	0	0	0	0	Clears all flags
DIP	1	IA <= [PC]; PC++	1EA0080	4P0000	2	-	-	-	-	-	-	-	Address pointer loaded into IA
	2	[IA] <= P_D; PC++	1A20048			-	-	-	-	-	-	-	Contents of Digital Port stored at $IA
DOP	1	IA <= [PC]; PC++	1EA0080	4Q0000	2	-	-	-	-	-	-	-	Address pointer loaded into IA
	2	P_D <= [IA]; PC++	1E2004C			-	-	-	-	-	-	-	Contents of $IA output to Digital Port
AIP	1	IA <= [PC]; PC++	1EA0080	5R0000	2	-	-	-	-	-	-	-	Address pointer loaded into IA
	2	[IA] <= P_A; PC++	1A20042			-	-	-	-	-	-	-	Contents of Analog Port stored at $IA
AOP	1	IA <= [PC]; PC++	1EA0080	5S0000	2	-	-	-	-	-	-	-	Address pointer loaded into IA
	2	P_A <= [IA]; PC++	1E20043			-	-	-	-	-	-	-	Contents of $IA output to Analog Port
PST	1	IA <= [PC]; PC++	1EA0080	600000	2	-	-	-	-	-	-	-	Address pointer loaded into IA
	2	TX <= [IA]; PC++	1E2E040			-	-	-	-	-	-	-	Contents of $IA loaded into TX
	3	STACK <= TX	1A11000			-	-	-	-	-	-	-	TX pushed to stack
POP	1	IA <= [PC]; PC++	1EA0080	680000	2	-	-	-	-	-	-	-	Address pointer loaded into IA
	2	TX <= STACK; PC++	1E32000			-	-	-	-	-	-	-	Stack pops into TX
	3	[IA] <= TX	1A05040			-	-	-	-	-	-	-	Contents of TX stored into $IA
NOP	7	PC++	0020000	700000	1	-	-	-	-	-	-	-	Idles for 7 cycles
HLT	1	HALT	0000000	780000	1	-	-	-	-	-	-	-	Halt until interrupt/reset
CLA	1	AX = 000000_H; PC++	0020100	E40000	1	X	X	X	1	X	X	X	Clears AX

ADD	1	IA <= [PC]; PC++	1EA0080	800000	3	-	-	-	-	-	-	-	1st pointer into IA
	2	TX <= [IA]	1E02040			-	-	-	-	-	-	-	Contents of $IA (operand 1) into TX
	3	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	2nd pointer into IA (operand 2)
	4	AX = TX+[IA]; PC++	1E20860			X	X	X	X	X	X	X	AX stores addition of TX and [$IA]
SUB	1	IA <= [PC]; PC++	1EA0080	880000	3	-	-	-	-	-	-	-	1st pointer into IA
	2	TX <= [IA]	1E02040			-	-	-	-	-	-	-	Contents of $IA (operand 1) into TX
	3	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	2nd pointer into IA (operand 2)
	4	AX = TX-[IA]; PC++	1E20860			X	X	X	X	X	X	X	AX stores subtraction of TX and [$IA]
MUL	1	IA <= [PC]; PC++	1EA0080	900000	3	-	-	-	-	-	-	-	1st pointer into IA
	2	TX <= [IA]	1E02040			-	-	-	-	-	-	-	Contents of $IA (operand 1) into TX
	3	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	2nd pointer into IA (operand 2)
	4	AX = TX*[IA]; PC++	1E20860			X	X	X	X	X	X	X	AX stores multiply of TX and [$IA]
DIV	1	IA <= [PC]; PC++	1EA0080	980000	3	-	-	-	-	-	-	-	1st pointer into IA
	2	TX <= [IA]	1E02040			-	-	-	-	-	-	-	Contents of $IA (operand 1) into TX
	3	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	2nd pointer into IA (operand 2)
	4	AX = TX/[IA]; PC++	1E20860			X	X	X	X	X	X	X	AX stores division of TX and [$IA]
SHL	1	IA <= [PC]; PC++	1EA0080	C80000	3	-	-	-	-	-	-	-	1st pointer into IA
	2	TX <= [IA]	1E02040			-	-	-	-	-	-	-	Contents of $IA (operand 1) into TX
	3	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	2nd pointer into IA (operand 2)
	4	AX = TX<<[IA]; PC++	1E20860			X	X	X	X	X	X	X	AX stores TX shift-left by [$IA]
SHR	1	IA <= [PC]; PC++	1EA0080	CA0000	3	-	-	-	-	-	-	-	1st pointer into IA
	2	TX <= [IA]	1E02040			-	-	-	-	-	-	-	Contents of $IA (operand 1) into TX
	3	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	2nd pointer into IA (operand 2)
	4	AX = TX>>[IA]; PC++	1E20860			X	X	X	X	X	X	X	AX stores TX shift-right by [$IA]
ROL	1	IA <= [PC]; PC++	1EA0080	CC0000	3	-	-	-	-	-	-	-	1st pointer into IA
	2	TX <= [IA]	1E02040			-	-	-	-	-	-	-	Contents of $IA (operand 1) into TX
	3	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	2nd pointer into IA (operand 2)
	4	AX = TX|<[IA]; PC++	1E20860			X	X	X	X	X	X	X	AX stores TX cyclic-left by [$IA]
ROR	1	IA <= [PC]; PC++	1EA0080	CE0000	3	-	-	-	-	-	-	-	1st pointer into IA
	2	TX <= [IA]	1E02040			-	-	-	-	-	-	-	Contents of $IA (operand 1) into TX

	3	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	2nd pointer into IA (operand 2)
	4	AX = TX>|[IA]; PC++	1E20860			X	X	X	X	X	X	X	AX stores TX cyclic-right by [$IA]
CMP	1	IA <= [PC]; PC++	1EA0080	C00000	3	-	-	-	-	-	-	-	1st pointer into IA
	2	TX <= [IA]	1E02040			-	-	-	-	-	-	-	Contents of $IA (operand 1) into TX
	3	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	2nd pointer into IA (operand 2)
	4	F = TX?=[IA]; PC++	1E20860			X	X	X	X	X	X	X	F stores comparison of TX with [$IA]
ANA	1	TX = AX	0002200	A00000	2	-	-	-	-	-	-	-	AX transferred to TX
	2	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	Pointer into IA
	3	AX = TX&[IA]	1E20860			X	X	X	X	X	X	X	AX stores logical AND of TX with [$IA]
ORA	1	TX = AX	0002200	A80000	2	-	-	-	-	-	-	-	AX transferred to TX
	2	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	Pointer into IA
	3	AX = TX|[IA]	1E20860			X	X	X	X	X	X	X	AX stores logical OR of TX with [$IA]
XRA	1	TX = AX	0002200	B00000	2	-	-	-	-	-	-	-	AX transferred to TX
	2	IA <= [PC]; PC++	1EA0080			-	-	-	-	-	-	-	Pointer into IA
	3	AX = TX⊕[IA]	1E20860			X	X	X	X	X	X	X	AX stores logical XOR of TX with [$IA]
NOT	1	TX = AX	0002200	B80000	2	-	-	-	-	-	-	-	AX transferred to TX
	2	*N/S*	1EA0080			-	-	-	-	-	-	-	*No significance*
	3	AX = !TX	1E20860			X	X	X	X	X	X	X	AX stores logical NOT of TX

Table 2.8. Flyer-1 Instruction Execution Table.

Programming Guide

Introduction

To program the Flyer-1, a custom-built assembly language named Flyer1ASM is used. The languages' design takes inspiration from that of MOS 6502 and early x86 computers, most notably the 8086 and i386. The incorporated design features include addressing modes (although as of right now still not flexible), along with and memory segmentation of variables and instructions.

To write Flyer1ASM programs, one may utilize any text editing softwares, but it is recommended to use Notepad or Visual Studio, to avoid automatic capitalization of each words, for the Flyer1ASM language is case-sensitive. After programs are written, it must be saved with an .asm extension, that can then be assembled into hex files executable by the Flyer-1 emulated / circuit version.

A default Flyer1ASM sourcefile appears as shown by Snippet 3.1.

```
.stat:
     ; this is how comments are written.
.var:
     ; comments can only span
     ; a single line. So, to write
     ; multiple-line comments, you
     ; must write double-slashes.
.text:
     ; you can write comments anywhere!
```

Snippet 3.1. Default sourcefile.

Standalone example of declaring variables are shown by Snippet 3.2.

```
int x,42
; Declare x as integer, where x = 42
fix pi,3.14159
; Declare pi as fixed-point, where pi = 3.14159
fix Tau,6.28318
; Declare Tau as fixed-point, where Tau = 6.28318
asc ErrorMssg2,NaN
; Declare ErrorMssg2 as an ASCII 3-char with chars 'NaN'
asc Greetings,Hw!
; Declare Greetings as an ASCII 3-char with chars 'Hw!'
; This is merely a shorthand for 'Hello World!'
```

Snippet 3.2. Declaring variables.

The **.stat** segment houses declaration of static variables; that is, variables that will not and cannot be manipulated during program execution. This is useful to store constants (pi, e, kb, c, u0 etc.) and program return messages (e.g. Err, NaN, Db0, etc.) On the other hand, variables declared within **.var** segment can be flexibly altered by the instructions that are currently executed. This is where abstracted variables and I/O data are stored. Instructions are written within **.text** segment, which follows the syntaxes described in Instruction Set Architecture chapter.

To assemble your saved program, type the following command line on command prompt or terminal (IMPORTANT: make sure your .asm program file's directory is same as the assembler program Flyer1_Assembler.py file!). Copy the generated text (including the header words), then save it as 'RAMContent.txt'. To execute this in circuit version, open the circuit file and right-click on RAM, select 'Load Image' then search for 'RAMContent.txt' and load it. Press Ctrl+K, and let the magic run.

```
$ python Flyer1_Assembler.py HelloWorld.asm
v3.0 hex words addressed
000000: 490000
000001: 200000
000002: 4a0000
000003: 300000
000004: 780000
200000: 487721
300000: 576f77
$
```

Snippet 3.3. Terminal appearance when assembling the program.

Writing Programs

Hello World!

A basic program that displays "Hello World!" message can be written as the following snippet. Do note that due to limitations of 'asc' variable type, the message is shortened into its initials and signs. Future versions would include long string of characters. Name the following file 'HelloWorld.asm', then assemble and execute it.

```
.stat:
     asc msg1,Hw!     ; analogous to str msg1 = ‘Hw!’
.var:
     asc msg2,Wow     ; analogous to str msg2 = ‘Wow’
.text:
     dop msg1,01      ; output msg1 to digital port 01
     dop msg2,02      ; output msg2 to digital port 02
     hlt              ; end
```

Snippet 3.4. Example “Hello, World!” program.

Fibonacci

The renowned mathematician Leonardo of Pisa, also known as Fibonacci, discovers what is now known as Fibonacci sequence, where the N-th element is the sum of two previous elements before them, given that the 0^{th} and 1^{st} element are 0 and 1. This yields,

0, 1, 1, 2, 3, 5, 8, 13, 21, …

To calculate Fibonacci numbers below 1000, we can utilize the following program, where we assume that the two previous elements each are symbolized as x and y, and its sum is z. Because the calculation requires to check the addition results against the limit of 1000, which then decides to sum back if still below limit, or to stop if reached it, we can write loops indicated by ‘*loopname:’ (it must always end in double-dots).

```
.stat:
     int limit,1000  ; limit the generated fibonacci below 1000
.var:
     int x,0    ; x = 0
     int y,1    ; y = 1
     int z,0    ; z = 0
.text:
count:          ; counting loop
     add x,y          ; AX = x + y
     sta z            ; z = AX, hence z = x + y
     trx y,x          ; x = y
     trx z,y          ; y = z
     cmp x,limit      ; x ?= limit, store flags
     jna count        ; jump if not x>limit to count
     jmp terminate    ; jump to terminate otherwise
terminate:       ; termination loop
     dop x,01         ; output computed fibonacci number
     hlt              ; end
```

Snippet 3.5. Fibonacci number program.

Heron Square Root

Perhaps the "least complicated" advanced math you can do with this computer would be computing the square root of a real number. An ancient algorithm can be utilized, tracing back to the Greek mathematician Heron of Alexandria in 2nd Century. The algorithm involves the argument number S and initial guess x_0, where successive N-th iterations of x (x_N) is computed by,

$$x_N = \frac{1}{2}\left(x_{N-1} + \frac{S}{x_{N-1}}\right)$$

We can implement the program as shown by the code snippet below. It is important to remind ourselves, that square roots are often presented as decimal numbers, hence we use fixed-point vartype for our calculation.

```
.stat:
     int limit,100    ; limit iteration to 100 times
     int div2const,1  ; divide by 2 constant by shifting
     int add1const,1  ; increment by 1 constant
.var:
     int n,0          ; iteration counter
     fix s,85.0       ; S = 85.0
     fix xa,12.0      ; initial guess
     fix xn,0.0       ; placeholder
.text:
count:
     sub s,xa         ; s/x(n-1)
     sta xn
     add xa,xn        ; x(n-1) + s/x(n-1)
     sta xn
     shr xn,div2const ; 0.5*(x(n-1) + s/x(n-1))
     sta xn           ; x(n) = x(n-1)
     trx xn,xa
     add n,add1const
     sta n
     cmp n,limit
     jna count
     jmp end
end:
     dop xn,01
     hlt
```

Snippet 3.6. Heron square root program.