



LIGHTPC MANUAL

Programming, Extending and Modifying LightPC

mark2222

CONTENTS

Introduction.....	3
LightPC Overview.....	3
About LightPC.....	3
Specifications.....	3
Comparison with Other Processors.....	4
Features.....	6
Limitations.....	6
Using LightPC.....	7
Loading Programs.....	7
Running Programs.....	7
Using the Hexadecimal Input Panel.....	7
Using the Touchscreen.....	8
Programming LightPC.....	9
The Programming Environment.....	9
LightASM.....	10
The Two Flavours of LightASM.....	10
Hello World in LightASM.....	10
LightPC Machine Code.....	12
The LightPC Instruction Set.....	13
Speed of Instructions.....	13
Low-Level LightASM.....	14
High-Level LightASM.....	22
Communicating with the Default External Devices.....	26
The Touchscreen.....	26
The Hexadecimal Display.....	27
The Pseudorandom Number Generator and Clock.....	27
Debugging.....	28
Errors Produced by the Cross-Compiler.....	28
Unintended Output.....	28
The Cross-Compiler.....	29
Circuit Design.....	30
LightPC Circuit Idioms.....	30
Logic Gates.....	30
Self-Resetting NOTAND Gates.....	30
Old Spark Delays.....	30

Photon Emitters.....	30
Photon Multipliers.....	30
Piston Demultiplexers.....	31
Filter Buses.....	31
Multi-Input Spark Buses.....	31
Components.....	32
The Instruction Pipeline.....	33
Buses.....	33
The RAM.....	34
The Arithmetic Logic Unit.....	34
The Adder-Subtractor.....	34
The Incrementor-Decrementor.....	35
The Bitshifter.....	35
The Boolean Logic Unit.....	35
The Control Unit.....	36
The Demultiplexer.....	36
The Register Reader.....	36
The Register Writer.....	36
External Devices.....	37
The External Device Communication Protocol.....	37
Device Ports.....	38
The Touchscreen.....	38
The Hexadecimal Display.....	38
The Pseudorandom Number Generator.....	39
The Clock.....	39

INTRODUCTION

Each section of this LightPC manual is targeted towards different audiences:

LightPC Overview: Casual readers who would like to know more about LightPC.

Using LightPC: Users of LightPC who have technical issues operating LightPC, or programmers intending to use LightPC to run their programs.

Programming LightPC: Programmers writing programs for LightPC or extending, modifying or fixing the cross-compiler.

Circuit Design: Engineers wishing to use parts of LightPC for their own creations, creating new devices for LightPC, extending or modifying LightPC, or attempting to learn or glean ideas from LightPC.

Throughout this manual, many references would be made to a non-standard “NOTAND” Boolean operator. In this manual, A NOTAND B means (NOT A) AND B. Do not confuse it with NAND which means NOT (A AND B).

Hope you will have an enjoyable read.

LIGHTPC OVERVIEW

ABOUT LIGHTPC

LightPC is an extensible general purpose 29-bit processor using state-of-the-art photon technology packaged with a touchscreen and a hexadecimal display. It was inspired by the save “HD Video” by drakide.

The name “LightPC” is derived not just from the fact that the processor is lightweight and lightning fast (relatively), but also from the fact that LightPC uses photon technology as its core logic and primary innovation. The “PC” part of the name is a tongue-in-cheek reference to the code-in-RAM and screen support features that allow LightPC to be used for personal computing, albeit very slow and difficult personal computing.

Like any other Powder Toy save, LightPC is not intended to be useful for society in any direct manner. It may be useful for facilitating logic and control for other Powder Toy machines, but often LightPC is overkill for such purposes and a dedicated circuit or even mapS by drakide and Rawing would probably be sufficient. LightPC could potentially be used for educational purposes as it covers a lot of ground pertaining to computer architecture and circuit design.

SPECIFICATIONS

Frames per instruction: 100 (on average)

Speed: 0.00976 instructions per frame

Speed at 60 fps: 0.586 Hz

Speed at 40 fps: 0.390 Hz

Word width: 29 bits

Instruction width: 29 bits (only 25 bits used)

Number of registers: 15

RAM space: 512 words (1.813 KB)

Device ports: 16

Screen resolution: 24 x 24 pixels

Screen pixel size: 10 x 10 px

COMPARISON WITH OTHER PROCESSORS

The file “Processor Specifications Comparison.xlsx” contains a comparison of LightPC with 6 chosen historical Powder Toy processors. Take note that the data there may not be fully reliable. The details of each column are:

Save ID: The Powder Toy save identifier of the processor. Since this manual was written before the publication of LightPC, no save ID for LightPC has been recorded.

Name: The name of the computer endowed to it by its creator. If no official name was given, a “name” derived from the save title was used instead.

Date published: The date when the save containing the processor was published. This gives an indicator of when the processor was designed for better comparison.

Owner: The owner of the save containing the processor. This is likely but not necessarily the creator of the processor.

Signal transfer technology: The main technology used to transfer spark signals around and between components.

Bus technology: The main technology used to transfer words between components.

Delay technology: The main technology used to time spark signals.

Demultiplexer technology: The main technology used to direct a spark signal to one of a set of possible paths based on the value of a word.

ALU technology: The main technology used in the Arithmetic/Logic Unit for computations such as Boolean logic and addition.

Memory storage technology: The main technology used to store variables used in computation.

Speed (instructions per frame): The number of instructions processed per Powder Toy frame. This was experimentally determined by connecting an incrementor to a wire that is sparked every clock cycle to count instructions and using an external timer (also an incrementor) to measure time, then running the default program for a while. The raw data can be accessed in “Processor Speed Measurements.xlsx”. Note that while this provides a good gauge of the order of magnitude of processor speed, the numbers are strongly affected by the default program provided by the processor’s creator. These numbers also depend on the instruction set of the processor – processors based on direct reads and writes into RAM may have more computing power per instruction, allowing it to perform certain tasks faster overall despite a lower processing speed per instruction. For LightPC, the Sieve of Eratosthenes program was used.

Speed (Hz at 60 fps): This was calculated, for the convenience of the layman, by multiplying the instructions-per-frame speed by 60.

Default program outputs correctly: This column denotes whether the processor is working properly as advertised. For devast8a’s processor, there was no output because no default program was provided. For mark2222’s computer, there was no output likely because I did not bother to wait long enough. china-richway2’s processor appears to have a bug since the output produced is wrong. It appears that I am not the only one facing this problem.

Instruction width: The maximum number of bits that an instruction can contain. Includes invalid instructions.

Word width: The number of bits per variable that the processor supports.

Number of registers: The number of registers that the processor allows for. Fake registers such as registers with a fixed value or registers used only for temporary storage are not counted. Using such a definition, some processors do not have any registers since data is directly read from and written to RAM.

RAM space: The number of words that can be both read and overwritten.

ROM space: The number of words that can only be read but not overwritten, usually used to store programs. Some processors store the program along with other data and thus do not have ROM.

External devices allowed: The number of ports available for external devices. Some processors do not explicitly provide ports, but provide some form of input and output devices – these devices are added to this count.

Rectangle of free device space: All components that can be considered external devices are removed, then the

largest possible empty rectangle is measured. This column gives an indicator of how much space the processor takes up and thus its extensibility.

Free device space area: This column computes the area of the previous column for better comparison.

Particles used: The number of particles used after all decorations and devices have been removed, giving an indicator of Powder Toy's performance when simulating the processor. Devices were removed because devices can be swapped for other devices and thus should not be considered when calculating the processor's size. Note that statistics involving the number of particles do not give an indicator of free device space since processors can be very dense.

Parts w/o RAM, ROM or registers: The number of particles used after all decorations, devices and memory storage components have been removed. This column exists to provide a better comparison between the core functionality of processors since memory components can usually be resized to reduce the number of particles as desired.

Parts w/o RAM per word length: The number of particles per bits in a word after all decorations, devices and memory storage components have been removed. This column exists to provide a better comparison between the core functionality of processors since the number of particles used by a processor is usually inflated by the word size of the processor, and the processor can usually be modified to accommodate different word sizes as desired.

Code in RAM: Whether the code is stored externally or together with data. This feature allows the theoretical creation of assemblers and operating systems.

Interrupts: Whether the processor supports interrupts. Interrupts allow processors to respond immediately to input devices, allowing for a theoretically more fluid user experience. As of yet, no Powder Toy processor supports interrupts.

WiFi channels used: The range of WiFi channels used by the processor, not counting those used to communicate with external devices. This provides an indication of the extensibility of the processor and its compatibility with other devices that use WiFi. Unfortunately this is difficult to measure, thus only an estimation is provided based on random sampling.

LightPC is evidently a strong contender in the Powder Toy information processing field, at least among the chosen processors.

FEATURES

Other than providing reliable, fast and compact computing, LightPC offers many new innovations that not only set standards for modern Powder Toy processors, but also heralds technological progress in a wide range of Powder Toy electronics.

- The use of photon and filter technology allows faster and more compact ALU components, including a logarithmic-time-and-space bitshifter and a two-frame-per-bit adder/subtractor.
- The use of filters in memory storage and computation allows compact 29-bit computing, a field currently only occupied by china-richway2's 32-bit processor.
- The integration of code and data in a large, yet compact 512-word RAM enables complex programs, including, theoretically, recursion, multithreading and operating systems.
- Constant time increments and decrements not only speeds up many programs but also allow for clocks of higher resolution.
- Filter bus technology allows rapid and compact information transfer without the need for WiFi.
- Pistons for spark signal control allows single-frame demultiplexing.
- Particle ray spark signal redirection technology allows for less cluttered, more compact components.
- Register writeback, incrementing the program counter and fetching the next program instruction in parallel increases processor efficiency.
- A maximum instruction width of 29 bits opens up many possibilities for extension.
- Small constants can be specified in instructions in place of register identifiers, reducing the need for RAM access.
- The on and off switches include built-in safety mechanisms to mitigate multiple presses.
- A fast, common interface for external devices allows the processor to be extended with a wide variety of devices, including but not limited to multipliers, binary-to-decimal converters, external memory storage devices, keyboards, screens, numerical displays, control panels, pseudorandom number generators, clocks, routers, microprocessors, filter thermometers, detectors, powered clones, nuclear reactors, robots, pumps, gates and bombs.
- A combination of particle rays and filters partly inspired by drakide allows for a high resolution touchscreen capable of displaying complex images and diagrams.
- Parallel demultiplexing allows instant numerical display.
- Hexadecimal display and touchscreen supports both delayed input and immediate input.
- Assembly language similar to real-life assembly provides a smooth learning curve for programming.

LIMITATIONS

LightPC is not perfect, and is indeed lacking a number of features theoretically possible with Powder Toy processors.

- Very rarely, a Heisenbug occurs. I still have no idea why.
- Increasing the number of registers requires a large amount of space per register, though this could be mitigated with an inverted register writer circuit design.
- Photon technology is physically limited to only support up to 29-bit words.
- LightPC does not support interrupts, resulting in less robust human-computer interaction.
- LightPC does not come with a multiplier by default, thus multiplication is very slow.
- Touchscreen may stop working if buttons are not pressed accurately.
- The LightASM cross-compiler is very poorly optimised, so code written in high-level LightASM may run slowly despite good hardware.

USING LIGHTPC

LOADING PROGRAMS

To load a program into LightPC, follow the steps below:

1. Obtain loadcode.lua from the LightPC forum post.
2. Copy loadcode.lua and the *.mc machine code file into the same directory as your Powder Toy executable.
3. Press "~" in Powder Toy to open the Lua console, type "dofile("loadcode")" (without the outside quotes), hit enter, type the filename of the *.mc file without the ".mc" file extension, hit enter again, the hit enter once more to exit the Lua console.

RUNNING PROGRAMS

If you wish to run a loaded program, follow the steps below. If no program is loaded, load a program first as described in the above section.

1. Spark the "On" button (found at the lower right) to begin the loaded program.
2. If anything goes wrong along the way, spark the "Off" button (represented by a red cross at the lower right) to halt the program.

USING THE HEXADECIMAL INPUT PANEL

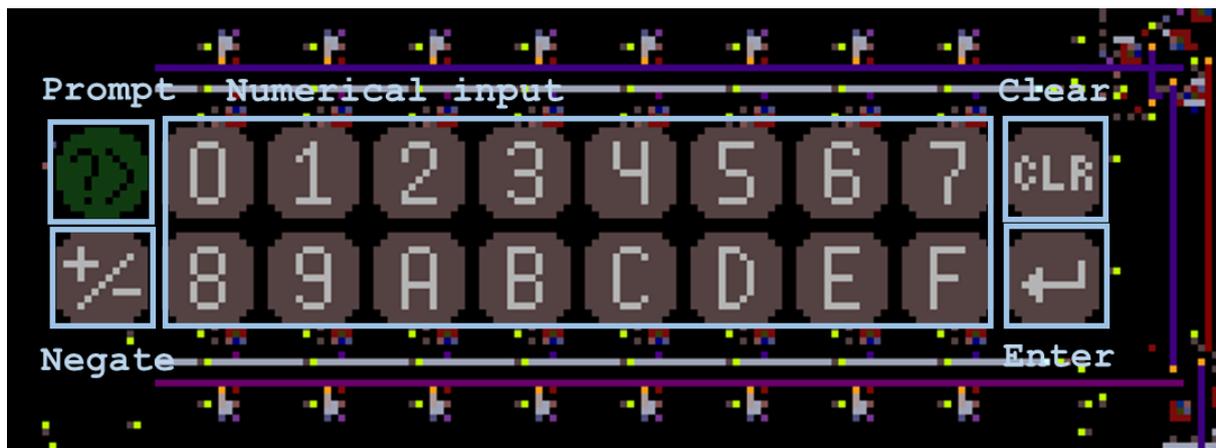


Figure 1 – The various components of the hexadecimal display

To enter a hexadecimal number, press the digits one at a time, starting with the most significant digit. Pressing the **negate** button would negate your input, but note that the program may not support negative input. Pressing the **clear** button would clear the input buffer without saving your input.

Pressing the **enter** button would save your input until the program requests for it, and clear the input buffer. Before the program retrieves your input, you can override your previous input by entering a new number and pressing the **enter** button again.

Some programs may halt temporarily while waiting for your input. If it does so, it would request for your input by lighting up the **prompt** indicator. Enter a number to continue program execution.

USING THE TOUCHSCREEN

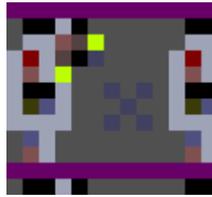


Figure 2 – A single pixel of the touchscreen

Use a small pen size when using the touchscreen. Spark only the metal cross in the centre of each pixel, being careful not to touch the electronics around it. Be especially careful not to spark the particle rays (shown in lime green) or the pixel would no longer respond to input.

Like the hexadecimal display, program execution can halt temporarily while waiting for user input through the touchscreen. The touchscreen has a green prompt indicator at the top left to inform you when the program is waiting for input. Spark a pixel to continue program execution.

PROGRAMMING LIGHTPC

LightPC comes with a cross-compiler for a high-level programming language anachronistically called “LightASM”. The cross-compiler is called “assembler.py” and can be downloaded from the LightPC forum post. If you would like to write machine code directly, skip to the sections “LightPC Machine Code” and “The LightPC Instruction Set”.

THE PROGRAMMING ENVIRONMENT

If you use Windows, the recommended text editor for programming LightPC is Notepad++ due to the availability of syntax highlighting. To enable syntax highlighting, copy “userDefineLang.xml” (obtained from the forum post) into the directory “%APPDATA%\Notepad++”, or integrate the language definitions into an existing copy by copying the “UserLang” node into the “NotepadPlus” node of your existing copy. Once you have done so, save any file under the extension .ac and syntax highlighting would appear.

If you would like to use assembler.py, you will need to download and install Python 3 (<https://www.python.org/>) and Python Lex-Yacc (<http://www.dabeaz.com/ply/>).

If you are using Windows, in order to use Python from the command line, you will need to add Python to your PATH variable. Go to Control Panel > System > Advanced System Settings > Environment Variables, select “Path” in the list of system variables, then click “Edit”. A window would pop out showing a list of directories separated by semicolons. Go to the end of the list, append a semicolon, then add the path of the directory containing python.exe. For me, it was C:\Python34.

LightASM code uses the file extension .ac (which stands for Assembly Code), and is compiled to a file containing hexadecimal numbers with the file extension .mc (which stands for Machine Code). To compile code for LightPC, follow the steps below:

1. Save the code under the file extension .ac in the same folder as assembler.py. That is, if you would like to name your file “helloworld”, save your code as “helloworld.ac”.
2. Run assembler.py by opening the terminal (Command Prompt if you are using Windows) and navigating to the directory containing assembler.py, then entering “assembler.py” (without the quotes).
3. When prompted for the input file, type in the name of your assembly code file, including the file extension.
4. When prompted for the output file, type in the name of the *.mc machine code file including the file extension. If such a file does not already exist, a new file with that name would be created automatically.

assembler.py can optionally take in two arguments, the first being the input file name and the second being the output file name. This allows you to compile the same assembly code file multiple times by pressing the up button in the terminal.

LIGHTASM

THE TWO FLAVOURS OF LIGHTASM

LightASM comes in two flavours – low-level and high-level. Low-level LightASM looks like ordinary assembly code and maps very closely to machine code. High-level LightASM looks like C, featuring automatic register allocation and compound expressions, but unlike modern compilers, it lacks many important optimisations.

Despite being allowed, the two flavours of LightASM should not be used at the same time. If they are, the automatic register allocation may conflict with your own register allocation, leading to unintended behaviour.

While high-level LightASM looks like C, many assumptions that a C programmer would make would be invalid when writing in LightASM. As such, it is recommended that you familiarise yourself with low-level LightASM before trying out high-level LightASM.

HELLO WORLD IN LIGHTASM

Assuming that the screen is connected to port 0, a typical high-level LightASM program that displays “Hello World” on the screen is:

```
define RETREG r14
define screen 0

func dev(devid, arg1, arg2) {
    DEV devid arg1 arg2
    return RETREG
}

func readram(dataid) {
    var res
    LOAD res dataid
    return res
}

data helloworld={
001010111010001000010000b,
001010100010001000101000b,
001110111010001000101000b,
001010100010001000101000b,
001010111011101110010000b,
00000000000000000000000b,
100010010011001000110010b,
100010101010101000101010b,
101010101011001000101010b,
101010101010101000101000b,
010100010010101110110010b
}

var cline=1
var scrcont=helloworld
do{
    @dev(screen, cline, @readram(scrcont))
    cline++
    scrcont++
} while(cline~=12)
```

The same program in low-level LightASM looks less readable, but compiles in 2 less instructions:

```
define screen 0

data helloworld={
001010111010001000010000b,
001010100010001000101000b,
001110111010001000101000b,
001010100010001000101000b,
001010111011101110010000b,
000000000000000000000000b,
100010010011001000110010b,
100010101010101000101010b,
101010101011001000101010b,
101010101010101000101000b,
010100010010101110110010b
}

alias (cline=r0,scrcont=r1,check=r2){
  MOV cline 1
  MOV scrcont helloworld
  do{
    alias (linecontent=r3){
      LOAD linecontent scrcont
      DEV screen cline linecontent
    }
    INC cline
    INC scrcont
    MOV check cline
    XOR check 12
  } while (check)
}
```

LIGHTPC MACHINE CODE

LightPC instructions and data are stored in the ctype of filters, each of which can store 30 bits of data. The first bit is always set as photons cannot have a ctype of zero. While data can span the entirety of the 29 bits, LightPC uses only 25 of the remaining 29 bits for its instructions, leaving the first 4 unused.

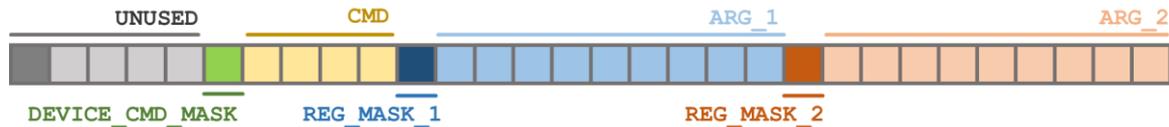


Figure 3 – Bit allocation for a single instruction

LightPC is a Reduced Instruction Set Computer (RISC) system that supports 16 non-device instructions that each take either 0, 1 or 2 arguments. If an instruction takes in only 1 argument, ARG_1 would contain that argument while ARG_2 would be left blank. If an instruction takes in no arguments, both ARG_1 and ARG_2 would be left blank.

When providing constants as arguments, the leftmost bits of ARG_1 and ARG_2 indicate if they are negative numbers. If that bit is set, the integer constant provided to LightPC would have all bits from bit 9 to bit 29 (1-indexed) set. For example, if ARG_1 is set to 0x111110110, it would be parsed as 0x111111111111111111111111111110110. This allows for negative integer constants, effectively setting the range of arguments to [-256, 256).

If an integer constant outside the range of [-256, 256) is required, it should be stored in memory and loaded to a register just before the instruction. The argument provided would then be the register that the integer constant is stored in.

The RAM address for LOAD and STORE instructions and the line number for GNZ and GN instructions are read as ARG&0x3F, so ARG would be in the range [0,512) rather than [-256,256) for those arguments in these instructions.

REG_MASK_1 and REG_MASK_2 indicate if the value for the first and second argument should be read from a register. If REG_MASK is set, the argument would be read from register ARG. If not, the argument would be the value given in ARG. Using integer constants and reading from registers both take the same amount of time.

DEVICE_CMD_MASK indicates if the instruction is a device instruction. If DEVICE_CMD_MASK is set, LightPC would send the two arguments to the device connected to the port identified by CMD. If not, one of the 16 non-device instructions would be executed according to CMD.

The mapping from CMD to the instruction executed is as follows. For what the instructions actually mean, see the next section.

0: SHUTDOWN	4: AND	8: SHL	12: ADD
1: MOV	5: OR	9: SHR	13: SUB
2: LOAD	6: XOR	10: INC	14: GNZ
3: STORE	7: NOTAND	11: DEC	15: GN

THE LIGHTPC INSTRUCTION SET

The 16 non-device instructions are:

SHUTDOWN: Halts the program permanently.

MOV a b: Copies *b* into register *a*.

LOAD a b: Loads the value in index *b* of the RAM into register *a*.

STORE a b: Stores the value in register *b* into index *a* of the RAM.

AND a b: Computes the bitwise AND of *a* and *b* and stores the result in register *a*.

OR a b: Computes the bitwise OR of *a* and *b* and stores the result in register *a*.

XOR a b: Computes the bitwise XOR of *a* and *b* and stores the result in register *a*.

NOTAND a b: Computes the bitwise AND of (NOT *a*) and *b* and stores the result in register *a*.

SHL a b: Shifts *a* leftwards by *b* bits and stores the result in register *a*.

SHR a b: Shifts *a* rightwards by *b* bits and stores the result in register *a*.

INC a: Increments *a* by one and stores the result in register *a*.

DEC a: Decrements *a* by one and stores the result in register *a*.

ADD a b: Adds *a* and *b* and stores the result in register *a*.

SUB a b: Subtracts *b* from *a* and stores the result in register *a*.

GNZ pred line: If *pred* is zero, go to *line*, if not go to the next line.

GN pred line: If *pred* is negative, go to *line*, if not go to the next line.

To communicate with external devices, the *DEV* instruction is used:

DEV portid a b: Sends *a* and *b* to the device connected to port *portid*. If the device produces an output, the output would be written to the last register (r14). If no output is produced, data in r14 would not be overwritten.

Overflow for addition and subtraction instructions are the same as any other computer, looping back from -2^{28} if the number is too large and looping back from $2^{28} - 1$ if the number is too negative.

SPEED OF INSTRUCTIONS

Regardless of instruction, all instructions have a 45 frame overhead to fetch and decode the instruction. The number of frames required for each instruction (including the overhead) is as follows:

SHUTDOWN:	58
MOV:	56
LOAD:	82
STORE:	68
AND:	70
OR:	70
XOR:	69
NOTAND:	69
SHL:	75
SHR:	75
INC:	74
DEC:	74
ADD:	131
SUB:	133
GNZ:	61
GN:	61

The speed of device instructions depend on the devices themselves. There is, however, an overhead for calling devices regardless of what the device actually does. For a device that almost immediately returns the spark signal it receives (writing to the screen), 104 frames were used.

It can be seen that computation instructions require about 70 frames per instruction, and core instructions (MOV, GNZ, GN, SHUTDOWN) require about 60 frames per instruction. The only exceptions are ADD and SUB, which take up about as much time as two instructions. Thus, it is generally favourable to avoid ADD and SUB when possible.

LOW-LEVEL LIGHTASM

COMMENTS

LightASM comments are similar to C comments. Single line comments are prefixed with `/// and multi-line comments are wrapped in /* and */. For example:`

```
//This is a single line comment
```

```
/*This is a  
multi-line comment*/
```

Multi-line comments cannot be nested.

INSTRUCTIONS

Low-level LightASM programs are essentially lists of instructions, separated by whitespace. Registers are zero-indexed and prefixed with the letter `"r"`, that is, the first register is `"r0"` and the second register is `"r1"`.

Integer constants are provided either in decimal, hexadecimal or binary. If the integer is in hexadecimal, it is prefixed with `"0x"`. If the integer is binary, it is suffixed with `"b"`. If the integer is in decimal, no prefixes or suffixes are used. In other words, the number 17 would be `"0x11"` in hexadecimal, `"10001b"` in binary and `"17"` in decimal. Front padding with zeroes is allowed.

Line numbers are zero-indexed. For example, to go to the start of the program, the instruction `"GNZ 1 0"` can be used. For such purposes, however, LightASM provides an additional GOTO instruction, in the format `"GOTO lineno"`, which is compiled directly to `"GNZ 1 lineno"`. Thus, `"GOTO 0"` can be used to go to the start of the program instead.

All LightASM programs, high-level or low-level, should end with the instruction `"SHUTDOWN"` to terminate program execution. If not present, the compiler would add one for you.

At this stage, without using any code structures, the Hello World program (assuming that the array to be displayed is stored in RAM index 10 and that the screen is connected to port 0) would look like this:

```
MOV r0 1
MOV r1 10
LOAD r3 r1
DEV 0 r0 r3
INC r0
INC r1
MOV r2 r0
XOR r2 12
GNZ r2 2
```

The first two lines sets registers r0 and r1 to 1 and 10 respectively, where r0 stores the line on the screen that the array should be drawn to, and r1 stores the index of the array that should be drawn to that line.

The third line loads into r3 the line to be drawn from the RAM as specified by r1. r3 is subsequently drawn onto the screen at line r0 with a DEV instruction.

The fifth and sixth lines increment r0 and r1 in order to move to the next line.

The last three lines checks if r0 is equal to 12 by performing an XOR. If r0 is not 12, the XOR would return zero and the GNZ instruction would direct code execution to the third line (line 2). If r0 is 12, the entire array has been printed and the program terminates.

IDENTIFIERS

Identifiers in LightASM can contain underscores, lowercase letters, uppercase letters and digits. Identifiers should not start with digits (or they would be mistaken for numbers), and should not be of the form “r” followed by only digits (or they would be mistaken for registers). They should also not conflict with the following keywords:

```
alias define data do while whilen ifz ifnn if else regstore break continue
func var return SHUTDOWN AND OR XOR NOTAND SHL SHR ADD SUB INC DEC DEV GNZ
GN GOTO MOV LOAD STORE
```

STATIC DATA

The Hello World program would not work without static data, since that is where the image containing the text “Hello World” is stored. Static data is declared with the “data” keyword to instruct the compiler to allocate RAM space for it. All static data declarations must be provided with an identifier used as a pointer to the data and a value to initialise it with. Initialisation does not require extra runtime, so if you do not need to initialise the data in the RAM space, provide a zero for the initial value.

Two types of static data can be declared. To declare an integer, use the following format:

```
data myint=5000
```

Here, “myint” is the name of the data pointer and 5000 is the initial value stored in the corresponding RAM space. “myint” can then be used in LOAD instructions as so:

```
LOAD r0 myint
```

To declare an array, replace the initial value, that is, the 5000 in the above example, with a list of integers separated by commas and wrapped in curly braces. For example, the following is the image array for the Hello World program:

```
data helloworld={  
001010111010001000010000b,  
001010100010001000101000b,  
001110111010001000101000b,  
001010100010001000101000b,  
001010111011101110010000b,  
000000000000000000000000b,  
100010010011001000110010b,  
100010101010101000101010b,  
101010101011001000101010b,  
101010101010101000101000b,  
010100010010101110110010b  
}
```

Note that binary and hexadecimal numbers can be used in static data declarations as well.

The data pointer “helloworld” now points to the first element in the array. The first element can thus be obtained as follows:

```
LOAD r0 helloworld
```

To access individual elements of the array, square brackets can be used. For instance, to load the second line of the array, the following instruction can be used:

```
LOAD r0 helloworld[1]
```

This shorthand only works with integer constants, however. If you would like to access an arbitrary element of the array, precede the LOAD instruction with an ADD instruction. For example, the following loads the element of the helloworld array with index provided in r1:

```
MOV r2 helloworld  
ADD r2 r1  
LOAD r0 r2
```

If you are looping through an array, it is recommended that you use INC instructions rather than an ADD since INC instructions are much faster than ADDs.

CODE LABELS

To facilitate GNZ and GN instructions, code labels should be used. Code labels mark out positions in the code that a GNZ or GN instruction should jump to. They are prefixed with a colon during declaration but not during usage. For instance, the Hello World program (with the static data omitted) can be written as follows:

```
data helloworld={
  /*data omitted for clarity*/
}

MOV r0 1
MOV r1 helloworld
:loopstart
LOAD r3 r1
DEV 0 r0 r3
INC r0
INC r1
MOV r2 r0
XOR r2 12
GNZ r2 loopstart
```

STATEMENT LISTS

LightASM allows the definition of statement lists, that is, lists of statements wrapped in curly braces. They can be defined anywhere in the code to visually separate distinct sections of the code. Specifically, they are defined as follows:

```
{
    statement_1
    statement_2
    ...
    statement_n
}
```

DEFINE AND ALIAS

To increase code readability, the “define” instruction can be used to create an alias for integer constants and registers. The alias is given first, and the actual value is given second. For example, the Hello World program can be written as follows:

```
data helloworld={  
  /*data omitted for clarity*/  
}  
  
define SCREEN 0  
define line r0  
define arrayline r1  
define check r2  
define displine r3  
define STARTY 1  
define ENDY 12  
  
MOV line STARTY  
MOV arrayline helloworld  
:loopstart  
LOAD displine arrayline  
DEV SCREEN line displine  
INC line  
INC arrayline  
MOV check line  
XOR check ENDY  
GNZ check loopstart
```

Unlike C #define pre-processor directives, define statements cannot store expressions. Furthermore, each alias can only be defined once.

For registers, the “alias” structure can be used in place of “define”. The advantage of using alias structures is that it provides some code structure and scoping, allowing the same alias to be used at multiple points in the code. Alias statements follow the following format:

```
alias (alias_1=reg_1,alias_2=reg_2,...,alias_n=reg_n) statement
```

The statement at the end can be a single statement or a statement list.

For example, the Hello World program can be written as such:

```
data helloworld={
/*data omitted for clarity*/
}

define SCREEN 0
define STARTY 1
define ENDY 12

alias (line=r0,arrayline=r1,check=r2,displine=r3) {
    MOV line STARTY
    MOV arrayline helloworld
    :loopstart
    LOAD displine arrayline
    DEV SCREEN line displine
    INC line
    INC arrayline
    MOV check line
    XOR check ENDY
    GNZ check loopstart
}
```

IF STATEMENTS

Low-level LightASM provides some shorthand structures. One of them is the if statement. If statements are defined as follows:

```
if_instruction(pred) statement_if else statement_else
```

There are three types of if statements, identified by the string provided in `if_instruction`. If `if_instruction` is "ifz" (if `pred` is zero), the if statement is compiled to:

```
GNZ pred else_begin
    statement_if
    GOTO ifelse_end
:else_begin
    statement_else
:ifelse_end
```

If `if_instruction` is "ifnn" (if `pred` is non-negative), the if statement is compiled to:

```
GN pred else_begin
    statement_if
    GOTO ifelse_end
:else_begin
    statement_else
:ifelse_end
```

If `if_instruction` is just "if" (if `pred` is non-zero), the if statement is compiled to an ifz statement, but with the if and else blocks swapped around.

For each of the instructions, the “else” block can be omitted. If so, ifz instructions would be compiled to:

```
GNZ pred if_end
      statement_if
:if_end
```

Likewise, ifnn statements would be compiled to:

```
GN pred if_end
      statement_if
:if_end
```

If statements would be compiled to ifz-else statements but with `statement_if` provided as the else statement and an empty statement list provided as the if statement.

As can be seen, when no else block is provided, ifz and ifnn are faster than if.

When using low-level LightASM, the predicate used for if statements should be a register or integer constant. If an expression is provided, the code may still compile, but automatic register allocation would be activated and that may conflict with your own register allocation.

WHILE LOOPS

Low-level LightASM supports while loops. They are defined as follows:

```
do statement while_instruction(pred)
```

The “do” structure draws a parallel to C, where the statement is executed for the first time even if `pred` is not satisfied. LightASM does not support true while loops, where the statement is executed for the first time only if `pred` is satisfied. If you would like to use a true while loop, wrap your while loop in an if statement as follows:

```
if(pred) do statement while_instruction(pred)
```

There are two types of while loops, identified by the string given for `while_instruction`. If `while_instruction` is “while” (while `pred` is non-zero), the code would be compiled to:

```
:while_begin
      statement
GNZ pred while_begin
:while_end
```

If `while_instruction` is “whilen” (while `pred` is negative), the code would be compiled to:

```
:while_begin
      statement
GN pred while_begin
:while_end
```

Infinite loops can be defined using “while” as `while_instruction` and using “1” as a predicate.

Within a while loop, the “break” and “continue” statements can be used. Break statements are defined as follows:

break number_of_loops

The number_of_loops argument indicates the number of loops to break out of. For example, if we write:

```
do{
    do{
        break 1
        break 2
    } while(1)
} while(1)
```

The code compiles to:

```
:outer_while_begin
:inner_while_begin
    GNZ 1 inner_while_end
    GNZ 1 outer_while_end
    GNZ 1 inner_while_begin
:inner_while_end
GNZ 1 outer_while_begin
:outer_while_end
```

Continue statements are defined as follows:

continue number_of_loops

They are similar to break statements, but goes to the start, rather than the end of the given loop. For instance, if we write:

```
do{
    do{
        continue 1
        continue 2
    } while(1)
} while(1)
```

The code compiles to:

```
:outer_while_begin
:inner_while_begin
    GNZ 1 inner_while_begin
    GNZ 1 outer_while_begin
    GNZ 1 inner_while_begin
:inner_while_end
GNZ 1 outer_while_begin
:outer_while_end
```

When using low-level LightASM, the predicate used for while statements should be a register or integer constant. If an expression is provided, the code may still compile, but automatic register allocation would be activated and that may conflict with your own register allocation.

TEMPORARILY STORING REGISTERS

For large programs, more than 15 registers may be required. If so, some registers would need to be spilled to RAM. There are two ways of doing this.

Firstly, a register could be made to “live” in the RAM. Before every instruction where the value of the register is read, a LOAD instruction would be used to load the value of the register into a temporary register, and after every instruction where the register is written to, a STORE instruction would be used to store the value back to memory. This is the technique used in high-level LightASM during automatic register allocation.

Secondly, if a register is only used before and after a chunk of code, the register could be stored temporarily in memory before executing the chunk of code, then retrieved after the chunk of code is executed. This is facilitated by the “regstore” statement, defined as follows:

regstore (reg_1, reg_2, ..., reg_n) statement

The regstore statement allocates some space in RAM for the registers provided as arguments, prepends STORE instructions for all the registers before the statement, then appends LOAD instructions for all the registers after the statement. For example, if we write:

```
regstore (r0, r1) {  
    statement  
}
```

This would be compiled to:

```
data r0_temp=0  
data r1_temp=0  
STORE r0_temp r0  
STORE r1_temp r1  
    statement  
LOAD r1 r1_temp  
LOAD r0 r0_temp
```

HIGH-LEVEL LIGHTASM

VARIABLES

High-level LightASM allows the definition of variables to be automatically allocated to registers. Variables are defined as follows:

var id_1[=init_1], id_2[=init_2], ..., id_n[=init_n]

The identifier id can then be used later to refer to the register allocated to it. Variables can optionally be initialised by appending “=” and the value to initialise it with after the identifier. This would be compiled into a MOV instruction.

For example, if we write:

```
var a,b=0,c
MOV a 3
SHL b c
```

The compiler may allocate register r1 to a, register r0 to b and register r2 to c. Then, the code would be compiled to:

```
MOV r0 0
MOV r1 3
SHL r0 r2
```

Variables exist only in the scope of the statement list they are defined in. For example, the following would not compile:

```
{
    var a
}
MOV a 1
```

REGISTER ALLOCATION

The compiler only allocates 12 registers to variables. r12 and r13 are reserved as temporary registers for spilled variables, while r14 is reserved for collecting the output of external devices.

The compiler would attempt to allocate the most used variables into the same registers, while preventing register conflicts by not allocating variables with conflicting scopes together. The usage counts of registers within while loops are multiplied by 100 every nested while loop during allocation.

When the compiler runs out of registers to allocate, some variables would be spilled into RAM. A space in RAM is allocated for that register, then LOAD instructions are prepended before every instruction that uses that register and STORE instructions are appended after every instruction that writes to that register.

Due to the lack of compiler optimisation, it is recommended that you declare variables at the innermost scope possible to reduce variable spilling. Wrapping independent parts of code with braces is another important optimisation as it would aid in register allocation.

EXPRESSIONS

LightASM supports compound expressions with rudimentary optimisations. The expressions supported are:

- a = b:** Sets variable *a* to *b*, and returns *a*.
- a += b:** Sets *a* to *a + b* and returns *a*.
- a -= b:** Sets *a* to *a - b* and returns *a*.
- a &= b:** Sets *a* to *a AND b* and returns *a*.
- a |= b:** Sets *a* to *a OR b* and returns *a*.
- a ^= b:** Sets *a* to *a XOR b* and returns *a*.
- a ~&= b:** Sets *a* to *(NOT a) AND b* and returns *a*.
- a <<= b:** Sets *a* to *a << b* and returns *a*.
- a >>= b:** Sets *a* to *a >> b* and returns *a*.
- a++:** Increments *a* and returns *a*.
- a--:** Decrements *a* and returns *a*.
- a:** Returns the negative of *a* without modifying *a*.
- a + b:** Returns *a + b* without modifying *a* or *b*.
- a - b:** Returns *a - b* without modifying *a* or *b*.
- a & b:** Returns *a AND b* without modifying *a* or *b*.
- a | b:** Returns *a OR b* without modifying *a* or *b*.
- a ^ b:** Returns *a XOR b* without modifying *a* or *b*.
- a ~& b:** Returns *(NOT a) AND b* without modifying *a* or *b*.
- a << b:** Returns *a << b* without modifying *a* or *b*.
- a >> b:** Returns *a >> b* without modifying *a* or *b*.
- ~a:** Returns NOT *a* without modifying *a*.
- a == b:** Returns a non-zero value if *a* is equal to *b* and zero otherwise.
- a ~= b:** Returns a non-zero value if *a* is not equal to *b* and zero otherwise.
- a < b:** Returns a non-zero value if *a* is less than *b* and zero otherwise.
- a > b:** Returns a non-zero value if *a* is more than *b* and zero otherwise.
- a <= b:** Returns a non-zero value if *a* is less than or equal to *b* and zero otherwise.
- a >= b:** Returns a non-zero value if *a* is more than or equal to *b* and zero otherwise.
- a && b:** Returns a non-zero value if *a* and *b* are non-zero and zero otherwise. Will not evaluate *b* if *a* is zero.
- a || b:** Returns a non-zero value if *a* or *b* is non-zero and zero otherwise. Will not evaluate *b* if *a* is non-zero.
- !a:** Returns a non-zero value if *a* is zero and zero otherwise.

The precedence rules for the operators are different from languages like C. In particular, bitwise operators are given priority over addition and subtraction operators. The precedence order, starting from the highest precedence, is:

Bitwise NOT (right associative):	~
Bitwise operations (left associative):	& ^ ~& << >>
Add and subtract (left associative):	+ -
Modification operations (left associative):	= += -= &= = ^= ~&= <<= >>=
Increment and decrement (left associative):	++ --
Comparison operations (not associative):	< > <= >= == ~=
Boolean NOT (right associative):	!
Boolean AND (left associative):	&&
Boolean OR (left associative):	

Expressions can be wrapped in parenthesis to override precedence rules. Expressions cannot be used as arguments to instructions, but can exist on their own as statements, as predicates for if statements and while loops and as arguments to function calls.

When using expressions, do not use registers. This is because most expressions are compiled to variables, which are automatically allocated to registers, which may conflict with your own register allocation.

Integer constants, variable identifiers, array accesses and function calls can be used as operands for expressions. Note, however, that array accesses are treated as pointers pointing to data, not the data itself. In order to obtain the data pointed to by an array access, a LOAD instruction is required. Similarly, to write to a RAM address indicated by an array access, a STORE instruction is required.

FUNCTIONS

Until now, programs can be written almost completely with expressions, except for LOAD, STORE and DEV instructions. To do away with these, functions can be used.

Functions are defined as follows:

```
func func_id(arg_1,arg_2,...,arg_n) statement
```

“func_id” is the identifier used to call the function with, the “arg”s are the arguments to the functions, and “statement” is the function body.

Functions can return values with return statements, defined as follows:

```
return expression
```

This statement would return the result of “expression” as the result of a function call.

Functions calls are defined as follows:

```
@func_id(arg_1,arg_2,...,arg_n)
```

Here, “func_id” is the identifier of the function to be called, and the “arg”s are the arguments to be passed to the function.

LightASM functions are more akin to C inline functions than C functions. When a function call is used, the entire function body is copied into that location, all function arguments in the function body are replaced with the arguments provided in the function call and all return statements are replaced with MOV instructions to the temporary variable that the function call returns.

One of the results of such an implementation is that recursion is not allowed. Another result is that any modification to arguments inside the function call would affect code outside the function call. If you would like to modify arguments in a function without affecting the arguments passed into the function themselves, create new variables in the function body and assign the arguments to them.

For example, the following function allows device instructions to be made in expressions:

```
func dev(devid,arg1,arg2) {  
    DEV devid arg1 arg2  
    return r14  
}
```

We can then write:

```
var retval=@dev(devid,arg1,arg2)
```

This would be compiled to:

```
var retval
  DEV devid arg1 arg2
  retval=r14
  GOTO func_end
:func_end
```

The following are the three most useful functions in LightASM. They are used to replace LOAD, STORE and DEV instructions, allowing LightASM code to be written purely without instructions.

```
func dev(devid,arg1,arg2) {
  DEV devid arg1 arg2
  return r14
}

func writeram(dataid,val) STORE dataid val

func readram(dataid) {
  var res
  LOAD res dataid
  return res
}
```

COMMUNICATING WITH THE DEFAULT EXTERNAL DEVICES

THE TOUCHSCREEN

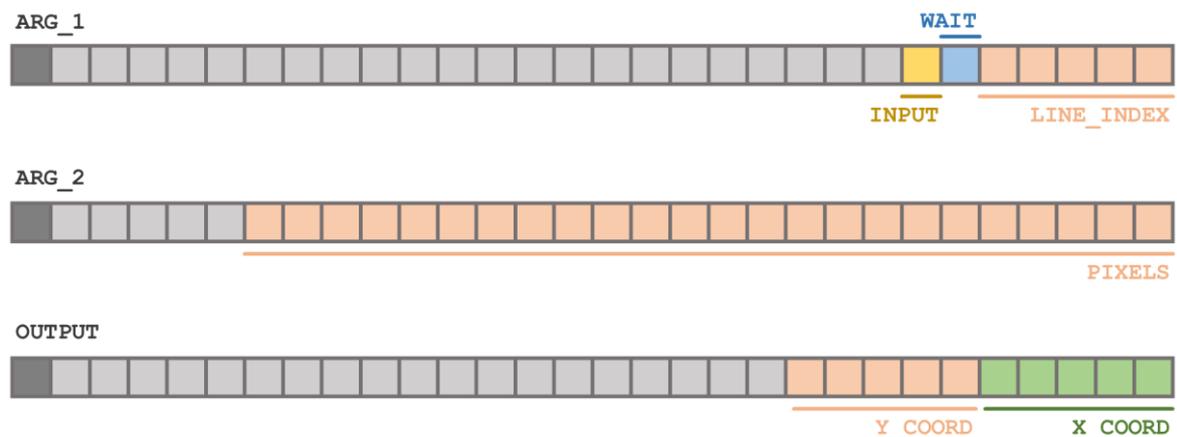


Figure 4 – Bit allocation for touchscreen I/O

By default, the touchscreen is connected to device port 0.

To read user input from the touchscreen, the INPUT bit should be set. If the WAIT bit is set, the program will pause temporarily until it receives user input (unless the user has already touched a pixel before that command). The return value would contain the y-coordinate of the user's input in Y_COORD and the x-coordinate of the user's input in X_COORD. The y-coordinate is zero-indexed and measured from the top, and the x-coordinate is zero-indexed and measured from the right. If the WAIT bit is not set and the user has not given any input when the touchscreen is polled, the value in ARG_2 would be returned.

To write to the screen, both the INPUT and WAIT bits should be unset. The 5-bit LINE_INDEX should contain the index of the line to write to, zero-indexed from the topmost row, and the PIXELS bits should contain the pixel configuration to write to that row, with a set bit indicating that the corresponding pixel should be lit and an unset bit indicating that the corresponding pixel should be unlit. The rightmost bit corresponds to the rightmost pixel on the screen.

THE HEXADECIMAL DISPLAY

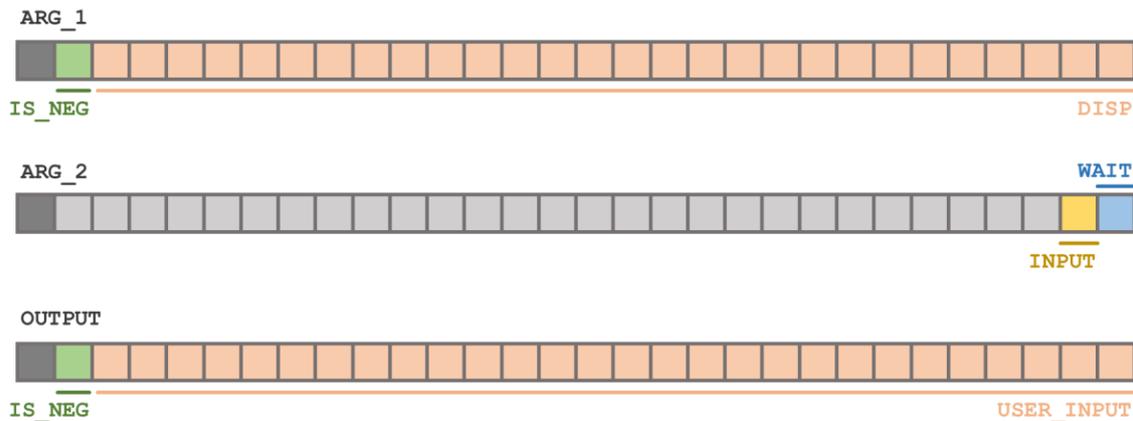


Figure 5 – Bit allocation for hexadecimal display I/O

By default, the hexadecimal display is connected to device port 1.

To read user input from the display, the INPUT bit should be set. If the WAIT bit is set, the program will pause temporarily until it receives user input (unless the user has already entered a number before that command). The return value would contain the absolute value of the user's input in USER_INPUT and whether the user's input is negative in IS_NEG. If the WAIT bit is not set and the user has not given any input when the display is polled, the value in ARG_1 would be returned.

Note that USER_INPUT is given in the wrong format for negative numbers. If you would like your program to support negative input, the negation must be done in software.

To write to the display, both the INPUT and WAIT bits should be unset. The DISP bits should contain the absolute value of the number to be displayed, and the IS_NEG bit should be set if the number to be displayed is negative. Again, DISP is in the wrong format for negative numbers, and the negation must be performed in software if the program is to output negative numbers.

While the display is hexadecimal, binary to decimal conversion can be done in software (or with the help of another external device) if decimal numbers are to be displayed.

THE PSEUDORANDOM NUMBER GENERATOR AND CLOCK

By default, the pseudorandom number generator is connected to port 2 and the clock is connected to port 3.

The pseudorandom number generator operates in $O(n)$ time and generates a number with every bit, including the negative bit, randomised. The clock increments a number every 60 frames (though the rate of increment can be easily changed in hardware) and produces that number when instructed to.

For both the pseudorandom number generator and the clock, the value stored in r14 when called is $(\text{OUTPUT} \& \text{ARG}_1) | \text{ARG}_2$ where OUTPUT is the raw output from the device and ARG_1 and ARG_2 are the arguments sent to the device.

DEBUGGING

ERRORS PRODUCED BY THE CROSS-COMPILER

assembler.py produces 4 types of errors: warnings, errors, critical errors and Python errors.

Warnings are errors that the compiler can still recover from and produce meaningful machine code output. However, they are not safe to ignore as warnings usually indicate programming mistakes in LightASM code.

Errors are errors that the compiler cannot recover from, such as broken instructions or unclosed braces.

Critical errors indicate errors that the compiler should have detected at an earlier stage but failed to do so, or bugs in assembler.py in general.

Python errors indicate bugs in assembler.py.

If you encounter warnings or errors, it is likely that there is a bug in your LightASM code, but you cannot exclude the possibility of bugs in assembler.py. If you encounter critical errors or python errors, there are definitely bugs in assembler.py and you will have to either fix them yourself (see the section “The Cross-Compiler”) or report them.

UNINTENDED OUTPUT

If LightPC does not produce the output you expect, there are three possible points where failure could have occurred: your LightASM code, the cross-compiler and LightPC hardware. Due to extensive testing, errors in LightPC hardware are very unlikely, but still possible. While there is no magic potion to find the source of error quickly, there are a few ways you can narrow your search.

If your program runs fast enough, try running it again to see if the same wrong output is produced. If so, it is likely that you have encountered a Heisenbug in LightPC hardware arising from timing issues. Powder Toy processes particles in a rather unpredictable order, causing errors to be produced at some times and not others. Such bugs are very rare, but if they do occur, there is unfortunately no easy way to fix them since they are very hard to produce in the first place.

simulator.cpp (or, if you are using it precompiled, simulator.exe) is a simulation of LightPC in C++. Type in the name of the machine code file, including the file extension, to simulate a run. If the simulator produces an output different from what is actually produced in LightPC, it could be due to a bug in the simulation or a bug in LightPC hardware.

A very useful tool to check for bugs in your code or in assembler.py itself is the file assembly_intermediate.ac, which assembler.py produces every assembly. It shows instructions exactly as they would be written to LightPC, except in a more readable format than the .mc file. Pausing LightPC in powder toy during a fetch phase and obtaining the x-coordinate of the RAM being read by looking at the HUD in debug mode (press “D” to enable debug mode), then subtracting 68 (the x-coordinate of the first RAM space in Powder Toy) would give you the line number currently being processed. You can then find out where in the code the program is currently executing and map its every action back to assembly_intermediate.ac.

Note that line numbers in most editors are one-indexed, so you will have to add one to LightPC line numbers to map them back to code in assembly_intermediate.ac.

Another way to debug LightPC is to read the ctype of registers directly. The registers can be found at the bottom left of LightPC and are ordered from left to right in increasing order. Usually, it is difficult to read the ctype of filters directly. In this case, the spectrometer provided just below the touchscreen would be useful. Replace the filter particle in front of the photon emitter with the filter particle you would like to analyse, then spark the INWR. The photons produced would be the set bits of the filter, least significant bit leftmost.

THE CROSS-COMPILER

assembler.py is built around Python Lex-Yacc, a clone of lexer generator Lex and LALR parser generator Yacc in Python. Cross-compilation follows 6 phases as follows:

1. Raw LightASM code is lexed into tokens by Lex, and these tokens are parsed by Yacc to form an Abstract Syntax Tree (AST). The nodes of the tree are named tuples, all defined at the beginning of assembler.py.
2. (preprocess) The AST is preprocessed with a depth-first search to extract and process data and define statements as well as to determine which identifiers belong to code labels. Functions are extracted but not parsed at this stage.
3. (replaceidentifiers, simplifyexpression, expandexpression, replacearguments) A depth-first search is conducted on the AST, this time keeping track of alias and variable identifiers and their scope. Expressions, if statements, while loops, restore statements and alias statements are converted into lists of instructions at this stage. Function calls are also replaced by their corresponding function bodies. All variables are given unique names, so two different variables with the same name originally would now be identified by distinct identifiers. The number of times that each variable is used is counted during this stage. The output produced at this stage should consist only of statement lists, code labels, variable declarations and instructions.
4. (allocateregisters) A depth-first search is conducted on the AST to allocate registers or, in the case of spilling, RAM space to variables.
5. (unravelstructure) A depth-first search is conducted on the AST to produce a list of statements. Variables are translated to registers or, in the case of spilling, LOAD and STORE instructions at this stage. The output at this stage should be a statement list containing exactly the number of instructions that would be written to LightPC.
6. (emitstatement) Instructions are translated to machine code, instruction by instruction. Code labels and data pointers passed as arguments to instructions are converted to numbers at this stage. The .mc file is written to at this stage.

The precise details of each functions are left as an exercise to the reader. The above information, along with the PLY reference (<http://www.dabeaz.com/ply/ply.html>), should be sufficient to isolate bugs in assembler.py.

CIRCUIT DESIGN

LIGHTPC CIRCUIT IDIOMS

LOGIC GATES

LightPC makes heavy use of simple logic gates for control flow management. When a wire is to be sparked for more than one reason, an OR gate is used. This is implemented by pointing two spark particle rays to the same wire.

When a wire is to be sparked only when two conditions are to be fulfilled, an AND gate is used. This is implemented by a piston. The piston would begin extended, blocking the path of the second input, which is to be sent by a single-spark particle ray emitter. When the first input arrives, it retracts the piston, allowing the second input to pass through and spark its destination.

NOTAND gates are implemented in the same way as AND gates, but with the piston initially retracted and extended only by the first input.

SELF-RESETTING NOTAND GATES

Most NOTAND gates in LightPC are self-resetting, meaning that they do not need another spark to reset. This is achieved by adding a delay between the piston retractor and the piston extender.

OLD SPARK DELAYS

At times, delays required are too short to use a delay particle. In such cases, the old way of producing delays, that is, with wire particles spaced one particle apart, is used.

PHOTON EMITTERS

Photon emitters are created by surrounding a photon powered clone by voids or insulators. A p-type semiconductor particle and an n-type semiconductor particle are placed in its vicinity. A single photon can then be produced by sparking the p-type semiconductor particle. In order for only a single photon to be produced, the n-type semiconductor particle must be on the right or below the p-type semiconductor particle. This is due to the order in which Powder Toy processes particles.

The colour of the photon produced is set by placing a filter particle three pixels away from the powered clone. The colour of the filter itself can be set by a detector.

PHOTON MULTIPLIERS

Many photons in a row of the same colour can be produced by lining the required number of photon emitters in a row and placing a line of filter particles three pixels away in the direction of emission. A spark particle ray is used to activate all emitters at once.

PISTON DEMULTIPLEXERS

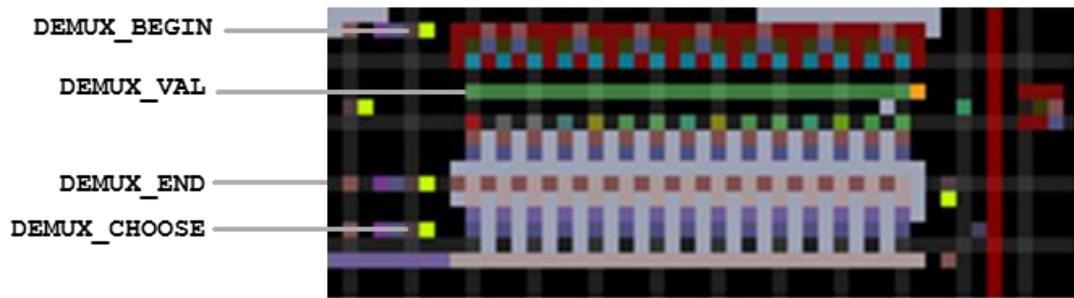


Figure 6 – An example of a piston demultiplexer

Piston demultiplexers consist of four layers: a photon multiplier, XOR filters of different colours, photon detectors and a row of NOTAND gates. They take in four inputs: DEMUX_BEGIN, DEMUX_VAL, DEMUX_CHOOSE and DEMUX_END.

DEMUX_VAL holds the identifier of the wire that the spark signal should be directed to. DEMUX_BEGIN begins the demultiplexing process, and DEMUX_CHOOSE sends the actual spark signal to be directed to the wire corresponding to DEMUX_VAL. DEMUX_END resets the demultiplexer.

FILTER BUSES

Long lines of filter particles are used as buses in many parts of LightPC. A bus can be read from anywhere along it by sending a photon through it. To write to the bus in the middle of it, a piston is used to push a detector into the bus to replace a filter particle, then to pull the filter particle back to replace the detector so that the bus remains connected.

MULTI-INPUT SPARK BUSES

Spark buses are usually slow as they require sparks at one end of the bus to pass through every transmitter in the middle before reaching the other end of the bus. Multi-input spark buses mitigates this by using a battery particle ray in a line of insulator particles. Sparking the particle ray would create a battery at the end of the bus, which would then activate a signal to destroy the battery with another particle ray and to transmit the spark signal. The time required to transmit a spark signal in this manner does not depend on the number of transmitters.

COMPONENTS

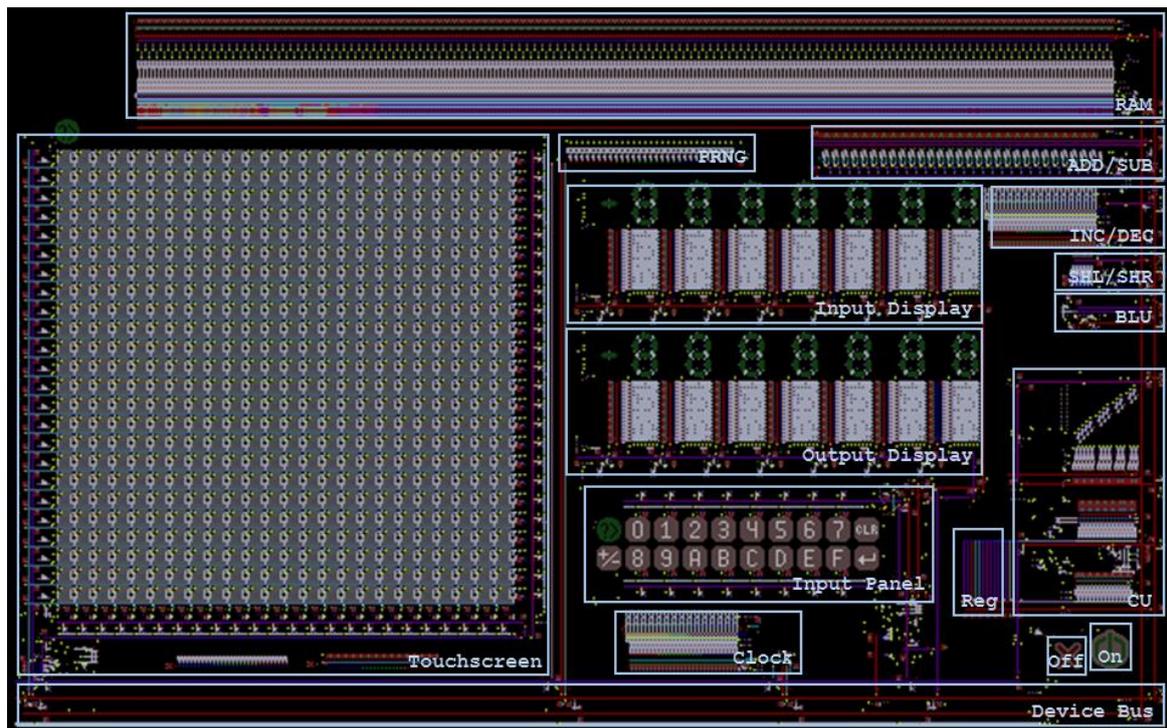


Figure 7 – The various default components of LightPC

LightPC can be divided into 3 main sections – the Central Processing Unit, comprising the Arithmetic/Logic Unit (ALU), Control Unit (CU) and the registers (Reg), the Random Access Memory (RAM) and the external devices.

Below are descriptions of each component marked out in Figure 7:

RAM: The Random Access Memory is responsible for storing and retrieving code and data. A number x is stored at index i by setting the colour of the i th filter from the left to x . The filter corresponding to index i is singled out by a demultiplexer and manipulated by the circuitry on the right.

ADD/SUB: The adder/subtractor computes additions and subtractions at a rate of 2 frames per bit. It comprises 29 half-adders and works by performing XORs and ANDs in parallel, performing a simultaneous carry and OR then performing a final XOR. Subtraction is performed by taking the NOT of the second input and supplying a starting carry, effectively finding the two's complement, before performing addition.

INC/DEC: The incremator/decremator increments or decrements an integer in $O(1)$ time by using a particle ray to single out the rightmost unset bit, then flipping it and all the bits to the right of it. An integer is decremented by performing a NOT at the start and the end of the increment, effectively finding the two's complement twice.

SHL/SHR: The bitshifter performs bitshifting in $O(\log n)$ time by using redshift and blueshift filters set to shift bits by powers of two.

BLU: The Boolean Logic Unit performs Boolean operations in $O(1)$ time by sending photons through filters with in-built Boolean logic capabilities.

Reg: The processor contains 15 registers, which take the form of filter columns. One of them stores the output from external devices.

CU: The control unit includes an instruction demultiplexer at the top, a register reader in the middle and a register writer at the bottom, all of which are fundamentally filter demultiplexers. The control unit also hosts the core circuitry to manage control flow.

On: The on switch switches the processor on and begins processing from line 0.

Off: The off switch halts the program upon completion of the current instruction.

Device Bus: The device bus sends and receives signals from external devices.

Touchscreen: The touchscreen is an external device that allows $O(1)$ updates of single lines of pixels and records the Cartesian coordinates of the last click made by the user.

PRNG: The pseudorandom number generator is an external device that generates a random 29-bit number in $O(n)$ time by using mercury as a per-bit randomiser.

Input Display: The input display uses parallel demultiplexers to display user input through the control panel and is updated every keypress.

Output Display: The output display displays an arbitrary hexadecimal number as instructed by the processor.

Input Panel: The input panel records per-digit input from the user and stores and updates the desired input integer.

Clock: The clock is an incrementor that increments an integer every 60 frames, the rate of which can be adjusted. It allows in-game time measurements that provide frame-rate-independent time measurements for processor benchmarking.

THE INSTRUCTION PIPELINE

LightPC's instruction pipeline follows a four-phase fetch-decode-execute-writeback cycle, with the fetch phase performed in parallel with the writeback phase. The specifics of each phase are:

Fetch: In parallel, the next instruction is fetched from the RAM and stored in the instruction register, and the program counter is incremented.

Decode: In parallel, the instruction is sent through a demultiplexer to direct a spark signal to the corresponding component and the two operands (that could either be a register address or integer constant) are converted to integers and sent into the operand bus.

Execute: The component corresponding to the current instruction is activated. It processes the operands from the operand bus and may return an output through the same bus. If the instruction is a jump instruction, the program counter is updated. Memory access is also performed at this phase.

Writeback: If the instruction requires, the value in the operand bus is written into the register corresponding to the first operand at the same time as the fetch phase of the next instruction.

BUSES



Figure 8 – The two filter columns of LightPC

At the right of LightPC are two columns of filter particle. The one on the left is the instruction register, which holds the instruction currently being executed. The column on the right is a two-way bus used to transport arguments to computation components as well as the return value from those components.

THE RAM

A large portion of the Random Access Memory is a demultiplexer compressed so that each word of memory requires only a one-pixel column. Below the demultiplexer is a row of photon emitters, a series of filter buses, detectors, and the filters storing the memory themselves.

The 7 layers of filters are used in the following manner:

Layer 1: Pads empty space.

Layer 2: Pads empty space.

Layer 3: Stores the data.

Layer 4: Acts as a bus to write to memory.

Layer 5: Acts as a bus to read from memory.

Layer 6: Pads empty space.

Layer 7: Pads empty space.

It can be seen that LightPC can be made slightly faster by reducing the size of the RAM so that each word could take up more space, reducing the distance a photon has to travel during a LOAD.

THE ARITHMETIC LOGIC UNIT

THE ADDER-SUBTRACTOR

The adder-subtractor is a circuit implementation of the following algorithm:

$$X[i] = A[i] \text{ XOR } B[i]$$

$$Y[i] = A[i] \text{ AND } B[i]$$

$$\text{CARRY}[i] = (X[i] \text{ AND } \text{CARRY}[i-1]) \text{ OR } Y[i]$$

$$\text{RES}[i] = (A[i] \text{ XOR } B[i]) \text{ XOR } \text{CARRY}[i-1]$$

The adder-subtractor has three phases. The first phase calculates X and Y in constant time. X is stored in the switches and Y is stored in the pistons. The second phase, which involve a spark signal and a photon moving leftwards simultaneously, calculates CARRY in linear time. CARRY is stored in the pistons, and is compiled into a single photon by a series of ORs. The third phase calculates RES by sending a photon through two XOR filters.

To subtract one number from another, we need to find the two's complement of the second number by taking the NOT of it and incrementing the result. This is achieved with an XOR filter to perform the NOT (to prevent the first bit from being unset) and supplying an initial carry to perform the increment.

THE INCREMENTOR-DECREMENTOR

The incrementor-decrementor exists because incrementing and decrementing a number is much faster than adding one to or subtracting one from it with the adder-subtractor.

Incrementing a number is the same as finding the least significant zero, then flipping all the bits to the right of it including the zero itself. This is achieved in circuit by using a demultiplexer to select the first zero, which then, at the same time, activates a photon emitter and pushes a detector into the filter bus to obtain a filter colour with all bits to the right of and including the first zero set. Taking the XOR of the original number and this new colour would result in the incremented number.

Decrementing a number is the same as negating it, then incrementing it, then negating it again. Negation involves taking the NOT of a number, then incrementing it, but since the increment of the two negations cancel out, we need only to take the NOT of the argument before sending it into the incrementor and to take the NOT of the result to decrement the argument.

THE BITSHIFTER

The bitshifter takes advantage of redshift and blueshift filters to compute bitshifts. Bitshifting can actually be done in constant time, but doing so would require selecting between 29 redshift and blueshift filters which would take up too much space. Instead, the bitshifter splits the second argument into bits and bitshifts the first argument in powers of two, taking logarithmic time. For example, $x \ll 13$ would be computed as:

$$x \ll 13 = ((x \ll 8) \ll 4) \ll 1$$

THE BOOLEAN LOGIC UNIT

The Boolean logic unit is responsible for computing AND, OR, XOR and NOTAND. These operations have a one-to-one correspondence to the AND, OR, XOR and “subtract colour” filters. To compute these, the Boolean logic unit colours a filter bus with the first argument, chooses the corresponding filter with a piston, then sends the first argument through the filter.

THE CONTROL UNIT

THE DEMULTIPLEXER

The demultiplexer is a demultiplexer that chooses which component to activate according to the instruction. It contains delays to ensure that the components read the arguments from the operand bus in sync with the register reader writing arguments to the operand bus, since the operand bus is used for both arguments. It also contains delays to ensure that the writeback and fetch phases are executed only after the component has finished executing the instruction.

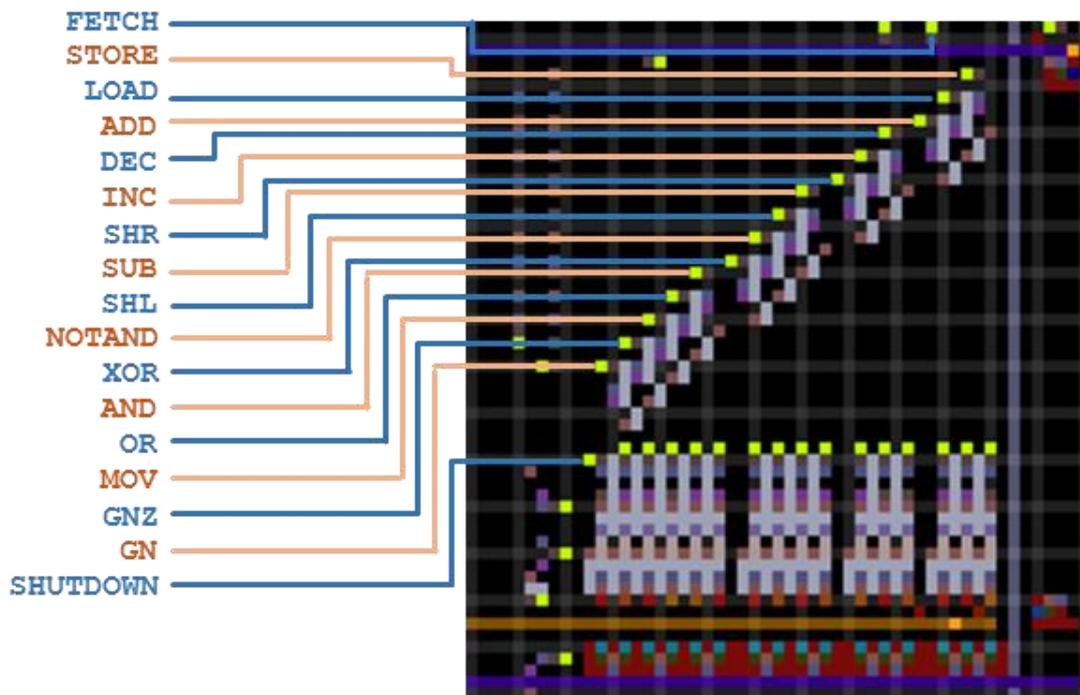


Figure 9 – The mapping between the demultiplexer and the individual instructions

Figure 9 – The mapping between the demultiplexer and the individual Figure 9 shows how the columns of the multiplexer correspond to individual instructions. Note that the spark signals used to activate the corresponding components are sent by the upward-pointing particle rays, not the leftward-pointing ones. The leftward-pointing particle rays are used only to time the beginning of the next writeback and fetch phases. The labels are directed to the leftwards-pointing ones only for clarity.

When adding to the Arithmetic/Logic Unit, no conductors should conflict with the paths of the upward-pointing particle rays, including the FETCH particle ray. The FETCH instruction is not in the LightPC instruction set, but is activated as part of the fetch phase of the LightPC instruction pipeline.

THE REGISTER READER

The register reader is a demultiplexer optimised to operate twice in succession. Additional circuitry is used to convert both arguments to the same format for decoding, as well as to choose between supplying an integer constant and supplying the value of the corresponding register to the operand bus.

THE REGISTER WRITER

The register writer is a demultiplexer and photon multiplier used to write the value in the operand bus to the register corresponding to the first argument of the instruction.

EXTERNAL DEVICES

THE EXTERNAL DEVICE COMMUNICATION PROTOCOL

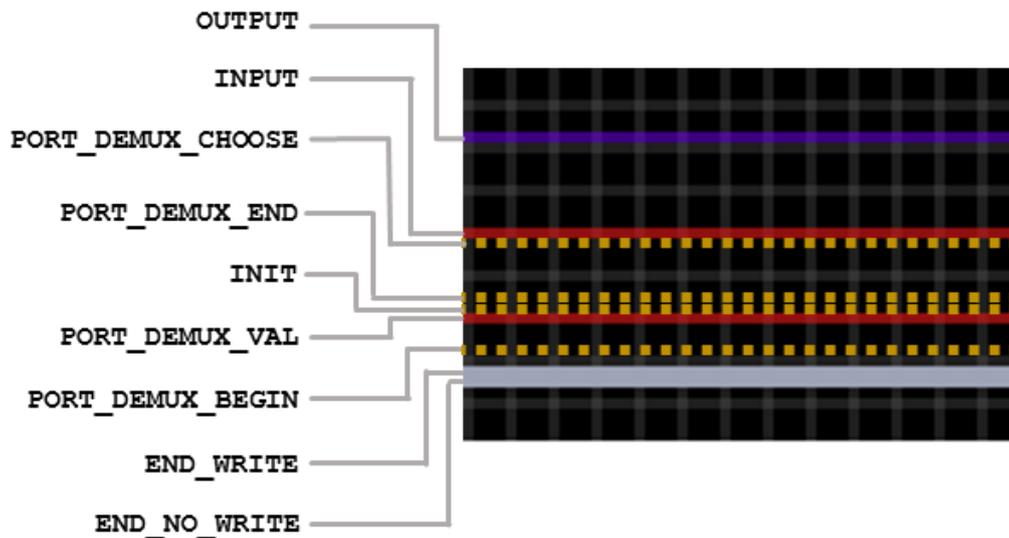


Figure 10 – Device communication buses

The yellow dotted lines are invisible particle ray buses. When creating new devices, do not put any conductors in those regions except for use in the device port.

INIT is sparked at the start of each program, when the On button is pressed. Use this to reset or initialise your device.

The device communication protocol begins with demultiplexing according to the device port identifier. Arguments are then sent to the device in a serial fashion. The following is the device input procedure for external devices:

Frame 0: The device identifier is written to PORT_DEMUX_VAL

Frame 2: PORT_DEMUX_BEGIN is sparked.

Frame 2: The first argument is written to INPUT.

Frame 9: PORT_DEMUX_CHOOSE is sparked.

Frame 11: PORT_DEMUX_END is sparked.

Frame 15: The second argument is written to INPUT.

When the device has finished processing the instruction, it should spark END_WRITE or END_NO_WRITE with a multi-input spark bus transmitter (see the section “LightPC Circuit Idioms”). If the device produces an output that is to be written to r14, END_WRITE should be activated. If not, END_NO_WRITE should be activated.

DEVICE PORTS

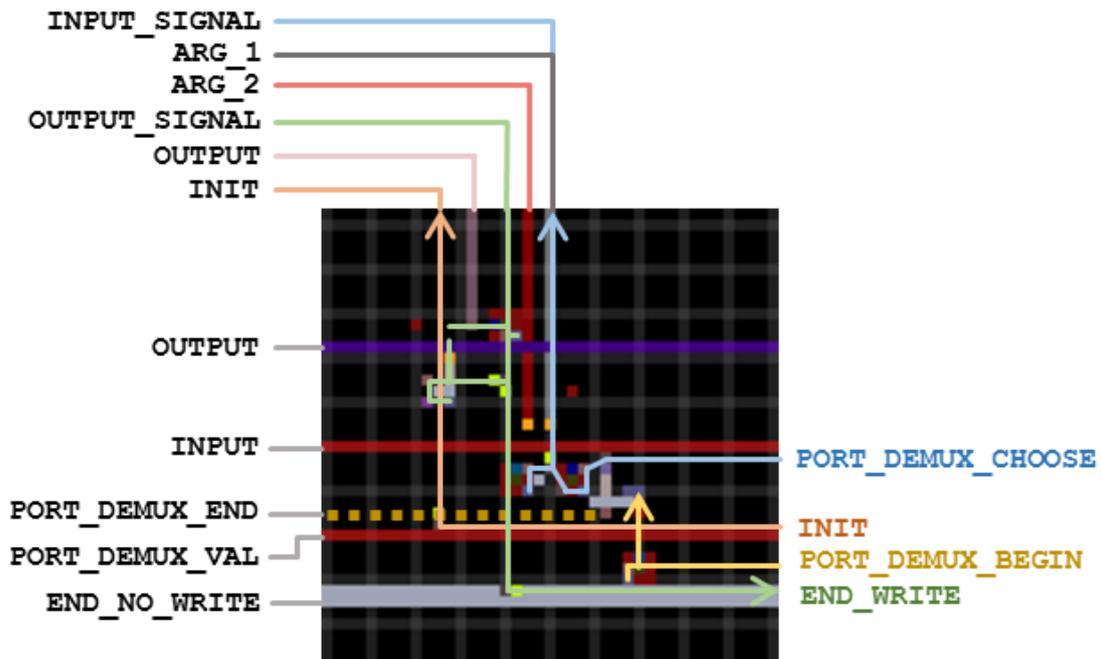


Figure 11 – The device port used for the pseudorandom number generator

Figure 11 shows the device port used for the pseudorandom number generator, which is sufficiently general to be used for any device. Two carefully timed photon emitters respond to DEMUX_CHOOSE to store the two arguments sent in series into parallel buffers (the buffer for ARG_2 is hidden behind the arrow for INPUT_SIGNAL) as well as to provide an INPUT_SIGNAL to activate the component. The device then sends an OUTPUT_SIGNAL which is used to write the output value to the output filter bus and directed to a multi-input spark bus transmitter in END_WRITE (the transmitter should be embedded in END_NO_WRITE instead if the device does not write to r14).

THE TOUCHSCREEN

The touchscreen is an amalgamation of two technologies: a keyboard and a screen. The screen is inspired by technology used in drakide's HD video and displays information by colouring a row of filter particles the pixel combination required, then using a photon multiplier and AND filter gates at every pixel to colour pixels accordingly.

The touch component comprises a multi-input spark bus for every row and column, sending a signal to the x- and y-coordinate bus writers corresponding to the pixel's row and column. A final two multi-input spark buses are used to relay the input to a central controller and instruct it to overwrite the buffer with the new input.

THE HEXADECIMAL DISPLAY

The hexadecimal display uses seven parallel demultiplexers to activate single units of compound NOTAND gates. The input panel, similar to the touchscreen, uses a multi-input spark bus to write the corresponding digit to a filter bus.

THE PSEUDORANDOM NUMBER GENERATOR

The pseudorandom number generator uses mercury to generate random bits. Each mercury particle can either be on the left or right of its insulator chamber at any one time. Since its position is mostly random, the mercury can be used to block the path of a spark particle ray to generate random bits. A series of OR filter gates collects the bits into a single number.

THE CLOCK

The clock is an incrementor attached to a spark loop with a 60 frame delay. Instructions to the clock read the number of the filter being incremented.