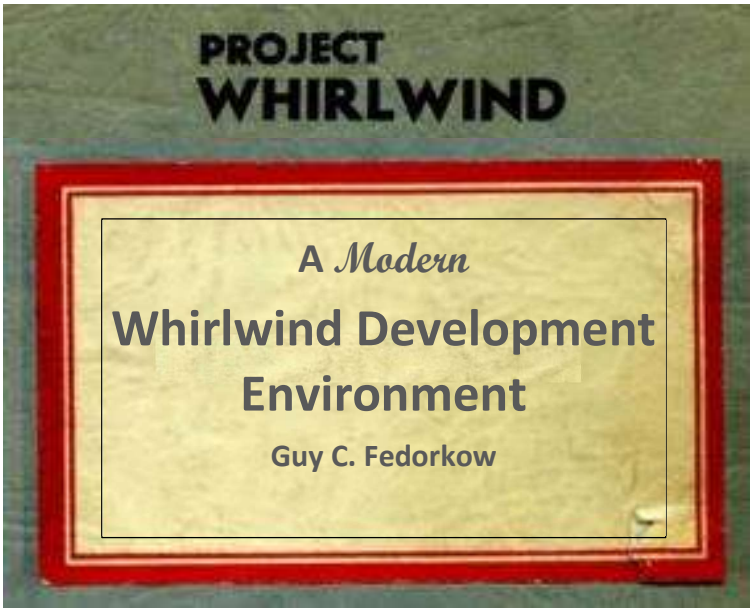


Revised Apr 30, 2024

Table of Contents

1. Introduction

The Whirlwind computer was built at MIT, and operated there from about 1950 to 1959, during which time numerous documents were written, and much code produced, some of which has been preserved on paper and magnetic tapes.

As part of our work restoring software from this project at the MIT Museum and the Computer History Museum, we've developed a development environment to decode existing Whirlwind programs, or write new ones, and to run them in a simulation environment, called the MitWiSE, MIT Whirlwind Simulation Environment.

This document outlines how to use the components of the MitWiSE.

The Environment contains a number of components:

- wwsim - An instruction set simulator that can execute Whirlwind programs

- wwasm – An assembler that can translate a modern-format source code representation of the instruction set into binary opcodes.
- wwdisasm – a disassembler that decodes (to the extent possible) existing Whirlwind binary programs, producing a format that can be easily commented and labelled, then converted back into executable binary format with embedded labels by the assembler.
- csii-convert offers a translator to convert source files written in Comprehensive System format into the MitWiSE assembler format, again, to allow code to be inspected, commented, labeled and assembled to an executable.

All of the tools are written in Python (approximately Python 3.11), and most can be found on GitHub at:

https://github.com/gfedorkow/Whirlwind-Instruction-Simulator

All of the tools, excepting the simulator itself, are command-line oriented, running in a Linux or emulated-Linux environment like Cygwin on Windows.

1.1.1 Emulator vs Simulator

Names Matter (up to a point)… there are Simulators, there are Emulators. Which is this?

At the extremes, a Simulator produces a complete environment disjoint from the physical world that can produce behavior similar to some other system. For example, a climate simulator is clearly a model that we want to behave like the real climate, but no one is going to confuse one for the other.

An Emulator is an object which is designed to stand in for another object. At the extreme, Control Data (and many others) made devices that 'emulated', i.e., were plug-compatible with, IBM disk drives. With a faithful emulation, the insides of the disk drive could be completely different from the originals, but they would operate interchangeably when plugged into an IBM mainframe.[1]

Our Whirlwind environment is half-way in between. It is a synthetic environment isolated from any dependence on specific physical object (i.e., a simulator), but it does run real Whirlwind code, without modification, at approximately real-time speed (i.e., an emulator).

But in the end, the difference doesn't matter much, so for now, it's a Simulator.

2. Whirlwind Assembler

The wwasm assembler is used to convert programs written in human-readable assembly language into code that can be executed in the MIT Whirlwind Simulator.

The input language inherits some characteristics of the original WW assembler (aka 'converter'), but adds on a number of more-modern constructs to allow for programs that, while (insignificantly) harder to parse, are substantially easier to document.[2]

2.1.1 ToDo: Proposed wwasm Feature Additions

In the course of reviewing, we've turned up a number of proposed feature additions to make the tool set a bit easier to use:

- `--Legacy_Numbers` command line option should also be available as assembler directives, to make it easier to get the right mode for any program that actually cares.

[1] Until they didn't, at which point a fraught debug session would ensue to assign 'blame' to one side of the plug or the other. Lawyers are standing by.

[2] i.e., variable names don't have to be one letter followed by up to three digits. And you can put in comments.

- Switch to a Real Parser to make arguments to .exec, .print .flex, and others more uniform, without surprises when they contain semi-colons, commas, or quotes
    - And implement more flexible expression parsing

2.2 wwasm Command Line Options

```
$ wwasm -h
usage: wwasm.py [-h] [--Debug] [--Legacy_Numbers] [-D] [--ISA_1950] [--outputfilebase
OUTPUTFILEBASE] inputfile

Assemble a Whirlwind Program.

positional arguments:
  inputfile             file name of ww asm source file

options:
  -h, --help            show this help message and exit
  --Debug, -d           Print lotsa debug info
  --Legacy_Numbers      guy-legacy - Assume numeric strings are Octal
  -D, --DecimalAddresses
                        Display traec information in decimal (default is octal)
  --ISA_1950            Use the 1950 version of the instruction set[3]
  --outputfilebase OUTPUTFILEBASE, -o OUTPUTFILEBASE
                        base name for output file
```

2.3 Source Code Format

[todo: write intro; insert a short sample program]

Source Code Syntax

- One instruction per line
- Multiple spaces and/or tabs may be used to separate fields[4]

```
   @Address:Value Label: opcode operand ; comment @@automatic-comment
```

e.g.:

```
   @0514:114243     a10: su     a11 + 4 ; subtract memory from the AC
```

Name	Optional?	Function

[3] Also available as a pseudo-op in the assembler

[4] Nothing depends on how many spaces or tabs are used, so tab spacing can be whatever you prefer in your fave editor (i.e., this is not python!)

Value:Addr	Yes	Added to the assembler output file if it's not in the source, and updated if it is, this field shows the result of assembly. See 2.9
Label:	Yes	
OpCode	No	One of 36 WW op codes in two-letter or three-letter format. Upper or lower case is accepted.
Operand	Yes	Zero is assumed [check this]
Comment	Yes	Free-form comment demarked by a semi-colon
Auto-Comment	Yes	The assembler will strip Auto-Comments and may insert new ones to indicate things like whether a line is the target of a branch somewhere else in the code. The Auto-comment is assumed to run to the end of the current line

2.4 Whirlwind operations – Op Codes and Operands

By default, the assembler follows the version of the instruction set described late in the Whirlwind program, in document 2M-0277 from 1958. Op-Code names are taken from that doc.

http://www.bitsavers.org/pdf/mit/whirlwind/M-series/2M-0277_Whirlwind_Programming_Manual_Oct58.pdf

Each opcode has a two- or three-letter mnemonic name, e.g. 'ca' for Clear and Add, followed by an argument. In most instructions, the argument is simply an address for an operand. There's only one addressing mode; the 11-digit field is used as an address in to Core Memory to find the operand.

A few instructions interpret the Operand field as a number rather than a memory address, and assign specific functions to bit locations, e.g., the shift instructions use operand bits to say how many bit positions the accumulator should be shifted, and what rounding mode should be used.

WWI programming depends on self-modifying code for indexed access to memory blocks, so it was common practice to use a zero in the operand field to indicate instructions that will be overwritten. Currently there's no capability to designate words in memory as "read only".

By convention, memory address zero always contains zero, memory address one contains one. So "`ca 1`" would result in a one in the AC.

There is (currently) no (syntactic) means to declare a location as read-only.

The assembler can be configured by a directive or command-line switch to use the 1950 version of the instruction set, prior to the introduction of the 'new' I/O system, where there were a number of op-codes dedicated to specific I/O devices (i.e., the CRT display). The instruction set version is set prior to execution of the first instruction, and can't be changed during execution.

2.5 Numbers

Whirlwind programmers had a (very annoying) habit of hopping back and forth between octal and decimal numbers. It's usually possible to tell which is which, but modern coders must stay alert.

In existing Whirlwind code[5]:

- If it's a six-digit number of the form 0.00000, i.e., zero or one, a decimal point, and five numeric digits, it's an octal number
- If it's a string of digits, it's probably decimal
- If it starts with + or -, it's decimal

In wwasm, we've tried to stick to the WW conventions, with one addition:

- if the number starts with 0o, ('zero-oh') followed by digits, it's octal *for sure.*

Wwasm flags numbers that won't fit in a 16-bit ones-complement word as errors, and won't generate an output file.

```
zero:     .word 0         ; plus zero
          .word 0o137     ; gcc-style octal
          .word 10        ; default decimal integer format
          .word 0.00010   ; Whirlwind octal positive number; must be five digits to
the right of the fixed point
          .word 1.11101   ; Whirlwind octal negative number; must be five digits to
the right of the fixed point
          .word 1.77777   ; Whirlwind negative zero
          .word +0.9      ; Whirlwind positive decimal fraction
          .word +0.9999   ; Whirlwind positive decimal fraction
          .word +0.99999  ; Positive decimal fraction Overflow
          .word +1.0      ; Positive decimal fraction Overflow
          .word -0.9      ; Whirlwind negative decimal fraction
          .word -0.9999   ; Whirlwind negative decimal fraction
          .word -0.99999  ; negative decimal fraction Overflow
          .word -1.0      ; negative decimal fraction Overflow
```

2.6 Expressions

The assembler will evaluate simple expressions comprising numbers, labels, add and subtract. The assembler currently cannot support parentheses, multiply, divide or other operations.

e.g.

```
      .pp table_fixed_offset=5
      ca table_base + table_fixed_offset - 1
      sp next_instruction
table_base:
      .word 12
```

[5] We do actually have quite a few programs in Comprehensive System source format, although with two-character variable names and no comments, the source is not a lot easier to read than the binaries!

Surprise Warning! At the moment, the expression parser is hopelessly ad-hoc… operators (‘+’ and ‘-‘) *must* be separated from numbers and labels by whitespace. The same character without a space is treated as the sign for a number. So “MyLabel - -1” would add one to MyLabel.

I hope we can add a cleaner expression parser!

2.7 Labels

Wwasm accepts variable-length, relatively free-form labels. Labels must start with a letter, followed by any combination of letters and numbers, and underscores (‘_’).

Note that rules for labels in Comprehensive System source code format have numerous restrictions and special cases, and require conversion to wwasm format. See M-2539-2 Comprehensive System manual and Section 2.10. None of the CS restrictions on variable names are enforced in wwasm.

2.8 Pseudo-Ops

Wwasm accepts a variety of pseudo-ops to control its behavior. None of these pseudo-ops were present in the original CS-II assembler (kind of except for the JumpTo and Program Parameter ops, which were handled as special cases).

Unless otherwise noted, *string* arguments are delimited by white-space, i.e., quotes or parens are interpreted as part of the string.

2.8.1 `.WW_TapeID:` *string*

```
.WW_TapeID: fbl227p50
```

This optional pseudo-op string passes along any “Tape ID” string detected by the disassembler from the 556 header into the core-file and assembler.

2.8.2 `.WW_File` *string*

```
.WW_File 102663328_fb131-0-2690_new_decoders_3of4.7ch/bounce_gs001_.tcore
```

This optional pseudo-op string passes along the posix source file name, so that it’s visible in the output .acore file.

[Oops, looks like TapeID does have a colon and WW_File does not]

2.8.3 `.isa [1950|1954]`

```
.isa
```

The assembler defaults to the 1954 instruction set, primarily featuring the new I/O system (SI, RC and RD). Earlier single-purpose I/O commands can be used if the 1950 instruction set is chosen.

The ISA can’t be changed dynamically, i.e., the instruction set is chosen before the first instruction is executed, no matter where the .isa pseudo-op appears in the code.

2.8.4 `.switch` *name abs-expr*

```
.switch LeftInterventionReg 0.00100
```

The .switch pseudo-op instructs the assembler to configure one of the (zillion) Whirlwind control panel switches. The *name* argument can be one of a list of named switches. The *abs-expr* is a numeric value to be loaded into the switch prior to the start of simulation. Most switches are either a single bit or can contain a 16-bit word.

The .switch pseudo-op is meant to configure many of the switches, some of which are mapped into the I/O address space, accessed via `si` and `rd` instructions, some of which are hard-wired to machine behavior (e.g. *`CheckAlarmSpecial`*.)

The .switch settings are not dynamic; they are applied prior to the start of execution of WW code.

[The intention so far is that the switch value is configured before the simulation starts to run, i.e., it's not (as of now) interpreted as part of the instruction stream.]

The assembler does not validate the name string, but just passes the name and value through to the simulator.

Flip-Flop Register Preset switches are a special case; the .switch pseudo-op can preset flip-flop registers anywhere in the lower 32 words of the address space.

The simulator implements two switch registers, "`LeftInterventionReg`", "`RightInterventionReg`", and thirty two Flip-flop Register Preset Switches formatted as "`FlipFlopPreset%02o`".

Other switches include:

- `CheckAlarmSpecial` – modifies behavior of the CK ("check") instruction to either Halt or not on a mismatch. (See 2M-0277)
- `FFRegAssign` – modifies which FF Register appears at which address. Don't mess with this unless you know what you're doing. By default, the FF Registers are assigned addresses `0o2, 0o3, 0o12, 0o16, 0o22.` [Check this!]

2.8.5 .dbwgt *addr-expr [step_value]*

```
.dbwgt Vxdt 0o100
```

This pseudo-op adds a numerical memory address or label to the list of "debug widgets" displayed on the xwin CRT emulator.

The simulator will display the current value of each memory location as the simulation runs, and a can be used to select and update values as the sim runs using PC Arrow keys. Up and Down arrows select and highlight the Var to be changed, Right and Left arrows increment or decrement the value. The optional *step_value* indicates what should be added or subtracted with each press of the Right or Left arroes.

The .dbwgt pseudo-ops are evaluated once prior to execution of the first instruction.

2.8.6 .jumpto *addr-expr*

```
.JumpTo 0o157
```

This pseudo-op instructs the simulator as to the starting address for the program.

The assembler doesn't pay attention to the start address, and the simulator will default to 0o40 if nothing is specified.

2.8.7 .org *addr-expr*

```
.org 0o40
```

The .org pseudo-op sets the current program counter, setting the address for the next instruction word.

2.8.8 .daorg *addr-expr*

```
.daorg
```

Later versions of the 556 loader format allowed specification of address in physical memory, but also the address to be used to store the file on the drum.

[I don't recall why this matters, but the tape decoder decodes it, so the Assembler passes it on\…]

2.8.9 .pp *expr*

```
.pp B1=0o40000
```

Preset Parameter is an assembler function like a #define, to configure addresses during assembly of a program. The CS-II implementation of Preset Parameters has quite a few rules on naming (see M-2539-2 Comprehensive System manual, pdf page 75) but this simulator treats preset params as additional symbols in the symbol table.

Preset Params are meant (I think) for doing address assignments and calculations, so the values associated with a PP are restricted to 0-2047.

```
# Preset Parameters are like #define statements; they don't generate code, but
# they do put values in the symbol table for later use.
# They can be "called" in the source code as a way to insert a word of whatever value was
# assigned to the pp.
#  Samples look like this:       .PP pp15=pp14+1408  ;  @line:19a
#  The program parameter label (e.g. pp15), apparently is always two letters plus one or two
digits,
#  but not necessarily always 'pp'
# I am not so sure how to handle these guys!
# See M-2539-2 Comprehensive System manual, pdf page 75 for the CS-II rules.  This assembler
# doesn't enforce any of the rules; a Preset Param is just another label in the symbol table.
# Preset Parameters are only represented as labels in the symbol table for the simulator, i.e.,
# the .pp pseudo-op is not passed through.
```

Labels used in .pp expressions must be defined before use.

E.g, this code is not legal:

```
      .pp L201=L203 + L204
L203:   ad L204
L204:   .word 123
```

While this is legal:

```
L203:   ad L204
L204:   .word 123
      .pp L201=L203 + L204
```

2.8.10 .word *abs-expr*

```
.word
```

Insert a 16-bit word initialized with the specified absolute value at the current address and advance the address pointer.

2.8.11 .flexh *char*, .flexl *char*

The .flex[h|l] pseudo ops initialize a word at the current assembly location containing a flexowriter character. The operand can expressed as an numeric string, in which case both .flexh and .flexl simply use the operand as the value to put in the allocated word.

The operand may also be single quoted ASCII character, in which case it will be converted to a Flexo code and shifted (or not) as indicated by [h|l].

```
.flexh "A"
```

Insert a 16-bit word initialized with the Flexo equivalent of the ASCII character given as an argument, shifted to the top six bits of the word.

```
.flexl "B"
```

Inserts a 16 bit word containing the Flexo equivalent of the ASCII char, but in the lower six bits of the word.

Note that there are many ASIC characters that don't have Flexo equivalents (e.g. "[" or ";"), and will raise an error.

2.8.12 .exec

```
.exec python-stmt
```

Bypass the usual simulator flow and execute a python statement.

This pseudo-op is executed *before* the following instruction.

i.e.

```
      ca 12
      cp branch_taken
      .exec if debug: print branch_not_taken
      cp 13
branch_taken:
      ad 14
```

2.8.13 .print

```
.print "format_str", addr1, addr2, …
```

Formats

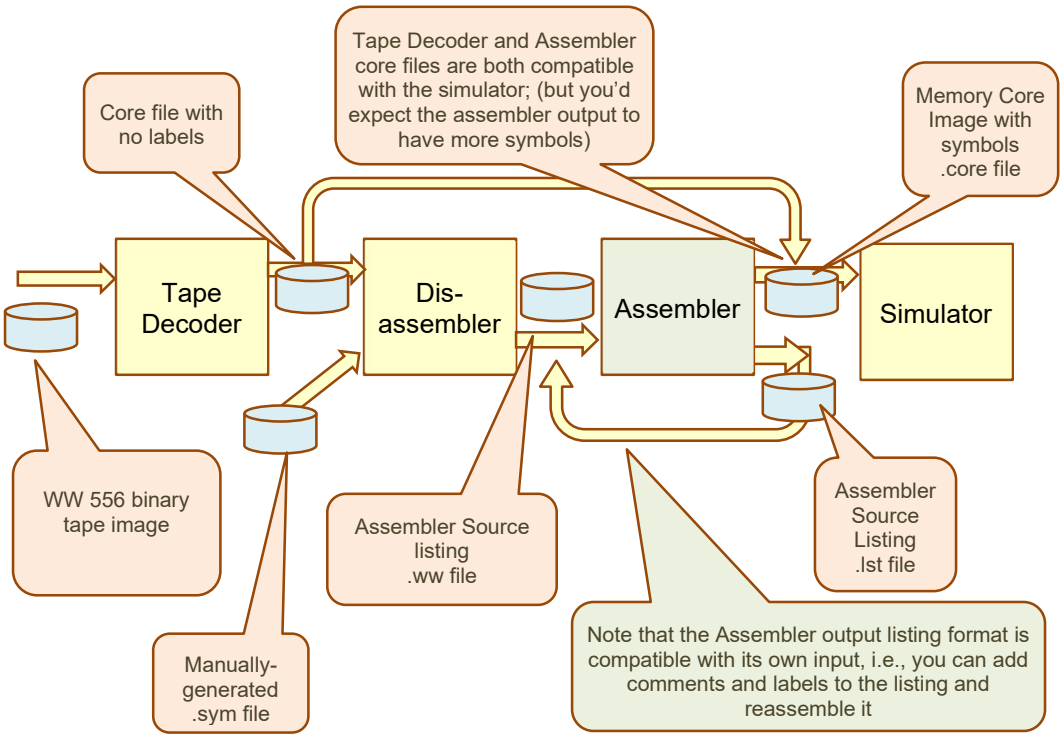
- %d, %o – format the corresponding memory addresses as decimal or octal
- %ad, %ao – insert the content of the accumulator, either in decimal or octal

This pseudo-op bypasses the usual simulation flow and causes Python to print a message, typically for debug. The string can contain printf-like formatting indicators, followed by a corresponding list of memory addresses.

The .print statement is executed prior to the following instruction, as with .exec.

2.9 Listing and Source File

The wwasm assembler is intended to fit in a tool chain mostly optimized around reverse-engineering archival Whirlwind material. As such, a critical use of the assembler is to re-assemble disassembled code as labels and comments are added, and to accommodate modest changes to program flow when necessary. To accomplish this goal, the assembler is designed to read its own output listing.



A typical flow might be:

a) Run a binary object (.tcore file) in the simulator to start to understand the flow
b) Disassemble the .tcore to obtain an initial uncommented .ww source file
c) Start adding comments and labels to the .ww file
d) Assemble the .ww file to produce a .lst file, showing source code plus comments, along with the octal representation of each instruction and variable in memory.
e) Re-name the .lst file with a .ww extension.
f) Run the simulation. Labels from the source will show up in simulation traces.
g) Keep editing the new .ww to add more comments and labels, plus debug .print statements as needed
h) Until (Done or It's_Dinner_Time), GoTo (f).

2.10 CS-II Code Converter

[this currently is a separate program that converts a file in CS-II format into .ww format]

2.11 Core File Format

The .ww files give source code in a more-modern assembler format than the WW guys used, so I hope that's more readable than CS-II source. The assembler outputs a listing .lst file (which can be used again as input to the assembler!) and a .acore file, which contains the core image.

The core image is ascii, and essentially describes what values to load into memory prior to launching the program execution. Any locations not specified, or indicated in the core file as None are set to None in the simulator's memory, triggering an error if the location is used before being initialized.

Lines in the file have the following formats:

- @Cnnnn: gives eight 16-bit octal words, starting at octal address nnnn. Use `none` to indicate that a location is uninitialized. Values to be loaded are always octal, up to seven digits long.

- @Snnnn: gives one entry in the symbol table. ‘nnnn’ is the address of the symbol, the name of the symbol follows the colon..
- @Nnnnn: records the source file comments (if any) for a given address word.
- @Ennnn gives lines that contain pseudo-ops.
- %JumpTo nnn gives the start address
- %TapeID: optionally gives the ASCII string used as the identifier for this binary
- %File: optionally gives the source file name

Lines which begin with a semicolon are treated as core-file comments and ignored. Blank lines are ignored.

In general, a “.tcore” represents the image read from a tape and reformatted from 5-5-6 into memory locations. A “.acore” file is generally the output of the assembler, which may have symbols and comments embedded. The two files have identical structure, just different sources.

For example:

```
; *** Core Image ***
%File: 102766750_fb131-0-2692_number_display.7ch
%TapeID: fbl3l-0-2692
%JumpTo 0o40
@C0000: 0000000 0000001 0000000 0000000  None    None    None None
@C0040: 0100200 0040002 0100201 0040003 0100202 0040600 0100203 0040601

@S0276: dot_xpos
@S0277: acctmp
@S0300: do_dot
@S0500: u0500
@S0501: chr_cnt

@N0040: load constant 0o12 into AC
@N0041: Store it in a FF Reg(??)
@N0042: load constant 0o140510 into AC

@E0042: print: "execute .print" ["1", "2"]
@E0043: exec: if True: print("executed .exec")
```

3. Simulator

The MitWE simulator offers instruction-level emulation of the Whirlwind instruction set, as defined (mostly) in 2M-0277, including a number of I/O devices (but not all). It does not emulate the internal timing “micro states” within a single instruction.

The simulator operates by first reading a “core file” into its simulated 2K memory unit. Upon start, it goes the address of the first instruction, fetches it, executes it, goes to the next, etc., until it hits an alarm or halt, or the cycle count specified on the command line runs out.

3.1 Command Line Options

There are quite a few command line arguments to the simulator, although the only one that's actually needed is the name of the core file containing code to be run. In many cases, *none* of the additional arguments may be required.

```
usage: wwsim.py [-h] [-t] [-a] [-f] [-j JUMPTO] [-q] [-D] [-c CYCLELIMIT]
                [--CycleDelayTime CYCLEDELAYTIME] [-r] [--AutoClick]
                [--AnalogScope] [--NoXWin] [--NoToggleSwitchWarning]
                [--LongTraceFormat] [--TraceCoreLocation TRACECORELOCATION]
                [--PETRAfile PETRAFILE] [--PETRBfile PETRBFILE]
                [--NoAlarmStop] [-n] [-p] [--NoZeroOneTSR]
                [--SynchronousVideo] [--CrtFadeDelay CRTFADEDELAY]
                [--DumpCoreToFile DUMPCORETOFILE]
                [--RestoreCoreFromFile RESTORECOREFROMFILE]
                [--DrumStateFile DRUMSTATEFILE] [--MuseumMode]
                [--MidnightRestart]
                corefile

Run a Whirlwind Simulation.

positional arguments:
  corefile              file name of simulation core file

options:
  -h, --help            show this help message and exit
```

3.1.1 Program Launch Options

This group of options sets up the behavior of the simulation before the first instruction is fetched.

```
  -j JUMPTO, --JumpTo JUMPTO
                        Sim Start Address in octal
```

JumpTo gives the start address for the first instruction. This defaults to 0o40 if not specified on the command line or in the core file. If both are present, the command line has the final say on where to start.

```
  -D, --DecimalAddresses
                        Display trace information in decimal (default is
                        octal)
```

WW programmers switched back and forth between Octal and Decimal with dismaying alacrity. Sometimes it's easier to debug when the traces are in the same base that the programmer was using.

```
  -c CYCLELIMIT, --CycleLimit CYCLELIMIT
                        Specify how many instructions to run (zero->'forever')
```

Many Whirlwind programs were written to "run forever"; this option allows a way to run some cycles and then bail out. The default, or a cycle limit of zero, causes the simulator to run 'forever' until a halt or some manual intervention.

```
  --CycleDelayTime CYCLEDELAYTIME
                        Specify how many msec delay to insert after each
                        Instruction
```

CycleDelayTime intentionally slows the simulation to "human speed". Sometimes useful to see what order graphics are being displayed.

```
  --AutoClick           Execute pre-programmed mouse clicks during simulation
```

Some simulations invoke a mechanism on startup to 'automatically' click a bunch of buttons or switches before reverting to user input for light-gun hits. Examples are Air Defense and Nim, both of which have startup sequences that need to be repeatable for reliable debug. The Autoclick function is mostly implemented in per-application project_exec.py files.

```
-r, --Radar            Incorporate Radar Data Source
```

This is a special-purpose argument that turns on the simulated radar system, specifically for the air-defense M-1343 simulation. Turning this on does have the (intended) side effect of repurposing two Toggle Switch Register addresses.

```
--NoAlarmStop          Don't stop on alarms
```

Default WW behavior is to halt on any alarm (especially annoying for arithmetic overflow. Often it's easier to debug a program by ignoring the overflows and following the flow, then going back later to see what caused the overlow.

```
--NoZeroOneTSR         Don't automatically return 0 and 1 for locations 0 and 1
```

The convention for Whirlwind programmers was to put constant zero in location zero, and constant one in location one. The simulator does that by default. But it is possible to change those first two locations (e.g., they could be used as the first two words of a program entirely in TSR).

```
--PETRAfile PETRAFILE File name for photoelectric paper tape reader A input file
--PETRBfile PETRBFILE File name for photoelectric paper tape reader B input File
```

There are I/O instructions for reading from the two Photo Electric Tape Readers. If encountered, the simulator will read bytes from the specified file to represent whatever would have been on the tape.

```
--DrumStateFile DRUMSTATEFILE
                       File to store Persistent state for WW Drum
```

There are instructions to read and write the drum; file is used as the non-volatile backing store for the emulated drum device

```
--DumpCoreToFile DUMPCORETOFILE
                       Dump the contents of core into the named file at end
                       of run
--RestoreCoreFromFile RESTORECOREFROMFILE
                       Restore contents of memory from a core dump file
```

These two optional args make it possible to glue two simulation runs together by carrying core memory state forward from one to the next.

3.1.2 Display Options

A critical part of the WW design was the Oscilloscope Display. i.e., a device for drawing points and vectors on a large cathode-ray-tube. Coupled with that was the Light Gun[6] pointing device.

The Oscilloscope Display was not used in every program. In the WW simulator, by default, the Oscilloscope is emulated in a graphical X-Window created specifically for that purpose. The graphics window is only created on the first access to the display device by the WW program, so it's not instantiated at all unless required. Also by default, mouse clicks in the graphics window are interpreted

[6] Fear Not! This 'gun' shoots nothing! However, it does *detect* photons from the CRT display. Even the TSA agrees that this 'gun' can be carried on an airplane.

by the simulator as “light gun hits”. The 2M-0277 documents which objects can be seen by the light gun and which cannot;[7] the simulator does the same as best we can.

There are a few arguments that control the behavior:

```
  --AnalogScope          Display graphical output on an analog CRT
```

This complex argument switches the display from an emulated oscilloscope on the laptop to a real analog oscilloscope, driven in X/Y/Z mode the same way as the original WW displays. This option only works on RaspPi with the Rainer display adapter (which needs an xref).

```
  -p, --Panel            Pop up a Whirlwind Manual Intervention Panel
```

The Panel option activates an additional window emulating the Whirlwind ‘control panel.’ (Yeah, ok, it was a Control Room with hundreds of buttons and lights. This is a small subset of the key control points.) See notes in 3.2.

```
  --NoXWin               Don't open any x-windows
NoXWin prevents a graphics window from being popped up at all.

  -n, --NoCloseOnStop    Don't close the display on halt
```

Normally when the WW simulator hits a halt or a trap, the sim process terminates and the oscilloscope display is closed. NoCloseOnStop holds the window in suspended animation until the simulator process is terminated. This is handy for debugging when the code hits an unexpected trap while painting the screen.

```
  --SynchronousVideo     Display pixels immediately; Disable video caching buffer
```

In the Real Whirlwind™, each graphics instruction has an immediate result on the CRT. There’s no buffering, no rasters, just voltages and points. With a modern PC-style display, that’s pretty inefficient, so the default behavior is to buffer a series of screen updates. SynchronousVideo causes the simulator to draw each and every point one at a time. Helpful for debug when you want to see which part of the code is drawing what part of a display.

```
  --CrtFadeDelay CRTFADEDELAY
                         Configure Phosphor fade delay (default=0)
```

The CRT emulator must not only draw points and lines, but erase them after a short delay to yield the effect of phosphor decay. This knob allows some tuning of the phosphor decay characteristics. The default almost always works, without using this parameter.

3.1.3 Instruction Trace

This group of arguments controls the type of trace information generated during a simulation run. The default is to print a line of output for each branch and each I/O instruction. [I’m thinking of changing the default to “-q” to cut the noise!]

```
  -q, --Quiet            Suppress run-time message
```

Quiet suppresses most of the non-error messages, so there’s less clutter on the console

```
  -t, --TracePC          Trace PC for each instruction
  --LongTraceFormat      print all the cpu registers in TracePC
```

[7] Basically, the light gun will respond to Points but not Lines or Characters on the WW display

TracePC outputs one line for each instruction executed, showing PC, instruction and a few of the key registers, allowing the Reverse Engineer to see where the program is going. Capturing this output in a file yields a way to retrospectively see "how the heck did we get here?". LongTraceFormat adds a couple more registers, but it's rarely needed (by me, anyways!)

```
--TraceCoreLocation TRACECORELOCATION
                      Trace references to Core Memory Location <n> octal
```

TraceCoreLocation prints a line each time a specific core location is read or written

```
-a, --TraceALU        Trace ALU for each instruction
```

TraceALU turns on more detail on each arithmetic operation. Essential for figuring out if Ones Complement is working, Too Much Information for anything else.

```
-f, --FlowGraph       Collect data to make a flow graph
```

See section 3.6. This complicated feature generates a 'flow chart' for the program as executed

```
--NoToggleSwitchWarning
                      Suppress warning if WW code writes a read-only toggle
                      Switch
```

The Toggle Switch Registers (TSR) are not writable (they're supposed to be physical switches, dude!) But that doesn't mean the first 32 words are all unwritable. The five Flip Flop registers overlay TSR address space, and they are writable. In the real machine, and by default in the simulator, writes to TSR are simply ignored, as they were often used when the programmer needed the side effect of a write instruction, but didn't want the result. This flag issues a warning if a write to unwritable TSRs is executed.

3.1.4 Specialized

These are meant specifically for unattended museum exhibit use.

```
--MuseumMode          Cycle through states endlessly for museum display
--MidnightRestart     Restart simulation daily at midnight
```

3.2 Control Panel

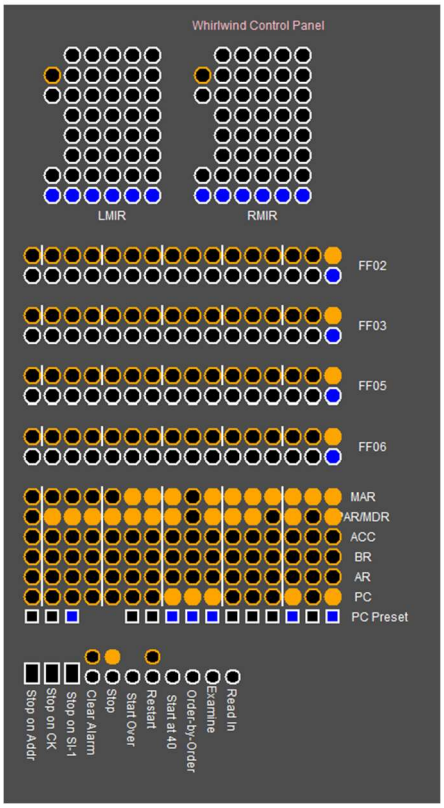
[New, as of Apr 2024]

The `-Panel` option for the simulator pops up a "Control Panel" similar to what the machine operator would have seen in the Whirlwind control room. [I need a Blog Article on this; xref the CHM object], allowing the sim user to set parameters and run programs more-or-less as they did on the Real Machine™.

There's a Graphical Code to follow:

- Orange is an Indicator light
  - Orange circle with a black center is a Zero indication.
  - A filled orange circle shows a One indication.
  - Clicking an indicator has no effect.
- White is a switch or push button
  - White outline with a black center is a switch set to Zero
  - White outline with a blue center is a switch set to One.
  - Clicking on a switch will change its setting.

All indicators are updated as the simulator runs.



The top segment of the panel represents the two “Manual Intervention Registers” with corresponding Activate buttons. Each MIR (“Left” and “Right”) represents one 16-bit number, coded as five one-of-eight radio buttons (and a “one of two” radio button for the extra bit!).

The two Activate buttons are ordinary push buttons; any press “sets” the corresponding Activate bit and turns on the lamp; the lamp is cleared when the bit is read by WW software. (For some reason, they’re “Upper” and “Lower” Activate buttons; I’m assuming Upper is on the left).

The next rows represent a subset of the five possible Flip Flop Registers. Each one displays its current state in 16 indicators. The associated row of buttons (really Toggle Switches) gives the preset values. When WW starts, or executes the Reset FF instruction, the value in the toggle switches is copied into the corresponding register.

While in the “real machine” toggle switches would have to be set manually, the simulated switches are initialized to whatever the FF Register Preset values optionally in the core file. [check the name]

The simulator currently shows four of the five FF Registers, indexed by their conventional address in the lower 32 address locations. In practice, FF2 and FF3 are often used for parameter input, while the others are not.

The next block of indicators shows the state of various registers, including Accumulator, B Register, AR and Program Counter. The row of switches gives a preset value for the Program Counter, used by Start Over (below). By default, the simulator sets the PC Preset switches to the Jump-To address if there is one in the core file.

The bottom row controls the run-time operation of the machine.

In normal operation without the control panel, the simulator wakes up in Run state, having already “read in” the core file, and immediately fetches the first instruction. However, with `-Panel`, the machine wakes up in Stop state and waits for a button to start.

- **Start Over** will reset the PC to whatever is in the PC Reset switches (defaulting to the Jump-To address).
- **Restart** will simply start to execute at whatever address is in the PC
- **Start at 40** will reset the PC to 0o40 (the default start address for many WW programs) and run from there.
- **Stop** sets the run state to stop, leaving registers all the way they were when the Stop button was pressed. Restart will pick up from wherever it left off.
- **Order-by-Order** is like Restart, but it runs just one instruction and then returns to Stop state.
- **Readin** starts over by reading in a new core file.

Other settings don’t change the Run state directly:

The Examine button only works when the machine is Stopped, and reads the memory location corresponding to the PC Preset register, leaving the result in the PAR/MDR[8] indicators.[9]

[8] In the WW docs, this indicator seems have been added to debug the data going into Parity Check circuitry, but it always shows the contents of the last memory location read, i.e., it’s a Memory Data Register, so it’s unclear if it’s a PAR or MDR. The last address used is shown in the Memory Address Register indicator, which is unambiguously a MAR.

[9] You can Peek, but you can’t Poke 😊

3.3 I/O Devices

By the end of the project, Whirlwind had a lot of I/O gear attached. The simulator includes some of it:

- Flexowriter
- Teletype (rarely used)
- Graphics CRT & Light Gun
- Photo Electric Paper Tape reader(s)
- Drum Storage

In addition, there are special-purpose I/O hacks for radar and light-gun to work with the M-1343 Air Defense demonstration.

3.3.1 Graphics Output and the Light Gun

Whirlwind had large CRTs used for graphical output, and (through the light gun) input. The programming and hardware interface to these devices is *much* simpler than conventional raster graphics devices, simply using a pair of D/A converters to position a beam on an x-y oscilloscope display, with a single bit for each 'scope to turn the beam on and off. As a result, the WW programmer is responsible for repainting the display often enough to keep the flicker down to an acceptable level. Real-time programming indeed.

Graphics primitives can draw points and line segments, with a hardware feature added later to trace out a seven-segment character as a series of short segments.

[picture]

For a modern programmer, this programming model is completely counter-intuitive. There are commands to draw points and lines, but there are no commands to erase them or clear the screen. That's because the WW instructions draw the point on CRT phosphor in about 65 usec, lighting it up for a fraction of a second before the natural decay of the phosphor causes the point to be extinguished. If you want a stable display, you need to keep redrawing it often enough to hide the inherent flicker.

The simulator has two [for now] ways of emulating this I/O mechanism.

- On RasPi, Windows or Mac, the simulator calls a library that creates a window on the screen, and emulates the drawing of points and lines on 'phosphor' that automatically fades with time. Mouse clicks in the graphics window can be interpreted as Light Gun hits.
- On RasPi only, the simulator alternately can activate Rainer Glaschick's "WWI-VecIF" hardware interface, which drives one or more analog oscilloscopes, using circuitry much closer to the "real" Whirlwind. This interface also includes a physical Light Gun that senses points drawn on the CRT in the same way the real WW did.

3.4 Simulator Files

3.4.1 Core Files

See Section 2.11.

3.4.2 Project_exec.py Files

3.5 Program Trace Log

3.6 Flow Graph

4. Ancillary Tools

4.1 wwdisasm – Disassembler for Whirlwind Binaries

4.2 wwdiff – Compare core files

4.3 wwutd – Universal Tape Decoder

5. End of Document