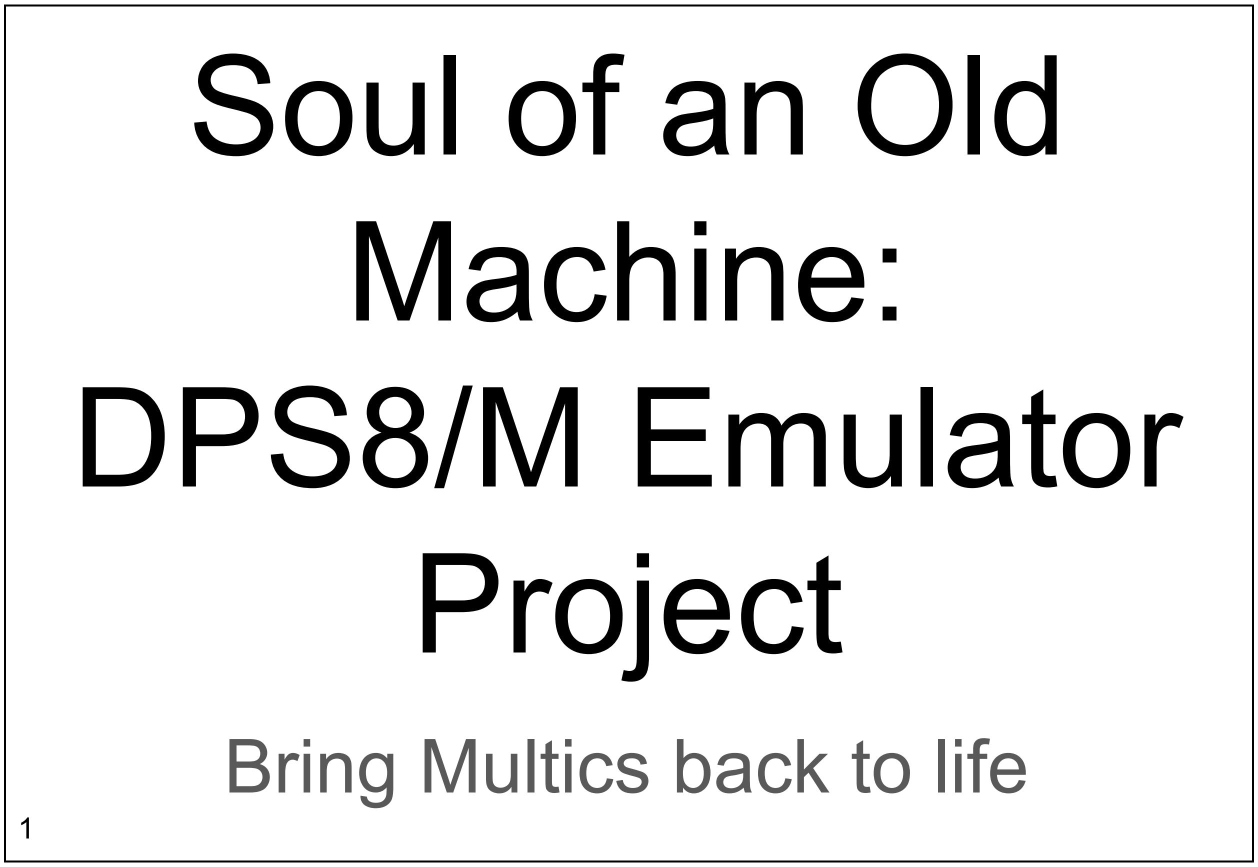
DPS8-M Emulator history



Soul of an Old Machine: DPS8/M Emulator Project

Bring Multics back to life

1

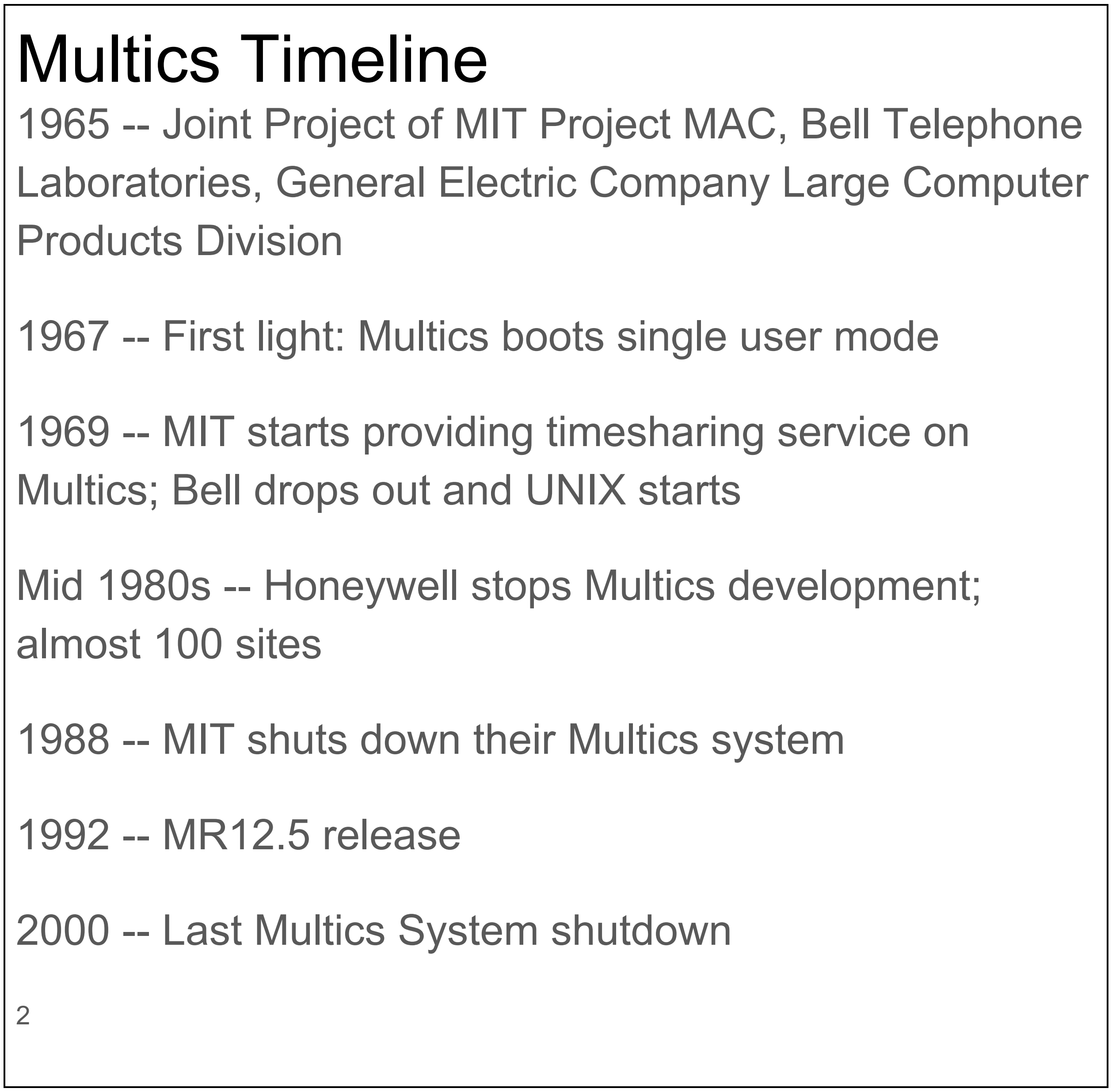
Welcome.

My name is Charles Anthony, and I am going to talk about what I have been doing for the last four years: bringing the Multics operating system back to life.

The DPS8/M emulator project is an open source collaboration with the aim of creating an emulation of the DPS8/M mainframe computer and running the Multics operating system on it.

I am going to briefly cover the history of Multics, the DPS8/M computer and DPS8/M emulators; and then go into detail about the dps8m emulator project and my history with it.

This is going to be both deeply technical and deeply personal -- this project has been my passion for the last four years, with victories and defeats. The thrill of shipping release 1.0; the agony of intractable bugs.



Multics Timeline

1965 -- Joint Project of MIT Project MAC, Bell Telephone Laboratories, General Electric Company Large Computer Products Division

1967 -- First light: Multics boots single user mode

1969 -- MIT starts providing timesharing service on Multics; Bell drops out and UNIX starts

Mid 1980s -- Honeywell stops Multics development; almost 100 sites

1988 -- MIT shuts down their Multics system

1992 -- MR12.5 release

2000 -- Last Multics System shutdown

The Multics timeline.

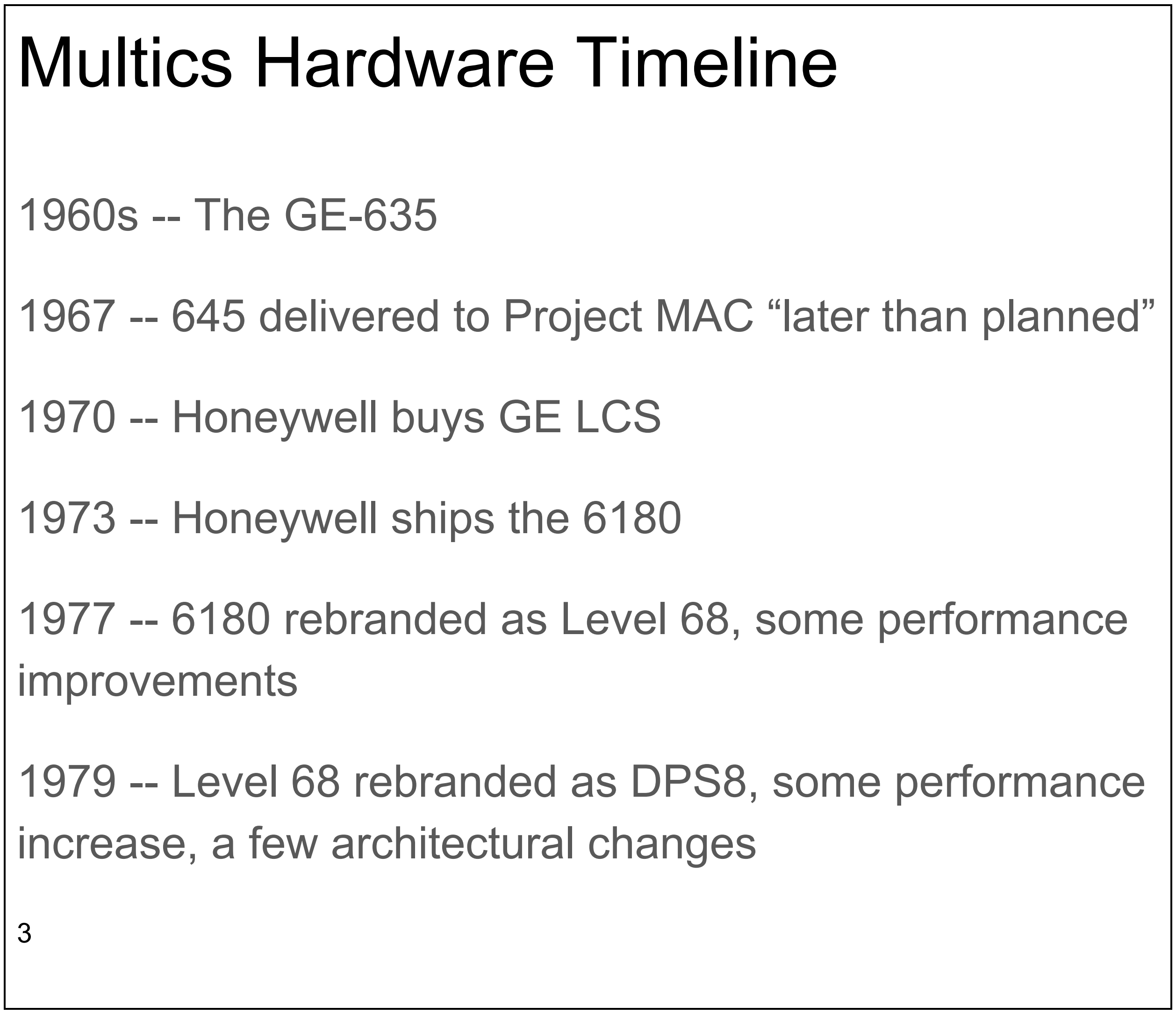
In 1965, Project MAC, Bell Labs and GE formed a joint project to create a timesharing operating system: Multics -- Multiplexed Information and Computing Service.

In 1967, Multics boots in single process mode; a few months later, multiple processes were running; then “gross discrepancies between actual and expected performance of the various logical execution paths throughout the software [led to] an unanticipated phase of design iterations.”

In 1969, a GE 645 with Multics was delivered to MIT and runs a time-sharing service. In the same timeframe, Bell drops out of the Multics project and UNIX starts development.

After 19 years, MIT shuts down their Mutics system.

The last release of Multics, MR12.5 was in 1992; in 2000, the Canadian Department of National Defence shuts down DND-H, the last running Multics system.



Multics Hardware Timeline

1960s -- The GE-635

1967 -- 645 delivered to Project MAC “later than planned”

1970 -- Honeywell buys GE LCS

1973 -- Honeywell ships the 6180

1977 -- 6180 rebranded as Level 68, some performance improvements

1979 -- Level 68 rebranded as DPS8, some performance increase, a few architectural changes

3

The Multics hardware timeline.

Multics needed hardware support that did not exist at the time; computer makers were solicited for proposals to provide systems with that support.

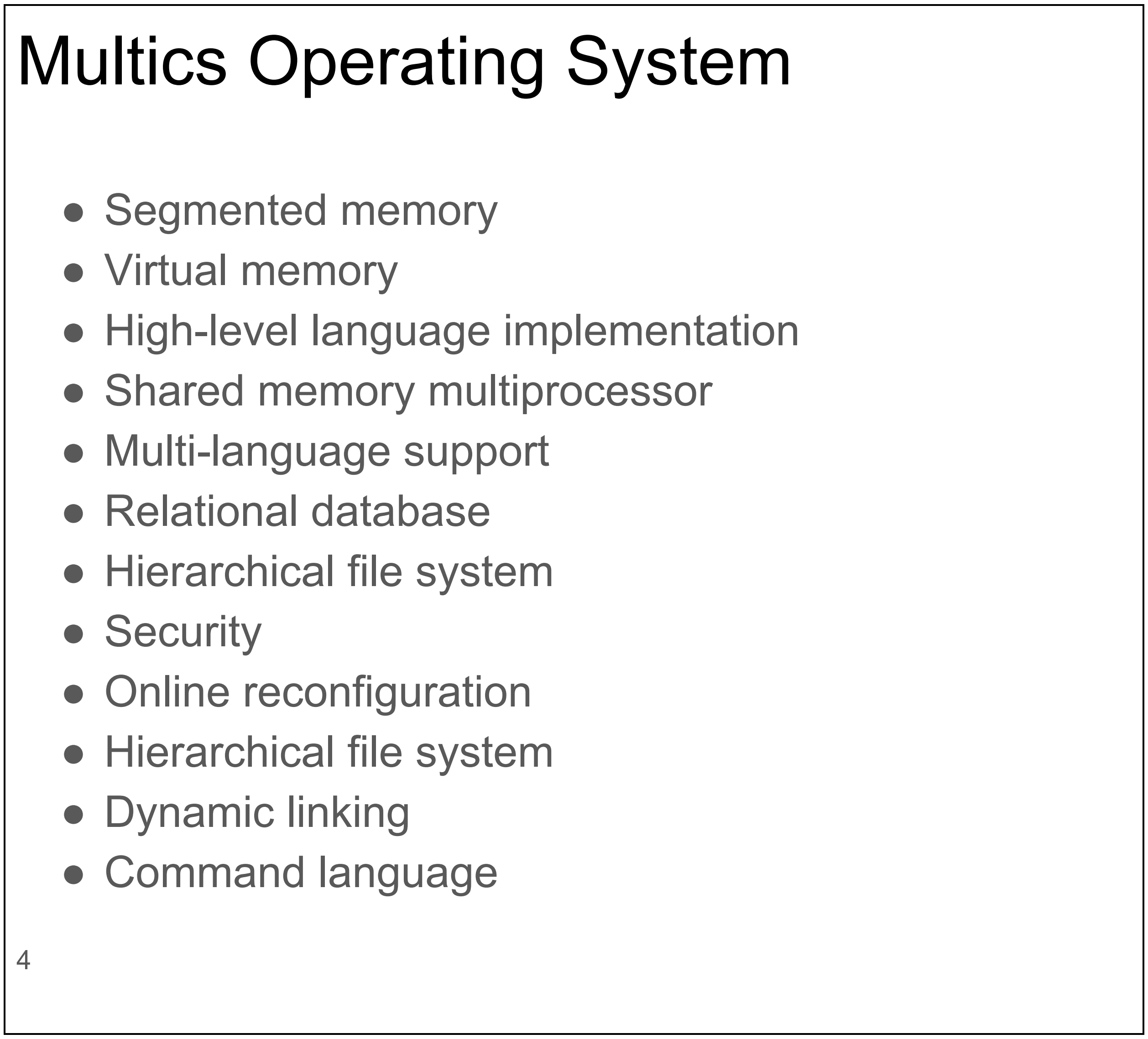
In the early 1960s, General Electric was shipping the GE-635 computer, and successfully bid a proposed extension to the 635 to meet these needs: the GE 645. The principal difference was the addition of the Append Unit, what we would now call “virtual memory”, but was more than that -- not just paging and virtual addressing, but also the hardware support of rings and segments and the security and capabilities that they provided.

In January 1967, the first 645 was delivered to the Multics developers, late.

In 1970, Honeywell purchased the General Electric Large Computer Systems division. The 635 continues on as the Honeywell 6080 and the 645 has improvements made and becomes comes the 6180, shipped in 1973.

A two-CPU system with 768KW of memory, 8MB of bulk store, 1.6GB of disk, 8 tape drives, and two DN355 serial line controllers, similar to MIT's system, had a purchase price of about $7 million; $38 million adjusted for inflation.

Improvements in semiconductor technology led to the DPS8/M, shipped in 1979. Larger capacity memory units allowed the reduction in the number of memory controller connectors from eight to four, the “blinking light” maintenance panels were replaced with a minicomputer based diagnostic system, and increased component density led to smaller and more power efficient systems.



Multics Operating System

- Segmented memory
- Virtual memory
- High-level language implementation
- Shared memory multiprocessor
- Multi-language support
- Relational database
- Hierarchical file system
- Security
- Online reconfiguration
- Hierarchical file system
- Dynamic linking
- Command language

The Multics Operating System.

The process address space is divided into segments, each segment with its own access controls, and the segmented address space is virtual with demand paging.

The bulk of the operating system is implemented in PL/I.

The hardware allows 8 CPU symmetric multi-processing.

PL/I, Assembly, COBOL, FORTRAN, BCPL, APL, BASIC, MACLISP, Pascal, C, and the first commercial relational database.

Multics was the first to provide an hierarchical file system. In addition: access control on individual files, long names, multiple names, symbolic links, storage quotas, removable devices and mandatory access controls. Multics introduced the idea of treating all devices uniformly as special files; it was the first OS to run on a symmetric multiprocessor, and the only general-purpose system ever to be awarded a B2 security rating by the NSA

Security designed in from the beginning, and enforced in the hardware.

Online reconfiguration: CPUs, Memory units, I/O controllers and peripherals could be added are removed from a running system. Several large sites would reconfigure their systems during off-peak hours by removing a CPU, a memory controller, and an I/O controller from the running production system, recabling those modules and booting test versions of Multics, and reversing the process at the return of peak hours, without interrupting the production system..

Dynamic linking is the norm; typically executables would only be statically bound when they were installed in production environments.

Command language: a shell, with search rules, I/O redirection, working directories. Multics did it first.

The Problem

Multics ran only on this hardware, and there are no remaining running instances of the hardware.

Solution 1: Port Multics to new hardware

Solution 2: Write an emulator

5

The Problem.

Multics ran only on this hardware; there were about one hundred installations, and none of them remain.

Port Multics to a modern architecture?

Multics was never intended to be portable; the thirty six bitness of the architecture, the source in PL/I and full of word size dependencies, and the lack of anything resembling the segments and rings of the native hardware in modern architecture makes this a daunting task.

That leaves the emulator route. What could go wrong?

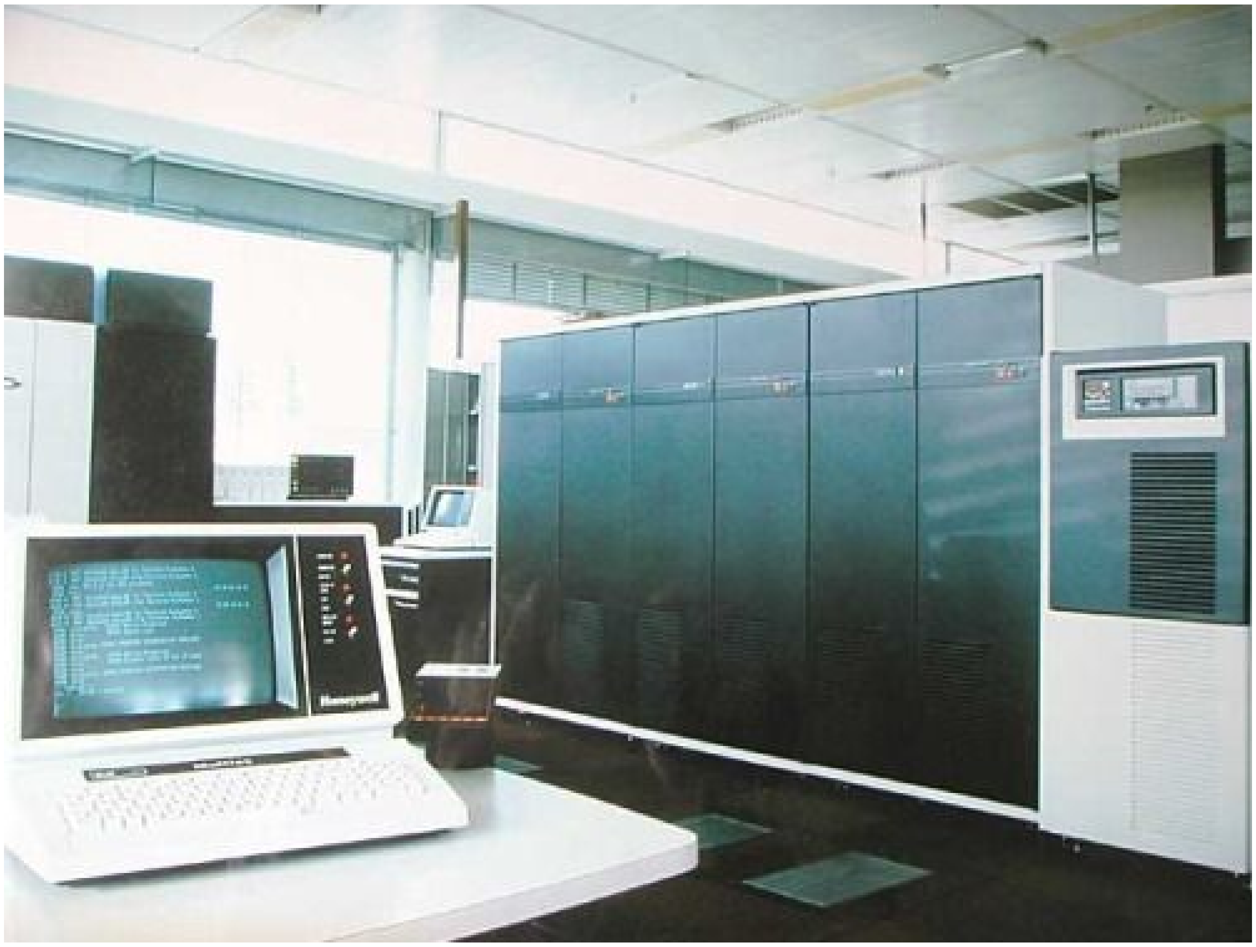
Multics Hardware

- 6180
- Level 68
- DPS8/M

6

The Multics hardware is a modular design; each module taking up a one or two full size cabinets.

Honey


Honeywell

Multics Hardware

36 bit word, 18 bit addresses, 15 bit segment numbers

Modular design

- CPU Central Processing Unit
- SCU System Control Unit: memory, interrupt management
- IOM Input/Output Manager: 56 controller channels
- Disks, drums, tapes, printers, card readers/punches
- FNP Front end Network Processor
- ABSI IMP interface
- Operators console

10

The CPUs run at about one MIP; instructions were 36 bit wide, with an 18 bit operand, allowing a segment size of 256 thousand words. Segment numbers are 15 bits, allowing 32 thousand segments of 256 thousand words each, or about a theoretical 8 gigabytes of address space per process; Multics table sizes and backing store sizes set a smaller limit on the size.

Two thirty-six bit accumulators, A and Q. Eight eighteen bit index registers, X0-X7. Eight pointer/address registers, each containing a fifteen bit segment number, eighteen bit address and six bit bit number.

An elaborate indirect addressing scheme, allowing chained indirection, tally decrements and address increments, register indexing and segment crossing. (An interesting side note: the no-operation instruction includes an operand address and address preparation is performed, which includes walking the indirection chain; this means that the no-operation instruction can alter memory, page fault, and incur access violation faults).

The CPU does thirty six and seventy-two bit integer and floating point arithmetic, ten digit decimal arithmetic, string move and compare, and decimal number formatting (as in COBOL and PL/I “PIC”).

The CPU operates in two modes, absolute mode with memory access to the low 256K of memory and privileged instructions and append mode, with segmented memory access.

Physical memory addresses are twenty four bits, allowing sixteen megawords of physical memory.

Memory is contained in the SCU modules; late model SCUs contained up to four megawords of memory, meaning four SCUs were needed for a full memory system.

The I/O controllers are attached to the SCUs; all I/O is “DMA”; the CPU had only a single I/O command “connect to I/O controller” with an operand specifying which I/O controller to signal.

The I/O controllers send interrupt signals to the SCUs; they contain the logic and mask registers that control the forwarding of the interrupt to a particular CPU.

The FNPs managed serial communication lines; asynchronous, bi-sync, X.25 and others. Depending on the model and configuration, up to ninety-six high speed asynchronous lines could be managed; eight FNPs were allowed, so potentially 768 lines per system.

Multics Terminology

Segment

Ring

Some quick Multics terminology.

“Segments” are central to both the hardware and Multics, but are semantically different between the two, which confused me no end for a while.

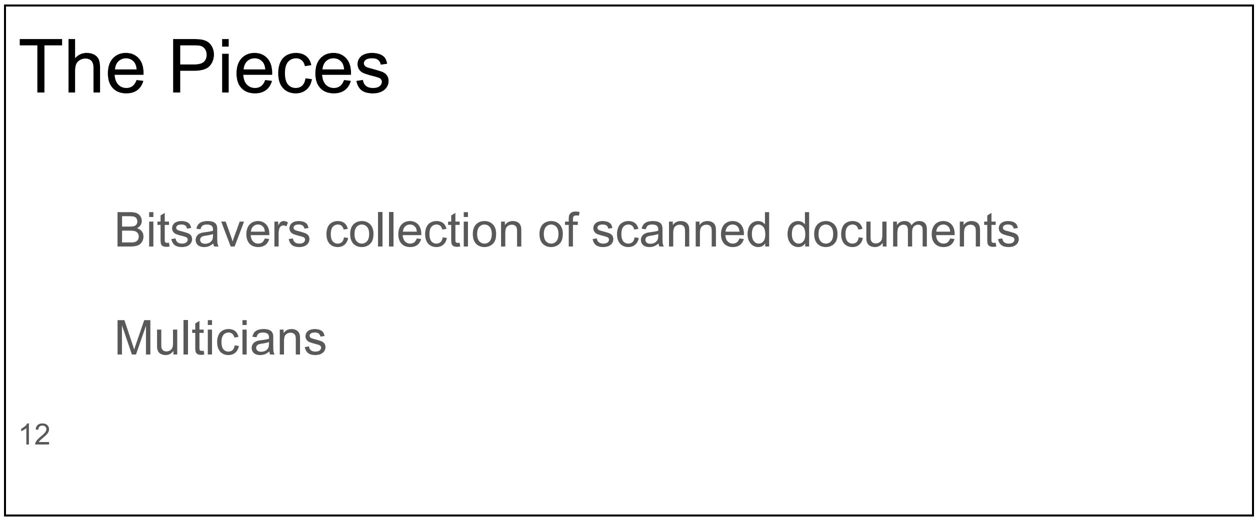
To the hardware, a segment is an entry in the Segment Descriptor table, indexed by a fifteen bit segment number. The entry is a page count and a list of 1024 word pages comprising the 128K word segment address space; for each page listed, a page present bit, a page modified bit, and a page aligned 24 bit physical address if the page is present. A virtual memory unit with an extra level for segment indexing.

The CPU has a Process Segment Register containing the segment number of the current process and a Temporary Segment Register used when referencing out-of-segment. The Pointer Registers each include a segment number field and indirect addresses may include a segment number field.

Multics defines segments as named, persistent storage. The process of ‘initiating’ maps a named segment into the process address space by assigning a segment number to it; somewhat analogous to a DLL load returning the address that the library was loaded at.

“Rings” are part of the Multics security structure. The innermost code -- interrupt handlers, process scheduler, resource allocation runs in ring zero; Multics code that does not need privileged hardware access in ring one, user processes in ring four, limited access processes in ring five.

Each segment has ring bracket registers that control read, write and execute access from the specified ranges of rings; this access is evaluated by the hardware when an address crosses rings; violations are rejected.



The Pieces

Bitsavers collection of scanned documents

Multicians

12

The pieces available to build an emulator.

What resources are available to the emulator writers?

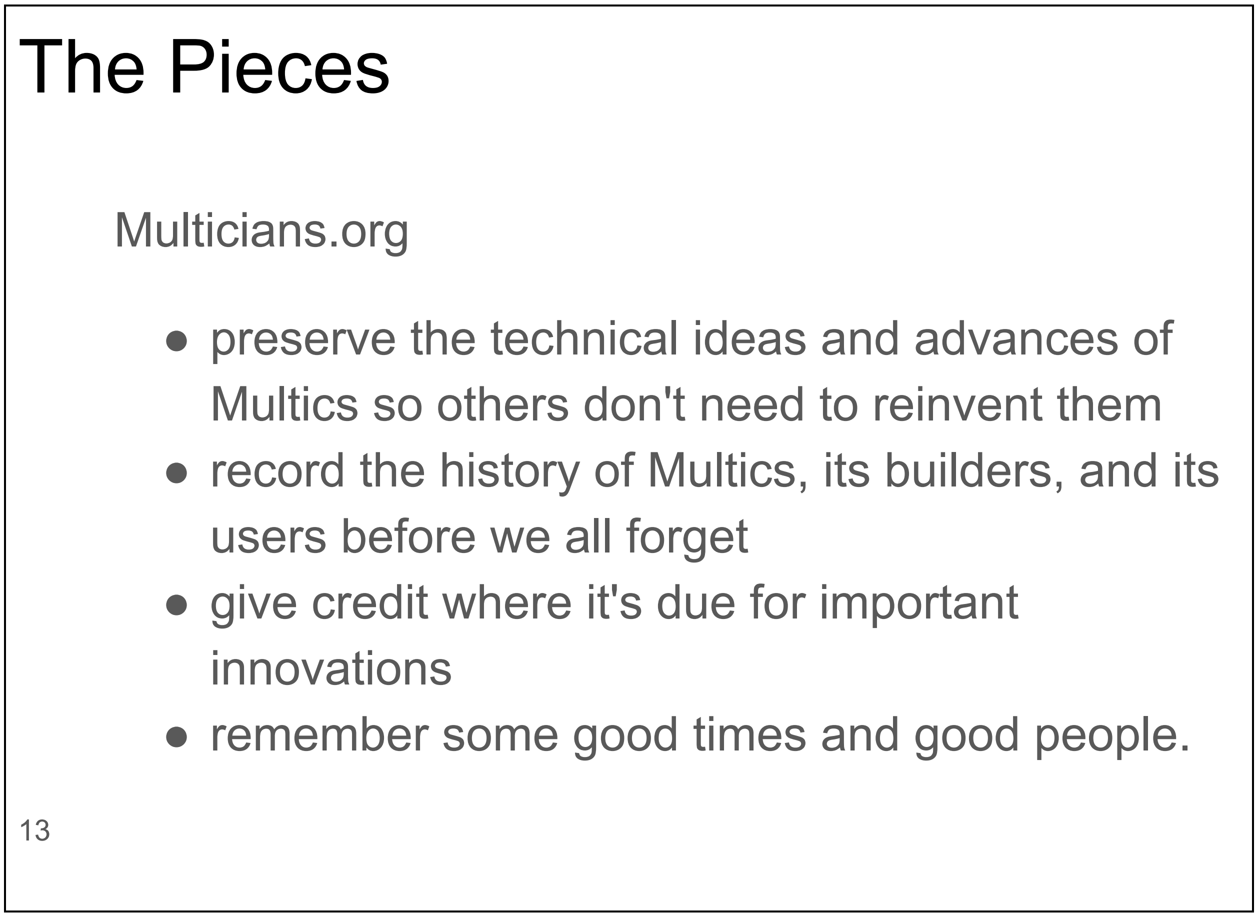
Bitsavers provided a treasure trove of documentation; the most important pieces being the “MULTICS Processor Manual”, “MULTICS Differences Manual”, “Hardware/Software Formats Programming Logic Manual” and “6000B Engineering Product Specification 1”

The “MULTICS Processor Manual”, known as AL39, defines the machine language -- the instruction set, registers, addressing modes, virtual memory…. Mostly.

The “differences” manual describes the changes made for the Level 68 to DPS8/M transition and provides insight into the underlying structure of the system. The Hardware/Software Formats manual defines the bit layout of many of the registers and data transfer words, with an all too brief definition of the bits. The I/O multiplexer manual defines the I/O controller and its interface to the SCU and the channel controllers, again mostly.

The GE 635 computer that the Multics hardware was evolved from continued on as the Honeywell 6080, Level 66, DPS8, and eventually as the NovaScale 9000 emulator, which is an active commercial product today, running legacy GECOS applications. Bull publishes RJ78, “NovaScale 9000 Assembly”, which describes a machine that is a close cousin of the DPS8/M and was help in covering some of the missing detail in AL39.

Multician: “Anyone who contributed to the development and success of Multics. There is no such thing as an ex-Multician.”



The Pieces

Multicians.org

- preserve the technical ideas and advances of Multics so others don't need to reinvent them
- record the history of Multics, its builders, and its users before we all forget
- give credit where it's due for important innovations
- remember some good times and good people.

13

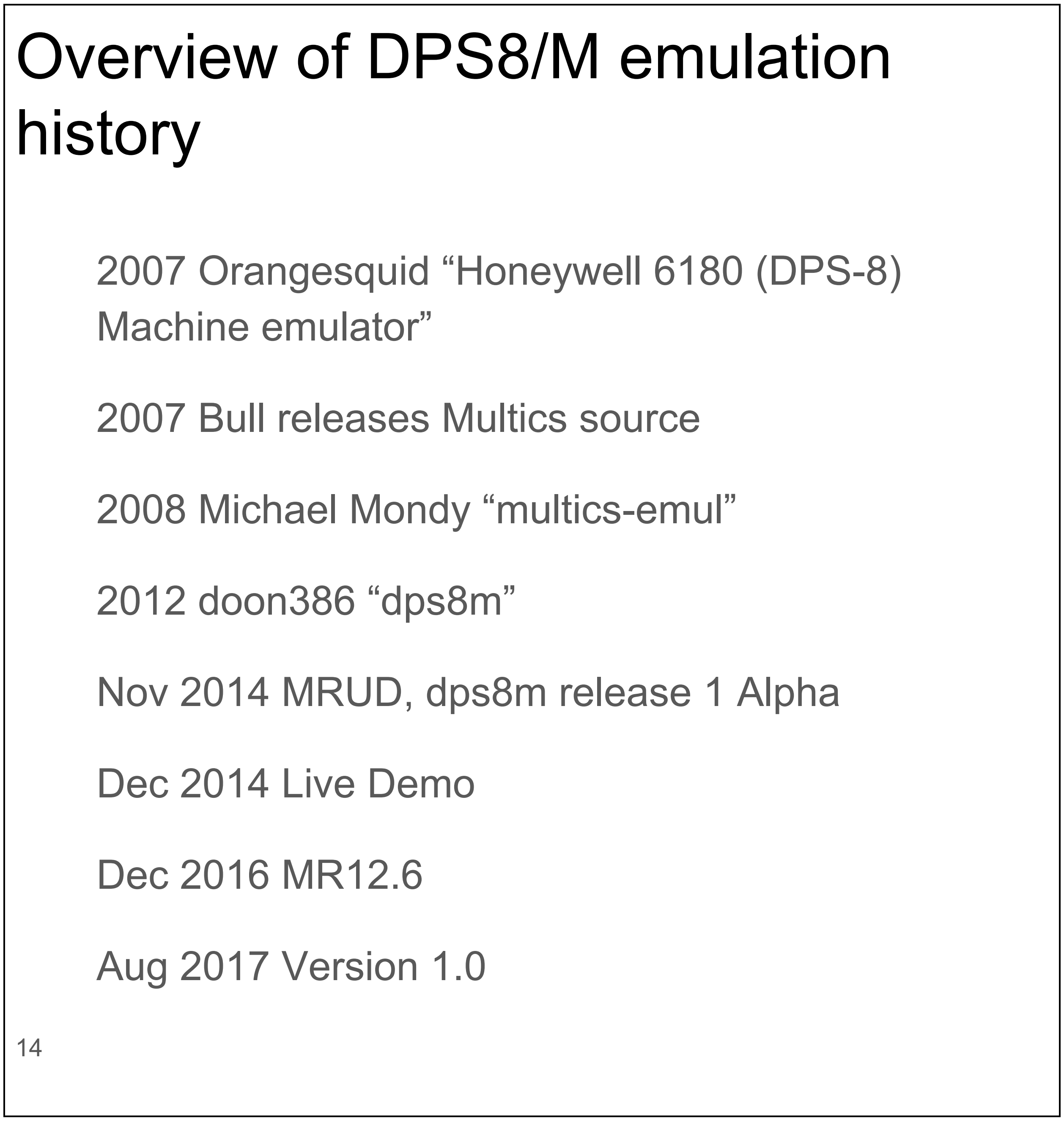
“The Multicians.org web site presents the story of the Multics operating system for people interested in the system's history, especially Multicians.”

The site's goals are to

- preserve the technical ideas and advances of Multics so others don't need to reinvent them.
- record the history of Multics, its builders, and its users before we all forget.
- give credit where it's due for important innovations.
- remember some good times and good people.

Multicians.org also runs a mailing list which they kindly allowed Harry Reed and I to join and ask stupid questions on, providing access to the people who really knew what was going on.

I was recently speaking to a Multician who vouchsafed that the Multicians had discussed the possibility of an emulator and had come to the conclusion that it was an impossible project, and that we were not expected to succeed.



Overview of DPS8/M emulation history

2007 Orangesquid “Honeywell 6180 (DPS-8) Machine emulator”

2007 Bull releases Multics source

2008 Michael Mondy “multics-emul”

2012 doon386 “dps8m”

Nov 2014 MRUD, dps8m release 1 Alpha

Dec 2014 Live Demo

Dec 2016 MR12.6

Aug 2017 Version 1.0

14

An overview of DP8S/M emulation history.

In 2007, Orangesquid created a SourceForge project; last updated in 2010. Basic thirty-six bit arithmetic functions implemented, no append unit or tape I/O.

In late 2007, Bull released the source to Multics Release 12.3 and the 12.5 update. With the sources, writing an emulator that could run Multics became feasible; Multics was no longer an arbitrary string of bits. When an emulator bug caused a Multics malfunction, it would be possible to understand what Multics was trying to accomplish at that point, and from that an understanding of what the emulator did wrong (hopefully with a reasonable amount of effort).

In early 2008, Michael Mondy created a Github “multics-emul” project and worked on it thorough 2012. The code successfully reads the boot tape boot record, the first stage bootloader and early portions of the boot code; completing about two million emulated instructions. The emulation fails at the first page fault; the fault handling code was not completed.

Quoting Mr. Mondy:

> “I started from a new code base. However, I would not have attempted to write this without having the foundation laid by the Multics community. The programming isn't hard. The hard part is getting specifications and descriptions and figuring out what the hardware is really supposed to do. In particular, discussions of the boot process, a working boot tape, and source to the boot code are so invaluable that I would not want to try this without them. Doing an IOM without having the boot tape as a sort of behavioral description would be impossible.”

Mr. Mondy adopted the SIMH emulator framework, an excellent platform for developing emulators on; it provides a debugging toolset, event scheduling, an extensible control and configuration interpreter, serial port emulation, platform portability and much more.

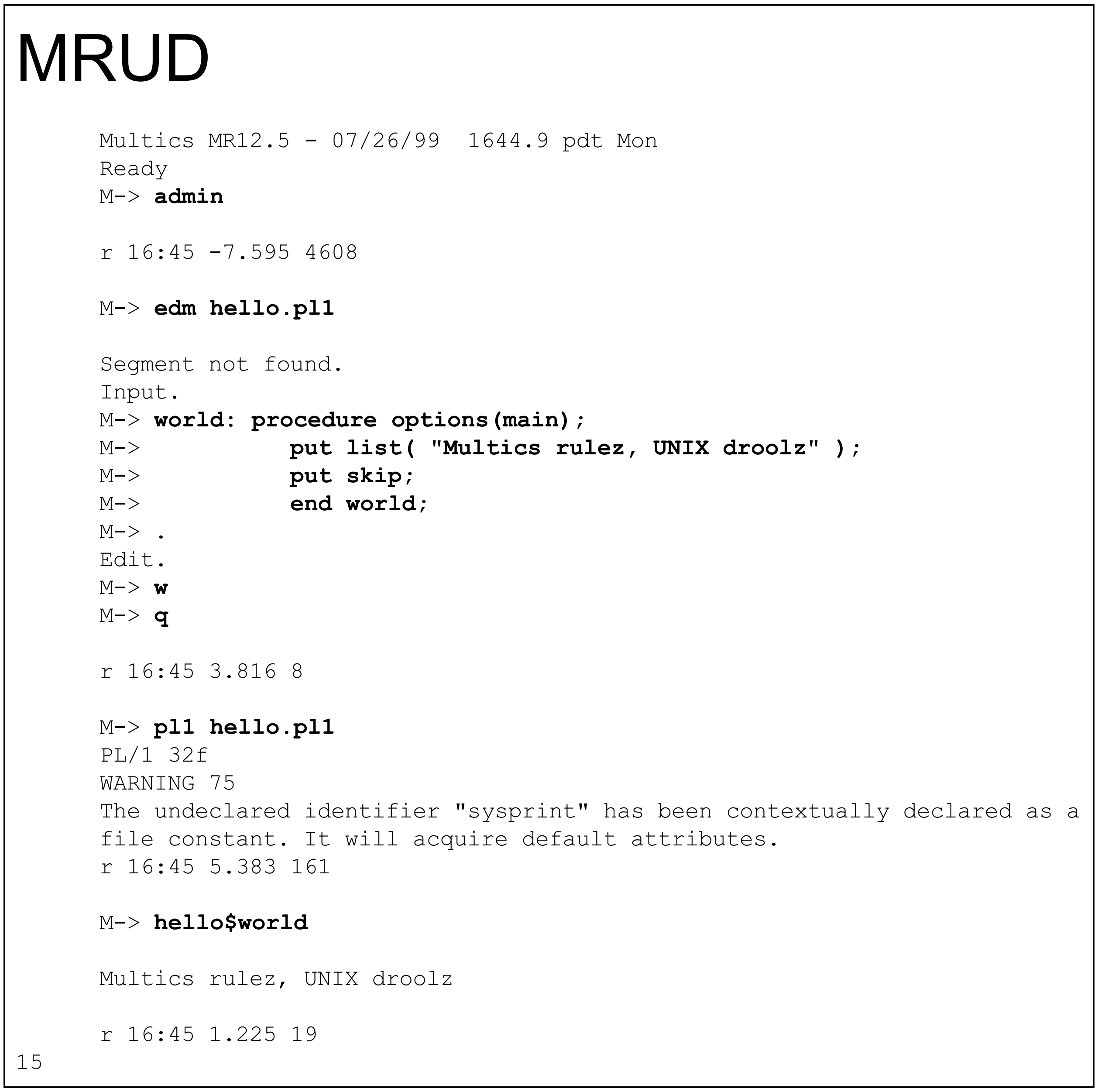
In 2013, Harry Reed created the SourceForge project “dps8m”, uploading work that he started in 2012.

From the project’s SourceForge page:

> “Hello all. Well, as you may have guessed another hapless individual has begun the task of resurrecting Multics. Granted, several others of more capability and valor than myself have attempted to create a simulator for the dps8m and have run screaming from the mere complexity of the task - I have, nevertheless, decided to go where angels fear to tread and attempt to write a simh based simulator capable of booting Multics”

He merged much of Mondy’s code base, but completely re-architectured the emulated CPU code to better follow the hardware architecture. He also wrote a cross assembler, enabling the creation of DPS8/M assembly language unit tests and started building FXE, the “Faux Execution Environment”, wherein Multics applications could be run in the emulator, with the FXE servicing the application’s system calls, making the application believe that Multics is running; somewhat akin to WINE.

Nov 9. 2014: A “hello, world” program is typed in, compiled and run.



MRUD

```
Multics MR12.5 - 07/26/99  1644.9 pdt Mon
Ready
M-> admin

r 16:45 -7.595 4608

M-> edm hello.pl1

Segment not found.
Input.
M-> world: procedure options(main);
M->          put list( "Multics rulez, UNIX droolz" );
M->          put skip;
M->          end world;
M-> .
Edit.
M-> w
M-> q

r 16:45 3.816 8

M-> pl1 hello.pl1
PL/1 32f
WARNING 75
The undeclared identifier "sysprint" has been contextually declared as a
file constant. It will acquire default attributes.
r 16:45 5.383 161

M-> hello$world

Multics rulez, UNIX droolz

r 16:45 1.225 19
```

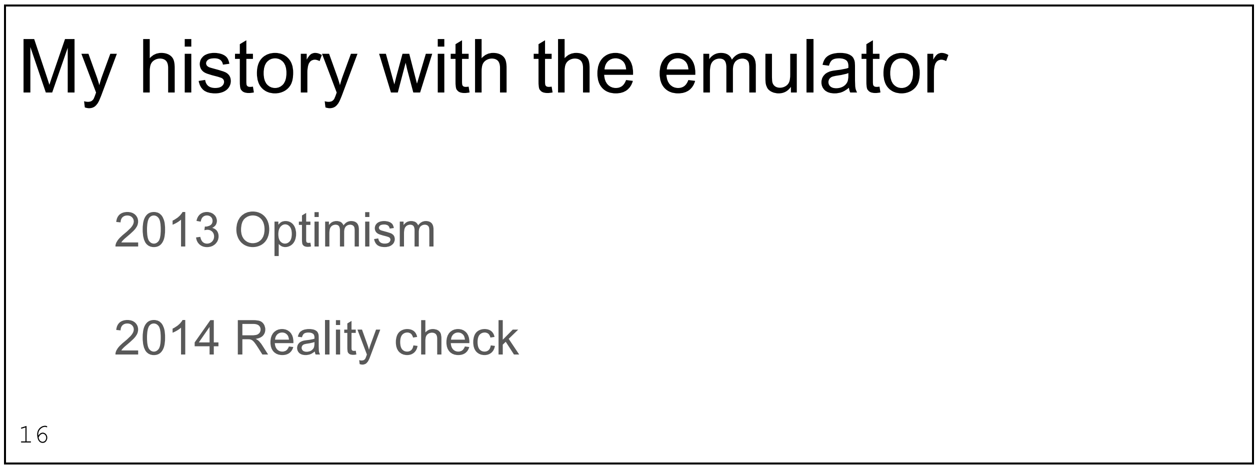
15

Shortly after this the first “alpha” release was pushed out. Single user mode only, look at it cross-eyed and it crashed and many, many things that just did not work. But a release.

On Dec 10, 2014, Multician Olin Sibert was the moderating the Distinguished Practitioner panel at the Applied Computer Security Applications Conference; he took the opportunity to demonstrate the DPS8/M emulator to an audience of about 400 computer security professionals and students.

In late 2016, another team fixed a small number of Multics Y2K issues and generated Multics Release 12.6.

And in August 2017, version 1.0 of the emulator was released.



My history with the emulator

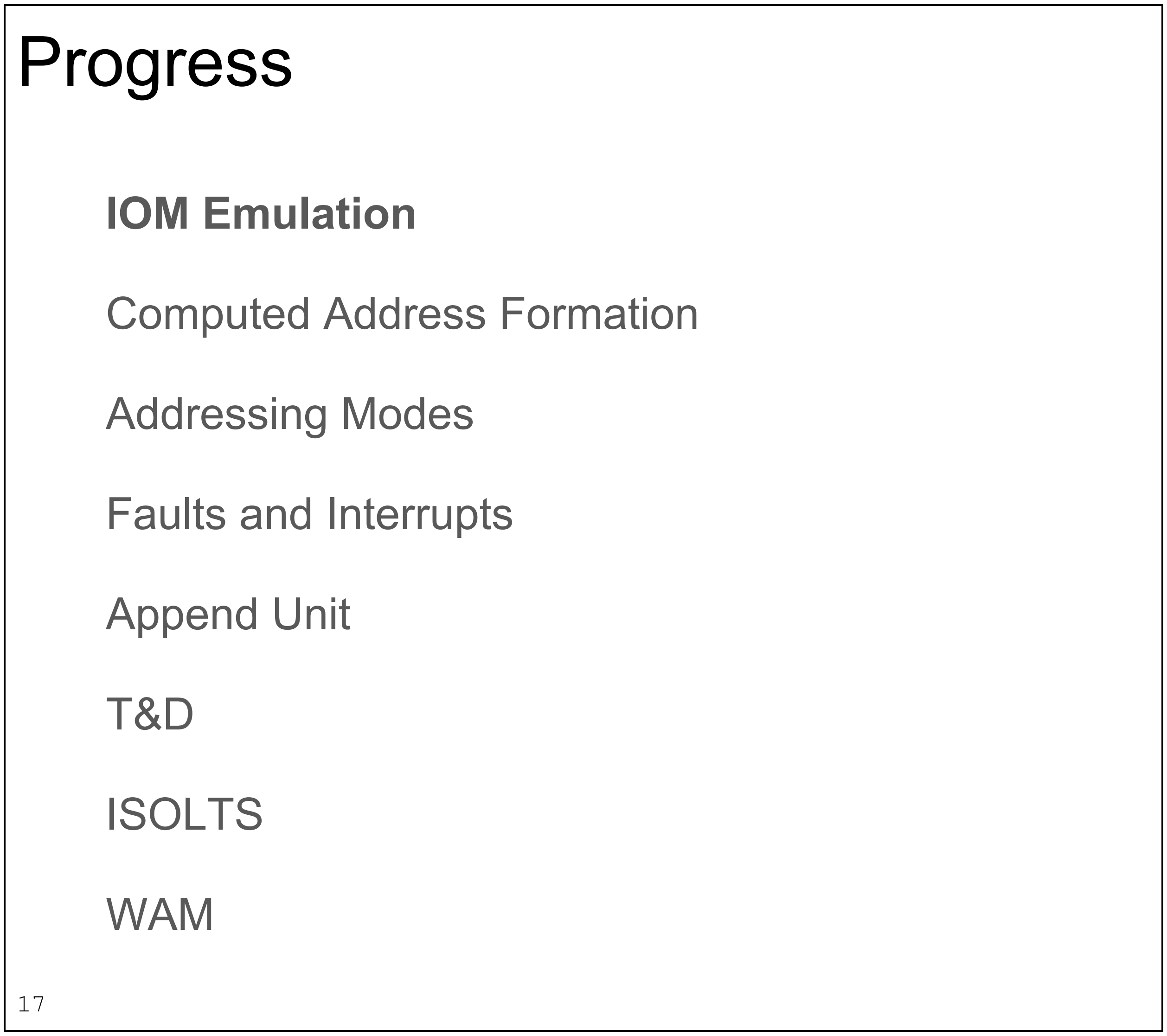
2013 Optimism

2014 Reality check

Sometime in late 2013. I was searching the web looking for a PL/I compiler for some project that I have long forgotten about. Multics had a PL/I compiler, and if an emulator were made to work, I would have a PL/I compiler. Although I had no DPS8/M or Multics experience, I did have a lot of mainframe experience (CDC 6000, IBM 360) and had written or worked on several emulators over the years. I took a look at the code and at AL39 and thought that yeah, a couple of months, no problem. Let’s get this puppy whipped into shape. Right….

At that time, the emulator was capable of executing the first few hundred boot tape instructions; we would look at the bootloader source code, AL39 and emulator instruction traces; figure out what was wrong, fix the emulator, repeat.

By early 2014, it was beginning to dawn on me that there was a fundamental problem: we had enough documentation to make the project possible, but so much information was missing that it was going to be very, very hard.



Progress

IOM Emulation

Computed Address Formation

Addressing Modes

Faults and Interrupts

Append Unit

T&D

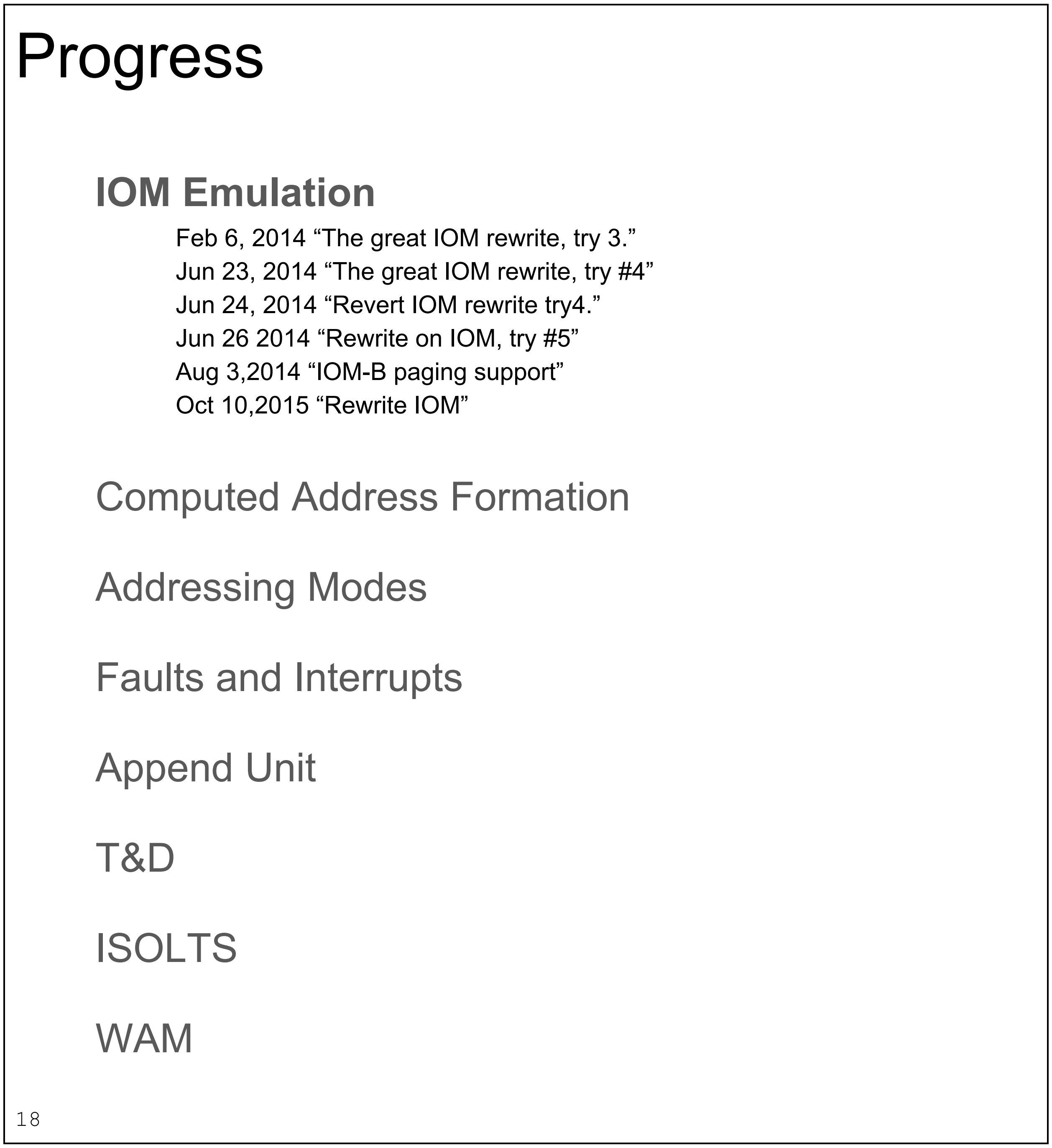
ISOLTS

WAM

17

DPS8/M Emulator progress.

Harry was making good progress on the CPU instruction set; I had experience in device drivers, so I tackled the I/O controller (IOM) emulation. The code had a lot of “just for now” bits in it, many of which dealt with eliding unit numbers and configuration concerns. I started chasing them down and realizing that we needed to be able model the modular nature of the hardware, added the “cable” command to simh, emulating the physical cable interconnections. This allowed the specification of what module connects to what module on which ports. (The current release has eighty cable commands in its default configuration.) Harry became quite irritated with me when his code stopped working, but forgave me when he read my check-in log note: “IOM now working well enough that correct configuration matters.” The IOM now could read tapes and interact with the operator console for any channel assignments or IOM unit numbers, but this would not the last time I was in the IOM code.



Progress

IOM Emulation

Feb 6, 2014 “The great IOM rewrite, try 3.”
Jun 23, 2014 “The great IOM rewrite, try #4”
Jun 24, 2014 “Revert IOM rewrite try4.”
Jun 26 2014 “Rewrite on IOM, try #5”
Aug 3,2014 “IOM-B paging support”
Oct 10,2015 “Rewrite IOM”

Computed Address Formation

Addressing Modes

Faults and Interrupts

Append Unit

T&D

ISOLTS

WAM

Looking over the git log, we see:

Feb 6, 2014 “The great IOM rewrite, try 3.”
Jun 23, 2014 “The great IOM rewrite, try #4”
Jun 24, 2014 “Revert IOM rewrite try4.”

Jun 26 2014 “Rewrite on IOM, try #5”
Aug 3,2014 “IOM-B paging support”
Oct 10,2015 “Rewrite IOM”

The ‘IOM-B paging support” was pretty special for me. Multics was passing channel control programs to the IOM that the IOM was unable to parse; there were bits set in the channel program that were defined to be zero. Perplexed, I was rereading the IOM EPS-1 from top to bottom over and over again, trying to suss out what was going on. For some definition of bottom. It dawned on me the ‘Appendix 1” of the document was not, as I had thought, describing an irrelevant late model functionality, but rather the functionality that Multics was expecting. Appendix 1 described extensive modifications to the IOM to allow it access to the page tables of the CPU, enabling I/O to the process’s address space as well as kernel space. So, a couple of hundred more lines of code for the IOM emulation.

Progress

IOM Emulation

Computed Address Formation

Addressing Modes

Faults and Interrupts

Append Unit

T&D

ISOLTS

WAM

Computed address formation is the operand address preparation phase of instruction execution, covering about twenty three pages of AL39. CAF is driven by an eighteen bit operand address, a one bit ‘use a pointer register’ flag and a six bit ‘Address Modification Field’

Address Modifier Field

6 bits controlling the interpretation of the operand field

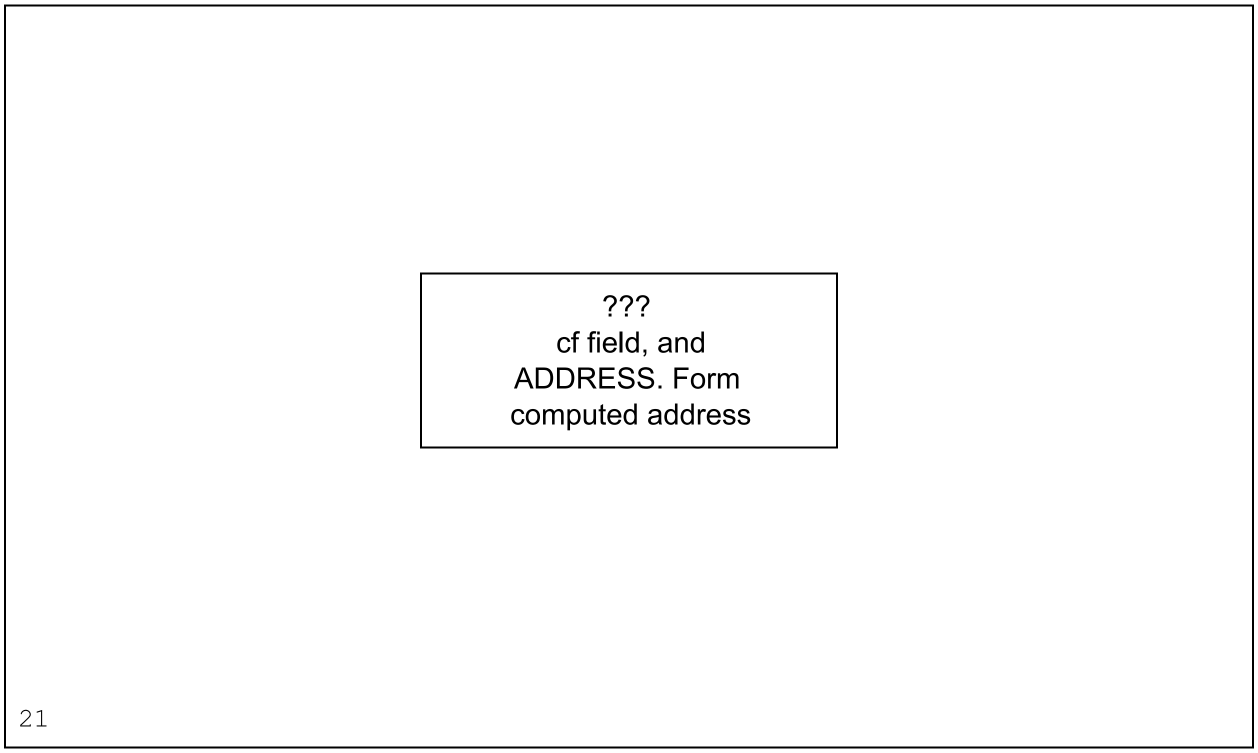
- N: No modification; the operand is the address
- AU: Add the upper half of the A register
- QU: Add the upper half of the Q register
- DU: The operand is the upper half of a literal operand
- IC: Add the instruction counter
- AL: Add the lower half of the A register
- QL: Add the lower half of the Q register
- DL: The operand is the lower half of a literal operand
- X0-X7: Add the indicated index register
- RI: Do the above calculation; the result is a pointer to a descriptor containing new operand and modifier fields; fetch the word and repeat from the top.
- IR: Remember the above modification but do not execute it until the end of the indirection chain is reached; the operand points to a descriptor; fetch and repeat from the top.
- F1: Do a page fault with an F1 tag
- SD: Subtract Delta; the word pointed to by the operand contains address, tally, and delta fields. Increment the tally and subtract delta from the address, the new address is the operand address
- F2: Do a page fault with an F2 tag
- CI: Character Indirect: the word pointed to by the operand contains address, character size and character offset fields
- I: The operand points to a descriptor; fetch and repeat from the top
- SC: Sequence Character; like Character Indirect, but the address is incremented and the tally decremented.
- AD: Add Delta; like SD -- the tally is decremented and delta is added to the address
- DI: Decrement address, increment tally
- DIC: Decrement address, increment tally and continue; address points to a descriptor, fetch and repeat from the top
- ID: Increment address, decrement tally
- IDC: Increment address, decrement tally and continue; address points to a descriptor, fetch and repeat from the top
- ITP; The address points do a two-word descriptor containing a pointer register number, an address, a bit number and a new address modification field; the named pointer register is combined with the descriptor data to form an segment number and address in that segment; the new address modification field is loaded, and repeat from the top
- ITS : The address points do a two-word descriptor containing a segment number, an address, a bit number and a new address modification field; the the descriptor data is used form an segment number and address in that segment; the new address modification field is loaded, and repeat from the top

A six bit tag gives 32 possible addressing modes; I am not going to go over these; I just want to make clear that operand address calculation is complex and convoluted.

The important bits are that length of the descriptor chain is unbounded, and that the processing of a descriptor can have side effects: modifying address and tally fields. This make page fault handling problematic; consider the case where the indirection chain includes an increment address, decrement tally and continue descriptor, and a later descriptor causes a page fault. On instruction restart, the address preparation cannot start from the original address as that would cause the increment and decrement cycles to be repeated, resulting in incorrect address calculation.

Frustratingly, the interaction of page faulting is completely elided in AL39, but is absolutely critical to correct operation.

AL39 spends only six pages on the address modification field, so details are a bit thin on the ground. For example, the IT MOD flowchart contains an node with this helpful gem:



The computed address formation code is is running at about 1600 lines of C code, I am still not convinced that that it handles all restarts or segment crossings correctly.

Harry and I spent many months debating the edge cases and stomping over each others changes; this code probably bought us as close to dysfunction as we ever got.

The outstanding fundamental implementation issue revolves around the interaction of the computed address formation and virtual address processing. Virtual address processing is done by the Append Unit; it is a state machine.

It has statements like “was the last cycle an indirect word fetch?”; this means correctly invoking the append unit is as important as its correct implementation, yet invocation is completely undocumented. When computed address formation is updating address and tally field, what cycle type should it use? Using the incorrect cycle type could cause the append unit to reset the segment number, causing the operand to address the incorrect segment.

We have reverse engineered cycle usage by tinkering with it until Multics runs, but I sure that there exists untested code paths that are just wrong.

Progress

- IOM Emulation
- Computed Address Formation
- **Addressing Modes**
  - Absolute
  - BAR
  - Append
- Faults and Interrupts
- Append Unit
- T&D
- ISOLTS
- WAM

AI39 states: “There are three modes of main memory addressing (absolute mode, append mode, and BAR mode).”

It lies; there are four.

BAR mode is a base and bound relocation; a region of memory starting and address “base” of length “bound” is treated as if it starts at address zero.

Append mode is segment addressing; address are composed of fifteen bit segment numbers and an eighteen bit offset in the segment.

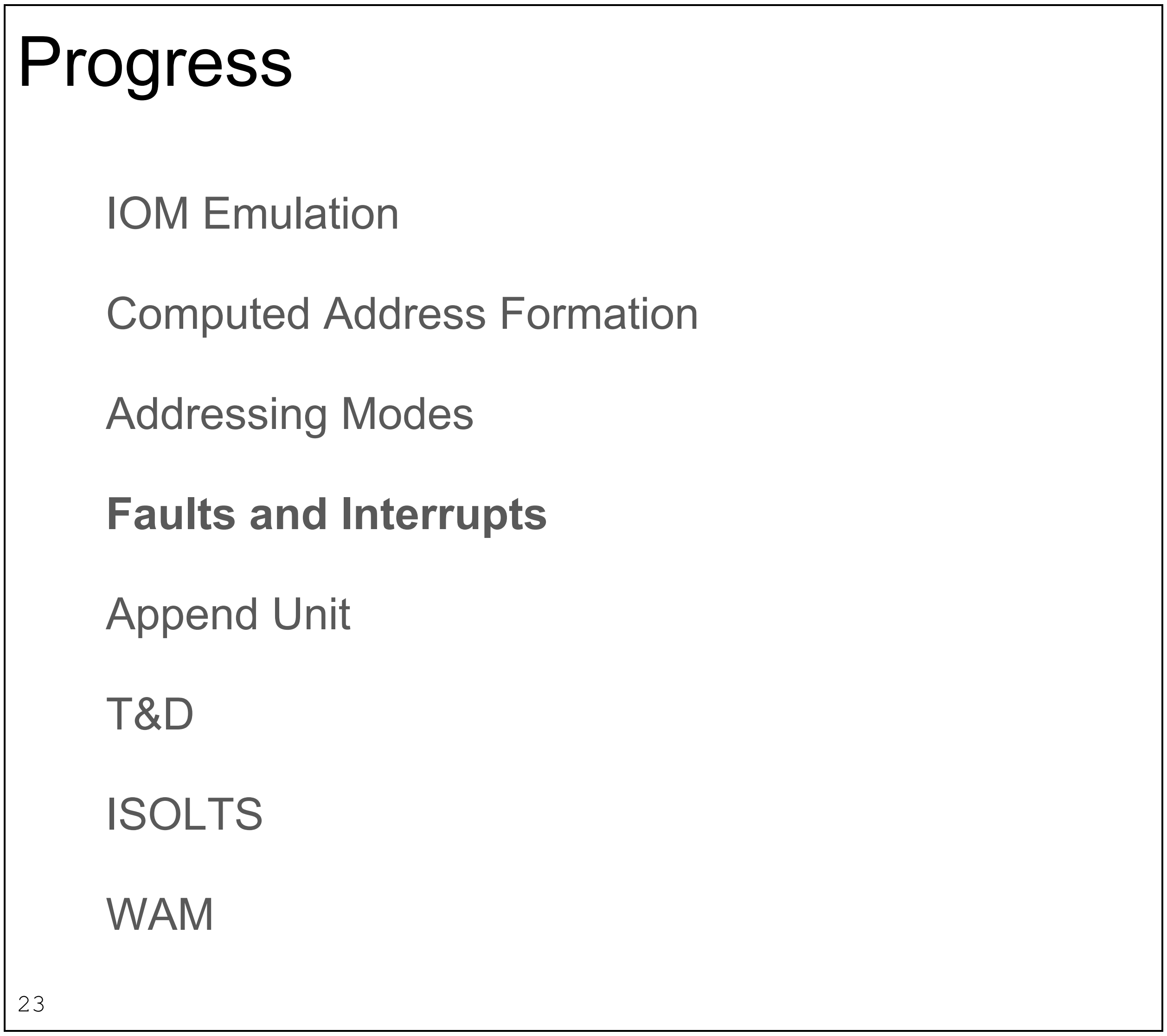
Append mode and BAR mode are independent, so four modes: Absolute, Absolute BAR,, Append, and Append BAR modes.

In Append BAR mode, a region of memory in a segment is treated as if it starts at offset zero in the segment.

Multics includes the GTSS subsystem, a GCOS simulator that allows GCOS applications to run under Multics without recompilation. GCOS applications run in BAR mode on the 635, so to enable the simulation, GTSS runs the application in a segment with BAR mode enabled; getting this to work in the emulator when the “three modes” meme of AL39 was present from the beginning caused a lot of grief.

The CPU maintains an eighteen bit instruction counter and a PSR register containing a 15 bit segment number.

In absolute mode (not appending, not BAR), instruction and operand addresses are 18 bits and address the low 256K words of memory. In append mode, the 18 bit address and 15 bit segment number are used to lookup a 24 bit address in the Descriptor and Page tables.



Progress

IOM Emulation

Computed Address Formation

Addressing Modes

Faults and Interrupts

Append Unit

T&D

ISOLTS

WAM

23

By late December of 2014, I was becoming aware that the CPU design was not capable of correctly managing page faults. The ability to restore the CPU state allowing instructions to resume where they left off after the fault was serviced was just not feasible.

As an example, the code that implemented the XEC and XED instructions (interpret the operand as an instruction or instruction pair and execute it) was implemented as a recursive call into the instruction execution routine; if the operand instruction execution encountered a restartable fault, there was no was to track that recursion in the available fault data space.

I rewrote the CPU code as a state machine, leading to a long period of loss of functionality and stability; I fixed a bug in that code as recently as September 2017.

The implementation has evolved; currently the states are instruction fetch, execute, fault, fault execute, interrupt, interrupt execute and fault return.

The emulator's CPU state machine has setjmp at the entry to the state switch logic. Access violations and page faults are implemented as longjmps back to the state machine rather than passing a fault return code back up the call chain.

The fault data space is an eight word data structure that is saved by a fault or interrupt and used to restore the CPU to the state that it was at when the fault occured. The saved data includes the two word Instruction Working Buffer, which contains the instruction pair that was being executed as the time of the fault. It also contains a handful of bits about the 'execute instruction' (XEC) and 'execute double instruction' (XED) instruction state; these instructions load their operands into the IWB (overwriting the XEC or XED instruction) to be executed, and sets various CPU control bits indicating that an XEC or XED is in progress. When the instruction is restarted, those bits are restored, allowing the CPU correctly continue the XEC/XED sequence.

There are also bits in the data structure indicating

- Repeat instructions (RPT, RPD and RPL) state
- Interrupt or fault flag
- Interrupt number
- Twenty fault status bits
- Append unit state
- Computed address formation state
- The XSF flag -- (there is only one reference to this flag in the documentation and Multics source, and no definition)

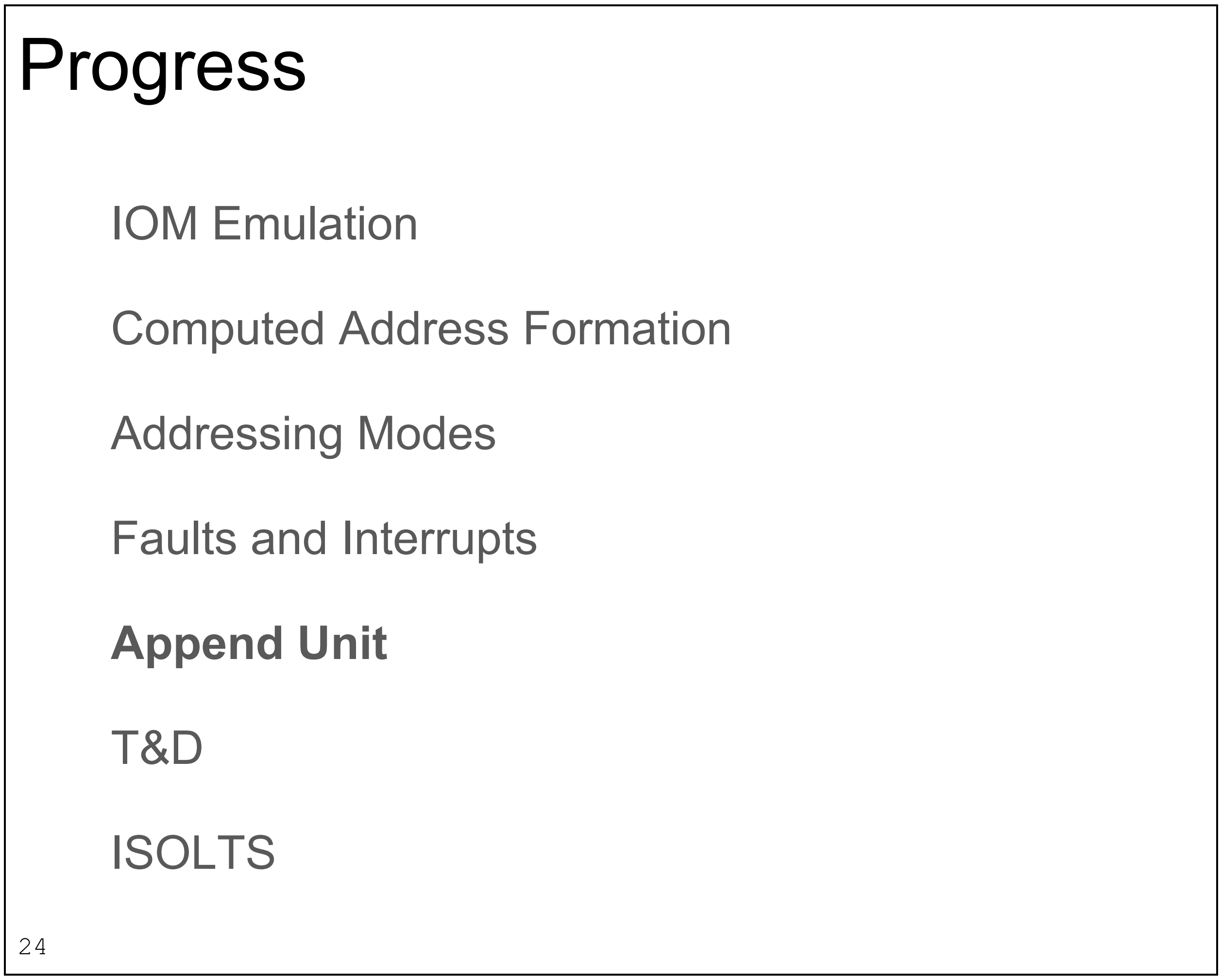
Again, the dearth of definitive documentation of the data has led to much heartache. One would hope that as long as the save and restore implementations were symmetrical in the emulator that all would be well, but Multics sometimes modifies the saved data before restarting the instruction so there is little room for loose interpretation of the semantics.

The XSF flag is particularly irksome. The documentation for the fault data space says "word zero, bit nineteen: XSF External Segment Flag". The strings "XSF" or "External Segment Flag" appear nowhere else in the documentation or the Multics source code.

From reading the Multics source code, it was deduced that the RFI bit (Restart this instruction) was set by the operating system, not the hardware, and was used to indicate if the instruction was to be resumed where it left off or restarted from the beginning, in the opinion of the operating system.

Another interesting oddity is the Master Mode Entry instructions; apparently intended as the instructions used by applications to make operating system requests; I knew them as “trap” instructions. They generate a MME fault; the operating system services the fault, performs the indicated request, and returns to the application. Multics does not use the instructions, but GCOS and therefore the GTSS GCOS simulator does.

In every other architecture that I have had experience with, the trap instruction resumes with the next instruction after the trap, analogous to a subroutine call instruction. On the DPS8/M, the instruction resumes back to the same instruction that trapped; it is the operating system’s responsibility to increment the return address in the fault data to perform a ‘return from trap’ functionality.



Progress

IOM Emulation

Computed Address Formation

Addressing Modes

Faults and Interrupts

Append Unit

T&D

ISOLTS

24

The append unit is the heart of the Multics system.

It is responsible for mapping virtual segment addresses to physical, tracking which segment is being addressed, triggering page faults, tracking page modification, and verifying access rights.

But we are not sure how it works.

There is a nine page flowchart of the Append Cycle, attesting to its complexity. (A quick google for “VAX virtual memory flowchart” images turned up single page flowcharts.)

Sadly, it is both incomplete and wrong, and I believe that the emulator is still incorrect.

An egregious typo was discovered recently; the incorrect ring register was being used for write permission checking, and it turned out that memory write protection actually didn’t work in the emulator; it was perfectly acceptable for user mode programs to write to any segment visible to it, including those belonging to the operating system. This typo is in every version of the append cycle flow chart that I can find in the manuals. Oops.

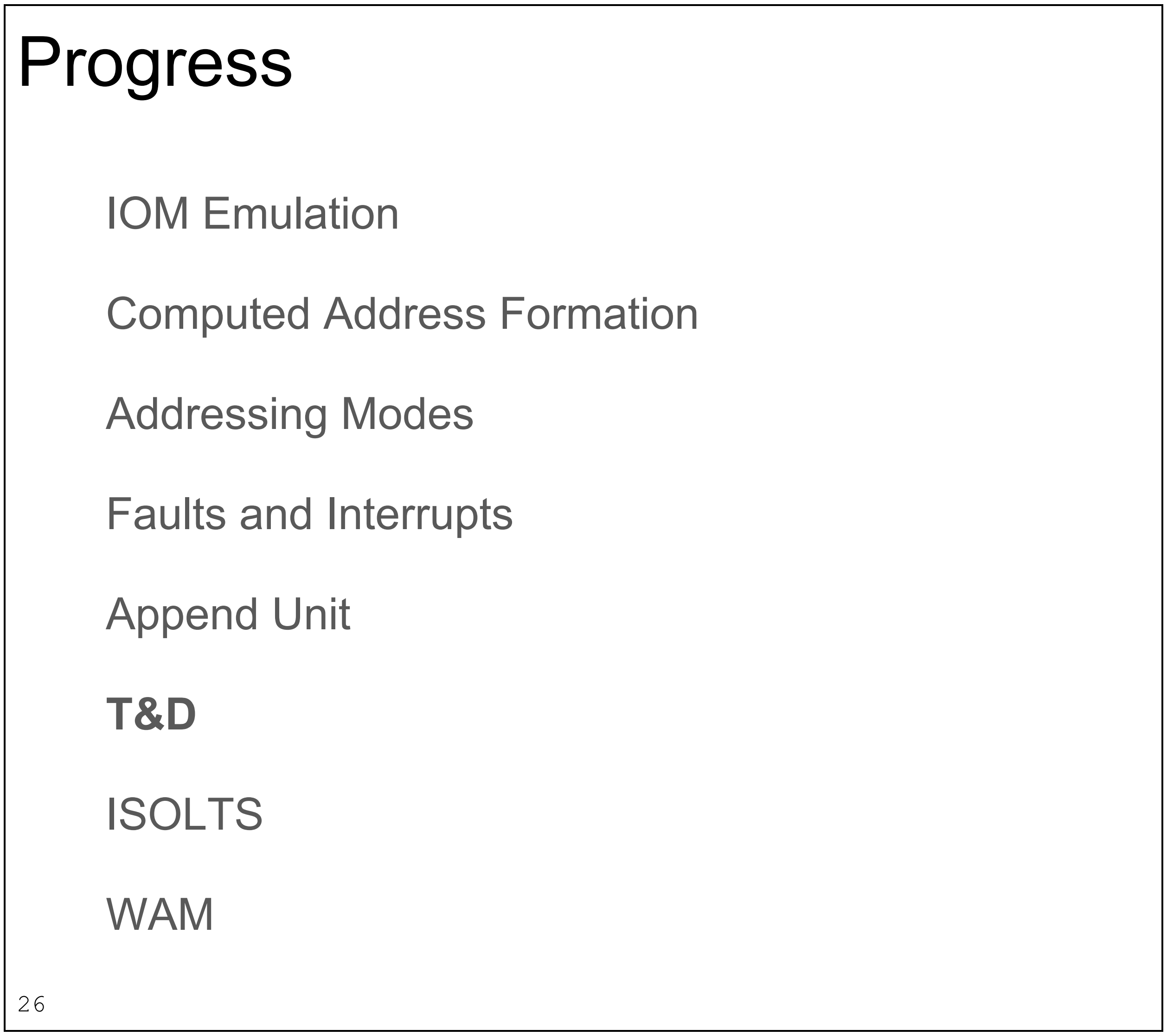
The fundamental implementation issue revolves around the append unit being a state machine. It has statements like “was the last cycle an indirect word fetch?”; this means correctly invoking the append unit is as important as its correct implementation, yet invocation is completely undocumented.

In addition there are items such as this node in the flowchart:

```
                                   |
                                   V
                              -----------
                            /  APU data   \  No ----+
                            \  movement?  /         |
                              -----------           |
                                   |  Yes           |
                                   V                |
                            --------------          |
                            | Load/store |          |
                            |  APU Data  |          |
                            --------------          |
                                   |                |
                                   |<---------------+
                                   V
25
```

This is the only mention of “APU data” in the documentation.

Again, we have managed to reverse engineer many details of the append unit by studying the Multics source code, consulting Multicians and tinkering with it until Multics works.



Progress

IOM Emulation

Computed Address Formation

Addressing Modes

Faults and Interrupts

Append Unit

T&D

ISOLTS

WAM

26

Bitsavers has a collection of Multics tape images, consisting of a MR12.5 boot tape, the MR12.3 Multics release, the MR12.5 update release, and a tape labeled T4D.

We eventually worked out that T4D was a transcription error; it should have read tee ampersand dee, or test and diagnostics. The Multicians indicated that this would be a standalone hardware diagnostic tape, and highly encouraged focusing on getting that running.

The tape turned out to contain a powerful and comprehensive diagnostic test suite. The first record on the tape is bootable, and consists of an overlay loader. The overlay loader searches the tape for records containing a specified key value, loading and executing the record when found. The records contain tests, and when each test is finished, it uses the overlay loader to start the next test.

The first few tests rigorously and comprehensively test a handful of instructions; the remaining tests are written entirely in that small set of instructions plus whatever instruction that particular test is covering. This technique gives a high degree of confidence to the test harness; ir a particular test fails, it is almost certainly due to a problem with the targeted instruction rather than a failure in the test harness code.

The first test covers the “jump” instruction. Does it jump to an even address correctly? Does it jump to an odd address correctly? Does it jump an from odd to odd correctly? And so on. The test is implemented by jumping through a thicket of “DIS star” instructions. This is a “delay until interrupt signaled” instruction; it will cause the CPU to halt, and the address of the instruction will be displayed on the maintenance panel. If the jump instruction lands at the wrong place, it will hit the DIS instruction halting the test. The Field Engineer will read off the address from the maintenance panel, look up the address in the T and D documentation, and follow the troubleshooting procedure outlined there.

We don’t have a copy of the T&D documentation.

We boot the tape on the emulator, it runs for a few thousand instructions, and halts at an address. Which has no meaning.

So the process becomes reverse engineering. Disassemble the code around the address and try to figure out what it was doing and what result if expected but failed to achieve.

Because the tests were written in the very small instruction set, many simple tasks took dozens or hundreds of instructions to accomplish, making comprehension of the test very hard.

Eventually we got the the first eleven tests to run (with one exception) , the twelfth test reads, rewinds and reads the tape endlessly; no one has been able to figure out if the problem is in the tape drive emulation not handling some command correctly, the tape image is incorrect and the record being searched for is damaged and unrecognized, or something is wrong in the instruction emulation.

The exercise was fruitful; there are many instructions that suffered from woefully incomplete documentation and that we now have high confidence in due to the T and D tape.

The instruction that we could never get right was the Load Indicator instruction. It loads the accumulator register with several computational state flags; such as the zero, negative, carry and overflow flags. It also loads the addressing mode bits BAR and Appending.

AL39 says the LDI instruction should set bit thirty one on if the CPU is in absolute mode and off if in appending mode. Examination of the Multics source code confirms this behavior, and other references in AL39 support this logic.

The T and D tape tests the LDI instruction while running in absolute mode and checks that the bit is off and signals a failure because the emulator, following AL39, sets it on.

Examination of the disassembled code showed a simple and straightforward test. The code was circulated among the Multicians, who agreed that the code was doing the wrong thing, and no one could come up with any plausible explanation.

We added a runtime configuration switch to the emulator that inverted the sense of bit thirty-one so that the T and D tests could continue on, and shelved that problem until someone could think of a plausible solution.

After several years, the solution finally dawned on us.

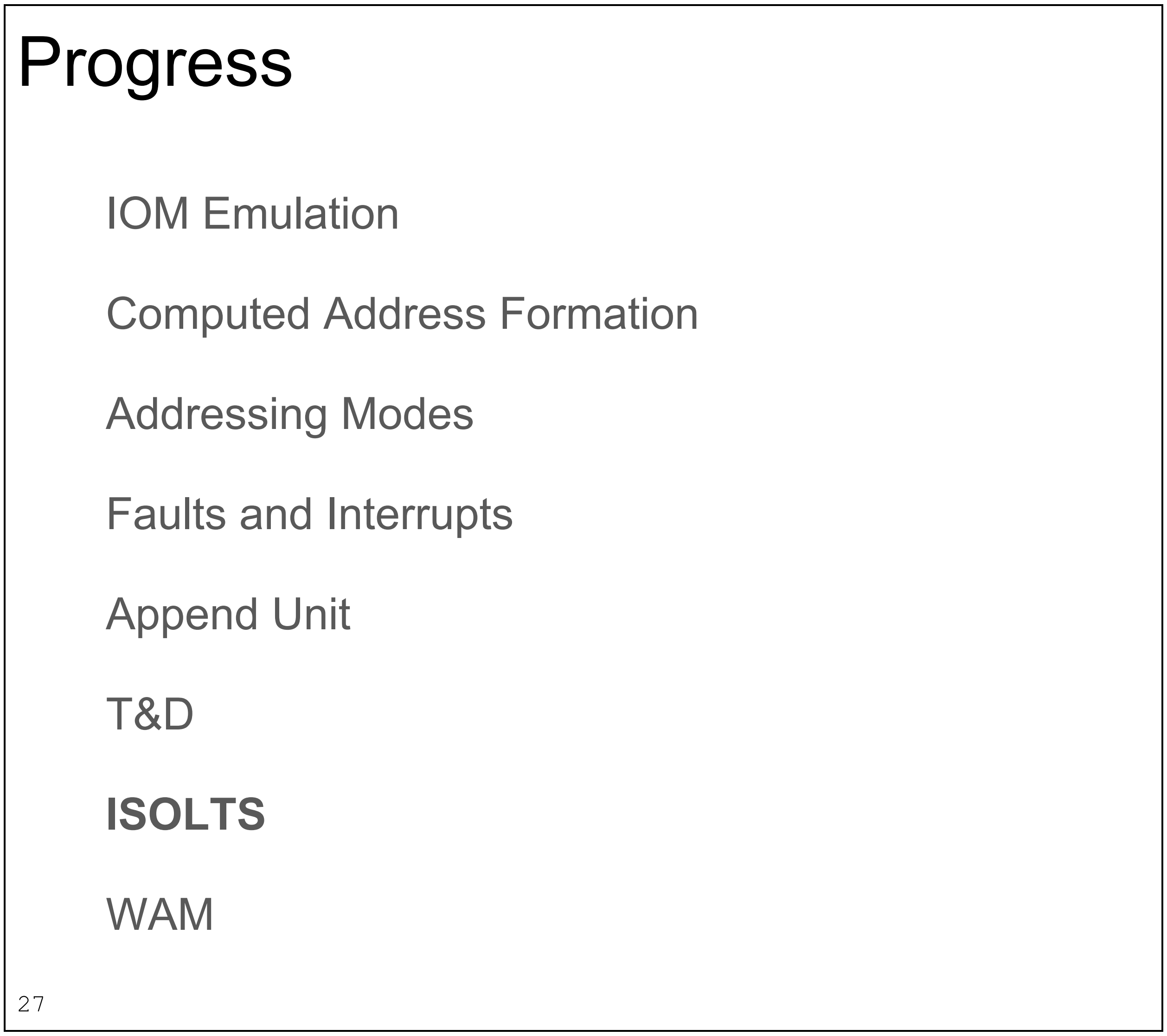
Wrong tape.

Booting the T & D tape loads the first and then the third record of the tape. What is in the second record?

Examination showed it to be a bootable record quite similar to the first record, except that it wanted the absolute mode indicator bit to work as documented.

It seems that the first record is for testing CPUs without the append unit installed; the non-Multics enabled model line that ran the GCOS operating system, To test a Multics CPU, skip the first record of the tape before booting.

The different boot records will set configuration bits differently in the overlay loader, causing it to select those test records on the tape appropriate for the CPU model.



Progress

IOM Emulation

Computed Address Formation

Addressing Modes

Faults and Interrupts

Append Unit

T&D

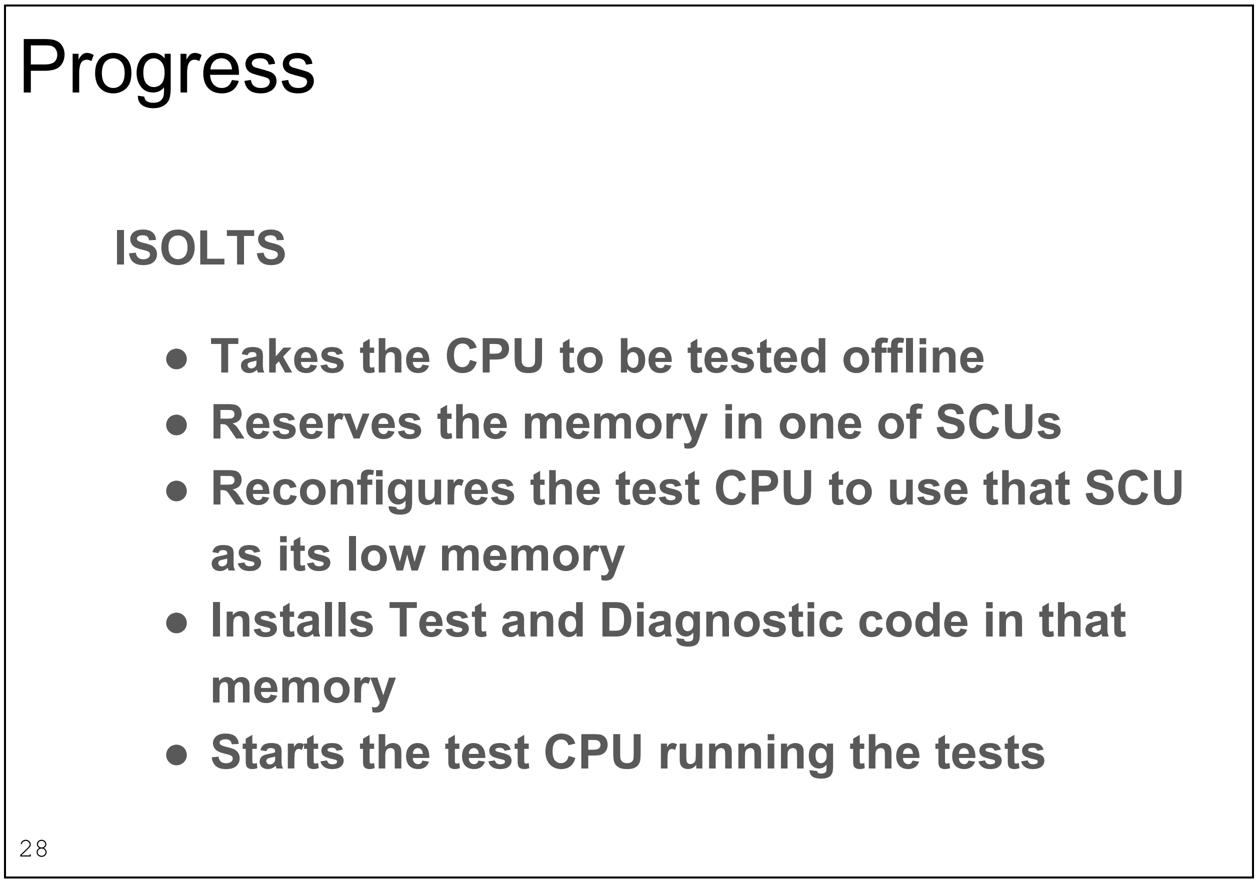
ISOLTS

WAM

Once the emulator was running well enough that Multics was bootable and stable, it not completely functional, we turned to ISOLTS.

The ISolated Online Test Subsystem is amazing. It is a rework of the T and D tests for use in a running system.

For sites with multiple CPUs and multiple SCUs, it was possible to run tests on one of the CPUs while leaving Multics running on the remaining CPUs with no service interruption.



Progress

ISOLTS

- Takes the CPU to be tested offline
- Reserves the memory in one of SCUs
- Reconfigures the test CPU to use that SCU as its low memory
- Installs Test and Diagnostic code in that memory
- Starts the test CPU running the tests

When the Field Engineer runs ISOLTS , it:

- Takes the CPU to be tested offline
- Reserves the memory in one of SCUs
- Reconfigures the test CPU to use that SCU as its low memory
- Installs Test and Diagnostic code in that memory
- Starts the test CPU running the tests.

It does this while Multics is still running on the remaining hardware, an impressive feat.

ISOLTS also has the benefits of much more verbose and generally helpful diagnostic messages, often including the purpose of the test, the expected result and the actual result.

Unfortunately, ISOLTS only runs on multiple CPU systems, Simh, and therefore the dps8m emulator only supports a single CPU.

Necessity calls.

I had always wanted to support a multiple CPU emulation with threading of some kind, but I knew that taking code of this complexity and size to threads would be a major undertaking, especially as threading considerations had never been taken into account as part of the design.

But I also new that one of the first steps toward that would be reorganizing the emulator's data structures into a coherent whole, not ad hoc mass of accretion that was the case.

I decided that if the data structure was properly marshalled, it would then be possible to make it an array and do a round-robin implementation of multiple CPUs.

Some rearranging and recoding later, I had multiple CPU support that didn't work. It turned out the the interrupt routing code was very broken for the case of more than one CPU; that took a fair amount of reverse engineering and testing to straighten out.

Now we could run a multiple CPU Multics, albeit running slower proportionate to the number of CPUs, but the ability to run ISOLTS at all makes the cost irrelevant.

ISOLTS runs the Primitive Function Tests from the T and D tape to prove the test harness instruction set; since that portion of the T and tape was already proven, the emulator passed the PFTs with flying colors.

However, upon passing the PFTs, ISOLTS displays hardware configuration data gleaned from its tests, including the CPU serial number. The instruction to retrieve the serial number is well documented, and Multics retrieves and displays the serial number correctly but ISOLTS displays gibberish for the serial number for unknown reasons.

ISOLTS then runs about ninety test suites, depending on the model and configuration of the CPU. Each test suite is composed of several subtests; each subtest deeply tests an instruction, exploring the corner cases. A tour de force of testing. It took a long time and a lot of code rewriting to get from being able to run ISOLTS to the point where we were happy with the ISOLTS results.

The remaining ISOLTS failures are mostly due to implementation comprises or features that we decided not to implement. For example the original hardware had a 512 Kilohertz countdown timer used to schedule process switching; the emulator estimates time by counting instructions; sacrificing timer accuracy in exchange for significantly reduced overhead; and ISOLTS can detect that discrepancy. Also, the hardware supported interruption of long-running instructions for timer runouts or hardware interrupts; since the single threaded emulator handles interrupts synchronously, those long-running instructions are allowed to complete; and again, ISOLTS detects that.

There are a couple of tests were we have convinced ourselves take the instruction under test is correct and that ISOLTS is misdiagnosing; it is unclear as to the root cause.

Progress

IOM Emulation

Computed Address Formation

Addressing Modes

Faults and Interrupts

Append Unit

T&D

ISOLTS

WAM

The speed bottleneck in the DPS8/M hardware is main memory; it reads and writes a 72 bit double word at one megahertz.

In appending mode, an instruction fetch requires

- Lookup the segment in the Descriptor Segment Page table
- Retrieve the segment descriptor from a Descriptor Segment Page
- Lookup the page address in the Segment Page Table
- Read or write the operand data at the addressed page.

Four memory cycles to just fetch the instruction.
In order to deal with this, segment descriptor and page table caches (16 entries deep on the 6180, 4 way 64 entries deep on the DPS8).

By the time I had arrived, someone had carefully coded a slightly buggy and incorrect (one way instead of four way) cache.

At some point, even though I am aware of the dangers of premature optimization, I did an execution profile of the emulator to see if there were any obvious hot spots. Yes there was. The emulator was spending ninety percent of its execution time searching the caches for entries that it could simply fetch directly from the blazing fast memory of the host in less time then it took to check a single entry. Making a quick pass through the code replacing every check of the cache with the assertion that the entry was not in the cache produced a ten times speedup in the emulator.

These caches were critical to the DPS8/M performance, so ISOLTS tests them heavily. THe cache code has been reinstated as a compile time conditional and the 4 way logic has been correctly implemented.

Other documentation highlights

PR/AR Dichotomy

There are eight pointer registers and eight address registers, but there are really eight pointer/address registers.

Pointer registers have an eighteen bit word number field, a fifteen bit segment number field, a three bit ring number and a six bit bit number field; they can specify any bit in any word in any segment.

Address registers have an eighteen bit word number field a two bit character number field, and a four bit bit number field, but no segment number; they can address any bit in any character in any work in the current segment.

The important difference is the way that they number bits in a word.

For the pointer register, the bits are number zero through number thirty five, specified by a six bit integer.

For the address register, the work is divided into four nine bit characters (zero through three), specified by a two bit integer, and a bit position in the character (zero through eight) specified by a four bit integer.

EIther register can specify a particular bit, but with different notations.

AL39 then gets a bit confusing.

“Because the Multics processor was implemented as an enhancement to an existing design, certain apparent anomalies appear. One of these is the difference in the handling of unaligned data items by the appending unit and decimal unit. The decimal unit handles all unaligned data items with a 9-bit byte number and bit offset within the byte. Conversion from the description given in the EIS operand descriptor is done automatically by the hardware. The appending unit maintains compatibility with the earlier generation Multics processor by handling all unaligned data items with a bit offset from the prior word boundary; again with any necessary conversion done automatically by the hardware. “

Examples were found in the Multics source code that demonstrated that the address and pointer registers were the same registers; places were found where a pointer register was set to zero and the code assumed that the corresponding address register had been set to zero.

We concluded that the register were the same bits, the interpretation of the six bits of bit offset data varied between the appending unit and the decimal unit, and the emulator was coded to store the bit offset in the pointer register format; those instructions that referenced the address register would convert to and from the character number and offset format as needed.

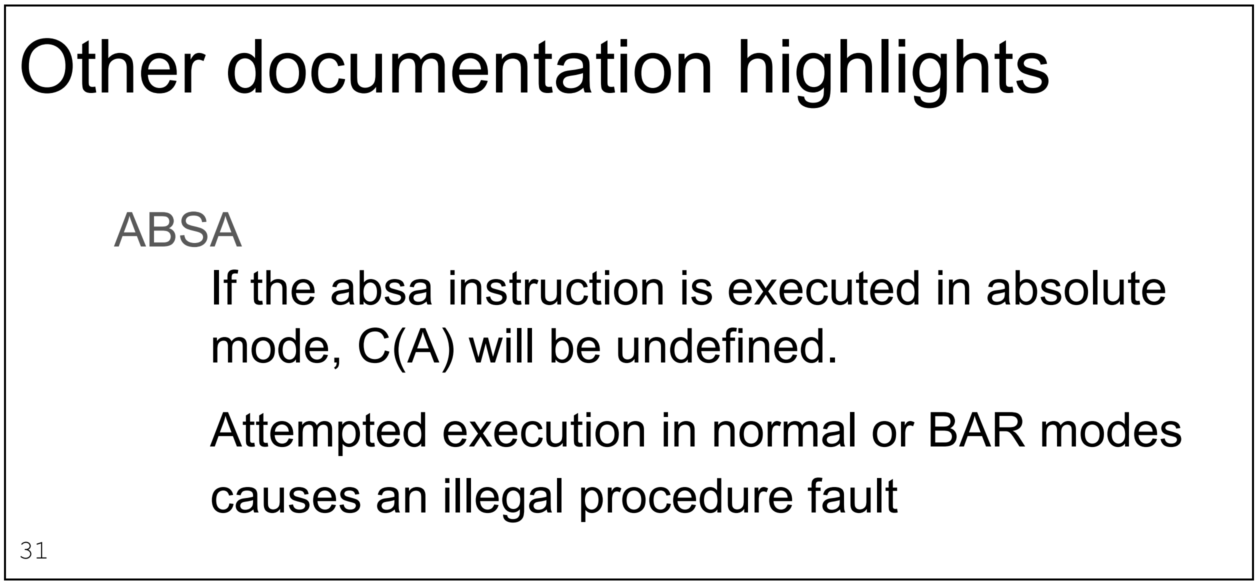
Multics seemed to run fine that way, but ISOLTS complained mightily.

Careful examination of the failing cases indicated that the solution was a bit more convoluted then we realized.

ISOLTS was carefully checking the conversion of bit offsets in the address registers including “illegal values”. It would set the address register to character zero, bit ten; the emulator would convert this to bit ten and store it in the pointer register. When ISOLTS read back the register, the emulator would convert bit ten to character one, bit one which ISOLTS considered incorrect.

Reworking the code to save the bit offset in the character address format caused ISOLTS to fail the pointer register tests.

More head scratching and wading through documentation led to the complex answer. The pointer and address register are separate but synchronized. Setting a pointer register sets the corresponding address register; and vice-versa. Setting and reading the same register preserves the data; reading the corresponding register retrieves the data that was changed to the appropriate format.



Other documentation highlights

ABSA

If the absa instruction is executed in absolute mode, C(A) will be undefined.

Attempted execution in normal or BAR modes causes an illegal procedure fault

31

Class, turn your AL39 to page 218: ABSA Absolute Address to A-Register.

At “Summary”, we see “Final main memory address to A register.”

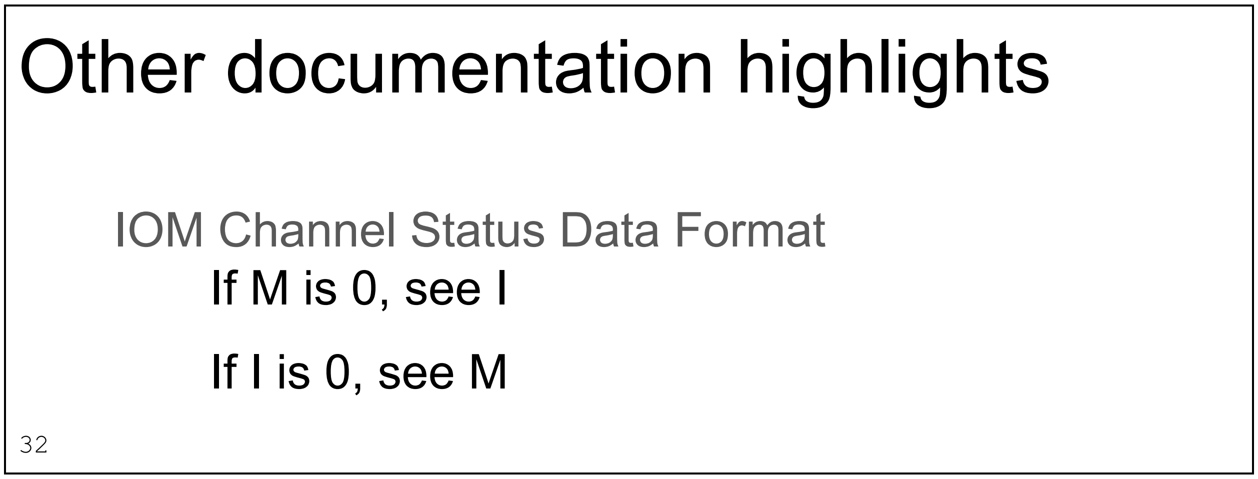
At “Notes”, we see:

“If the absa instruction is executed in absolute mode, C(A) will be undefined.”

And

“Attempted execution in normal or BAR modes causes an illegal procedure fault.”

That doesn’t seem to leave any modes that the instruction actually works in. The T and D tape and ISOLTS indicate that AL39 is just wrong; ABSA works in all modes, albeit with interesting differences between the modes.



Other documentation highlights

IOM Channel Status Data Format

If M is 0, see I

If I is 0, see M

32

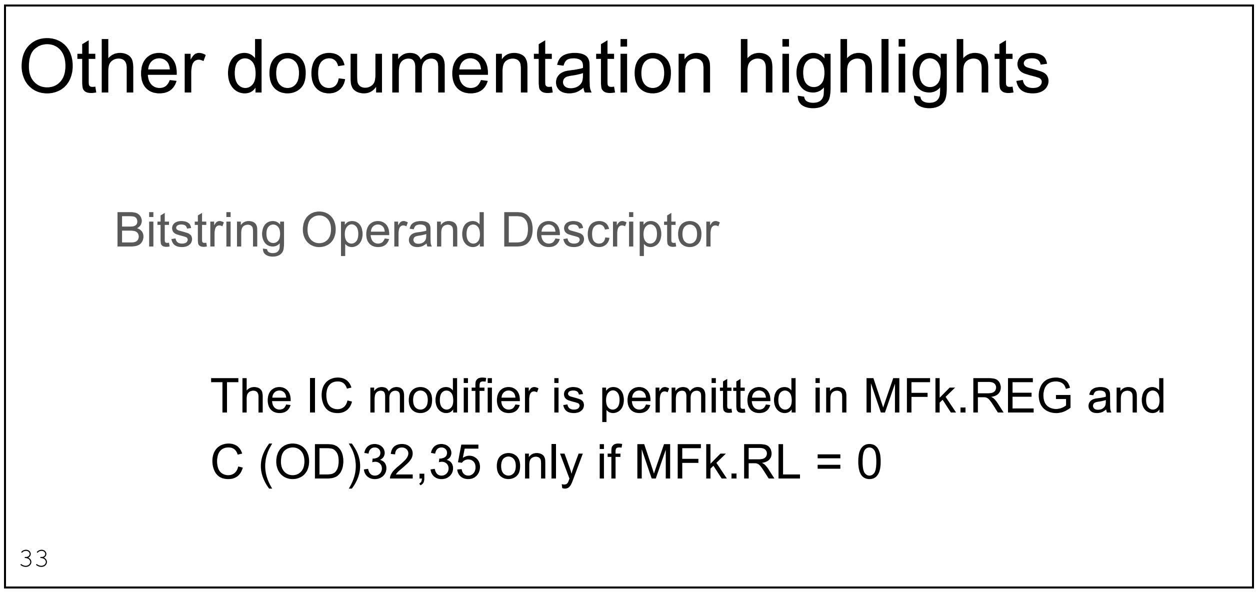
AN87 Hardware/Software Formats Programming Logic Manual, page 3-11, IOM Channel Status Data Format.

In the even word of the data, bit thirteen is the “M” bit and bit sixteen of the “I” bit.

``“M” is the marker status bit. If zero, initiate/terminate status as per “I” bit described below; if one, marker interrupt status.

``“I”, the initiate status bit. If zero, initiate/terminate status as per ‘M’ bit described above, if one initiate status in response to a request status or a reset status command.’’

Turns out they are trying say that the status word will never have both the “I’ and “M” bits set at the same time.



Other documentation highlights

Bitstring Operand Descriptor

The IC modifier is permitted in MFk.REG and C (OD)32,35 only if MFk.RL = 0

There are a few instructions that operate on a string of bits; for those instructions, the operand address consists of a word address, a starting bit position in that word and a bit count.

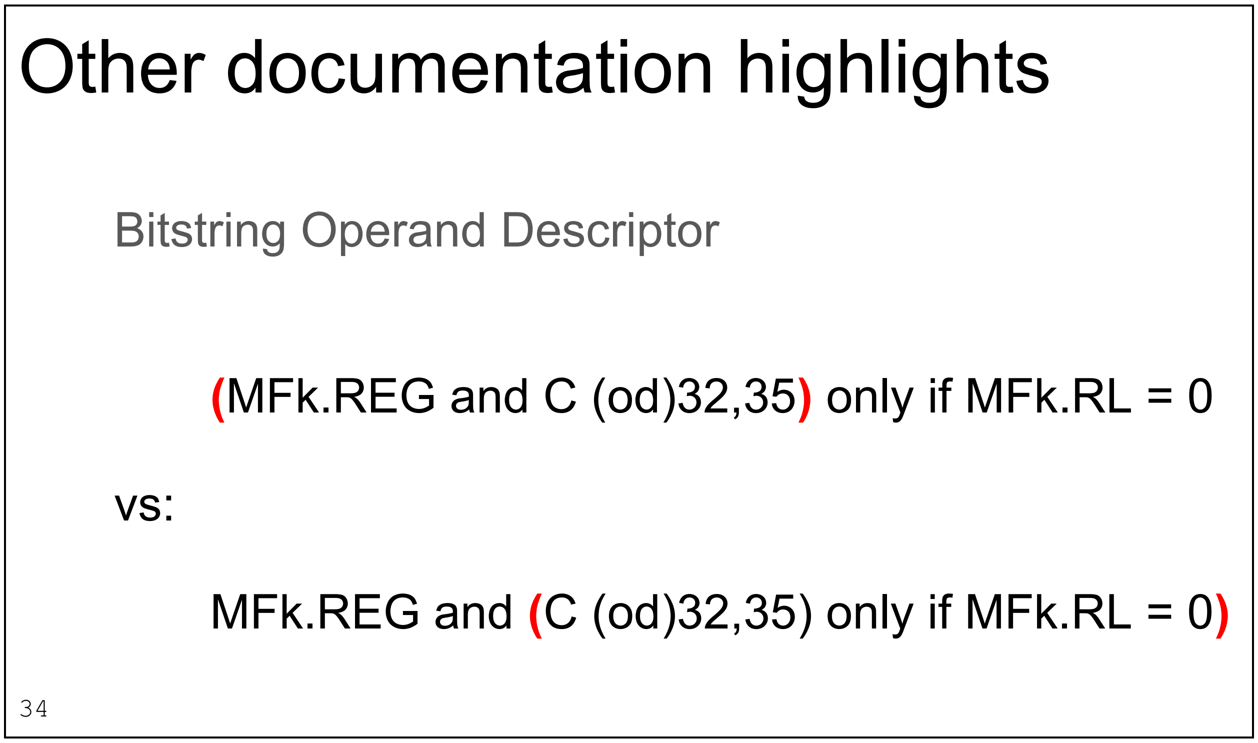
The operand descriptor contains this data and various bits controlling the operand address interpretation.

MFk.REG specifies how the address is computed from the offset value in the descriptor; for the case of IC, the value in the descriptor is added to the instruction counter to form the address -- “PC relative addressing”.

MFk.RL specifies if the bit count is specified in the descriptor or in a register; if zero, the count is in the descriptor; if non-zero the count is in register specified by bits thirty-two through thirty-five of the descriptor.

This was coded into the emulator; if a bit string operand descriptor matched the specified condition, a fault was thrown.

Multics kept throwing that fault; apparently Multics thought that it was a valid construct, so the test was commented out in the emulator. ISOLTS then complained that the instructions were not faulting when the should; much head scratching ensued.



Other documentation highlights

Bitstring Operand Descriptor

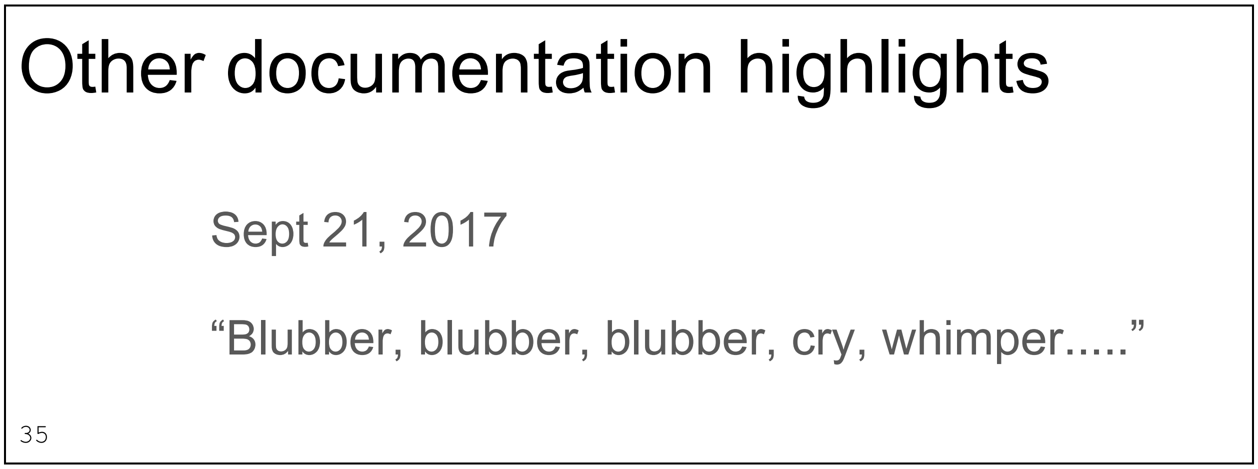
(MFk.REG and C (od)32,35) only if MFk.RL = 0

vs:

MFk.REG and (C (od)32,35) only if MFk.RL = 0)

34

By carefully analyzing which cases ISOLTS passed and which failed, it was realized that we were parsing the text wrong; evaluating terms in the correct order made everyone work correctly.



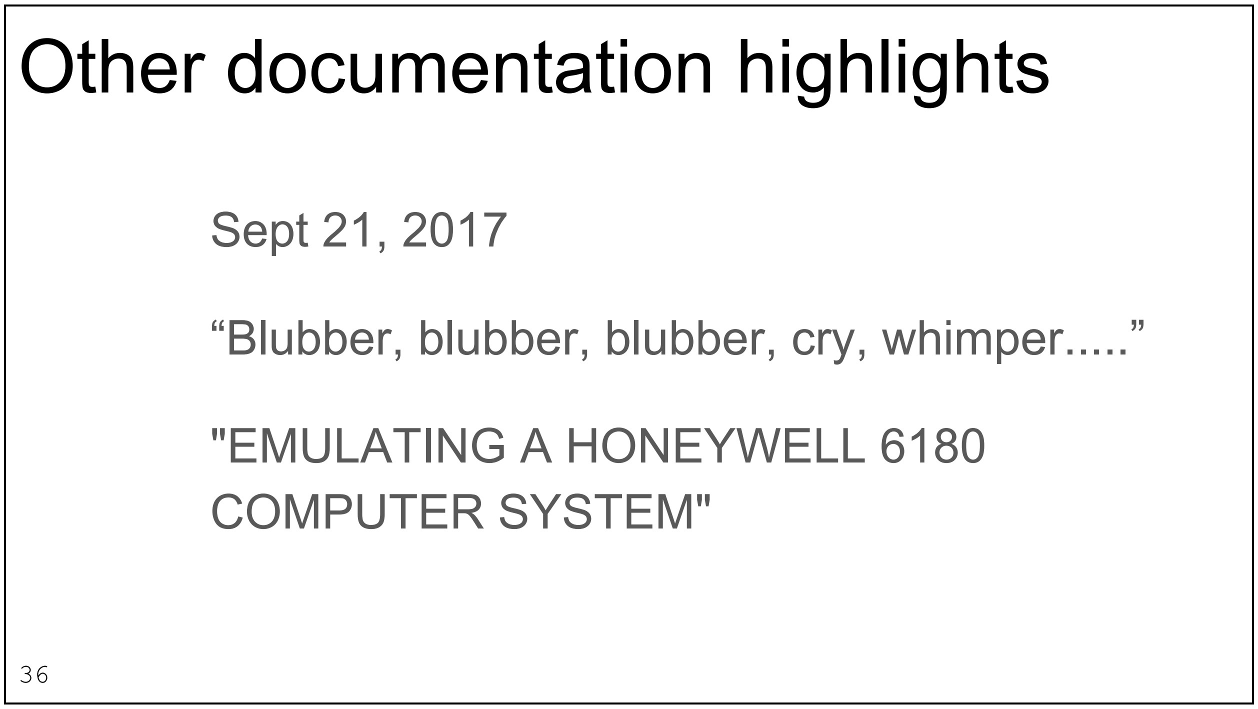
Other documentation highlights

Sept 21, 2017

“Blubber, blubber, blubber, cry, whimper.....”

35

On Sept. 21, I stumbled across a document:



Other documentation highlights

Sept 21, 2017

“Blubber, blubber, blubber, cry, whimper.....”

"EMULATING A HONEYWELL 6180 COMPUTER SYSTEM"

36

"EMULATING A HONEYWELL 6180 COMPUTER SYSTEM"

A report prepared by Mitre Corporation for the Rome Air Development Center, dated June, 1974.

Here was a document containing answers to everything we had spent years struggling with. Fortunately for our mental health, that was not the case.

The report is a proposal discussing the feasibility and scope of such a project.

It contains a 39-page summary of the 358 pages of AL39, including the nine page Append Cycle flowchart compressed down to two pages, and interestingly, fixing the ring write bracket register number typo.

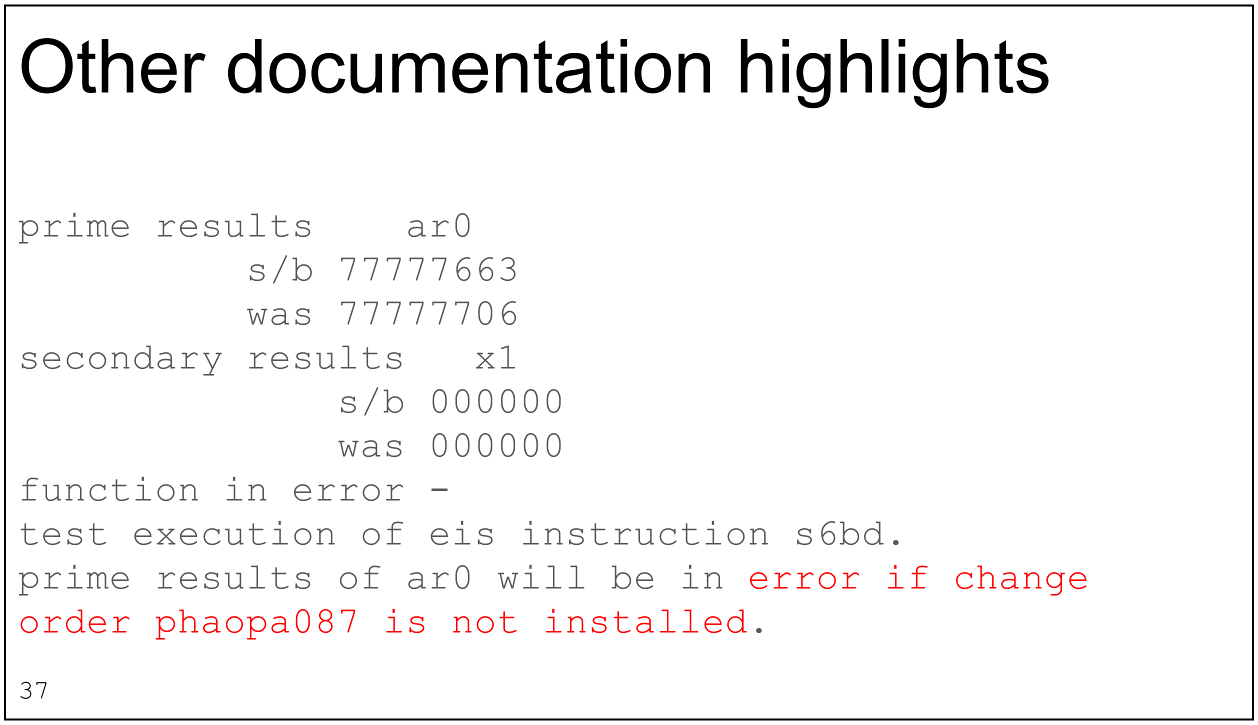
It then spends six pages discussing design strategies; it is interesting that they note that the emulation may be significantly slower, and suggest implementing the append unit in microcode if at all feasible.

They then describe a “benchmark” emulaton used to judge the suitability of various host computers. The benchmark consists of writing enough of the CPU and Append Unit to fetch and execute a “store accumulator” instruction in append mode and using a pointer register as part of the operand address; a quite typical instruction generated by the Multics Pl/I compiler.

They then evaluate the feasibility of implementing the benchmark on a Burroughs D-machine, a Nanodata QM-1, and a Burroughs B1700.

They then estimate emulation speeds on the three machines.

They never actually write any code or estimate the scope of writing an complete emulation.



Other documentation highlights

```
prime results    ar0
          s/b 77777663
          was 77777706
secondary results   x1
              s/b 000000
              was 000000
function in error -
test execution of eis instruction s6bd.
prime results of ar0 will be in error if change
order phaopa087 is not installed.
```

ISOLTS was not always helpful. Just install a hardware change order in the emulator.

Adventures in Third Party Libraries

Leap seconds

We all love third party libraries; stop reinventing the wheel!

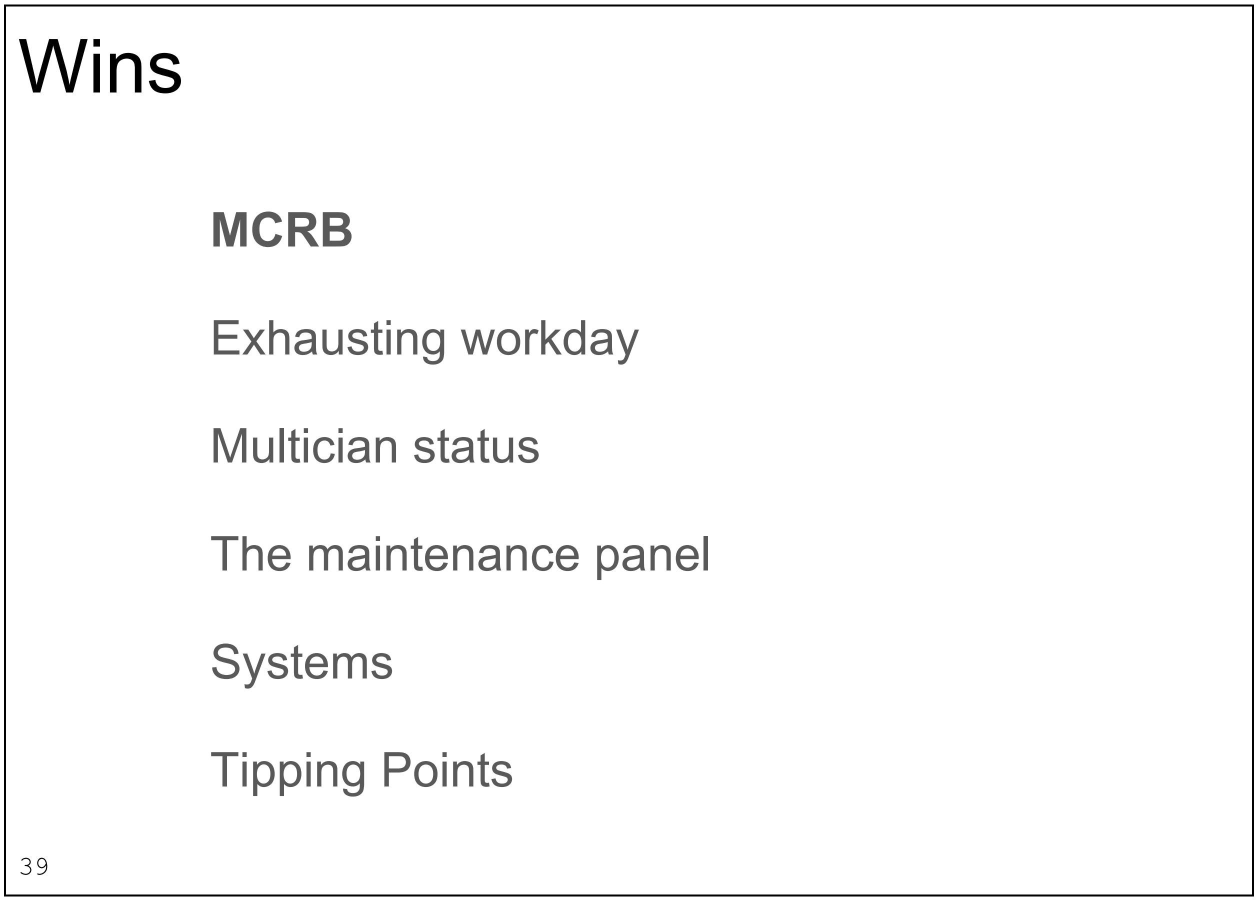
At one point, we were using a third party IPC library; very easy to use, solved the problem, and one day it threw an assertion.

A bit annoying; assertions in third party code are difficult to recover from, so this cast some doubt on the suitability of the library. It did, at least, leave a core dump so I could analyze what happened and decide on the best approach to resolving the issue.

The analysis revealed that the code was indeed experiencing an “impossible” condition, and it was… awesome.

The library was very, very careful about keeping track of time; including checking for consistent behavior from the system clocks.

With a staggeringly improbable timing, the code was running and doing it’s careful checking of the clock when a leap second was propagated to the system, causing a sufficiently large discontinuity in the system clock that the library panicked.



Wins

MCRB

Exhausting workday

Multician status

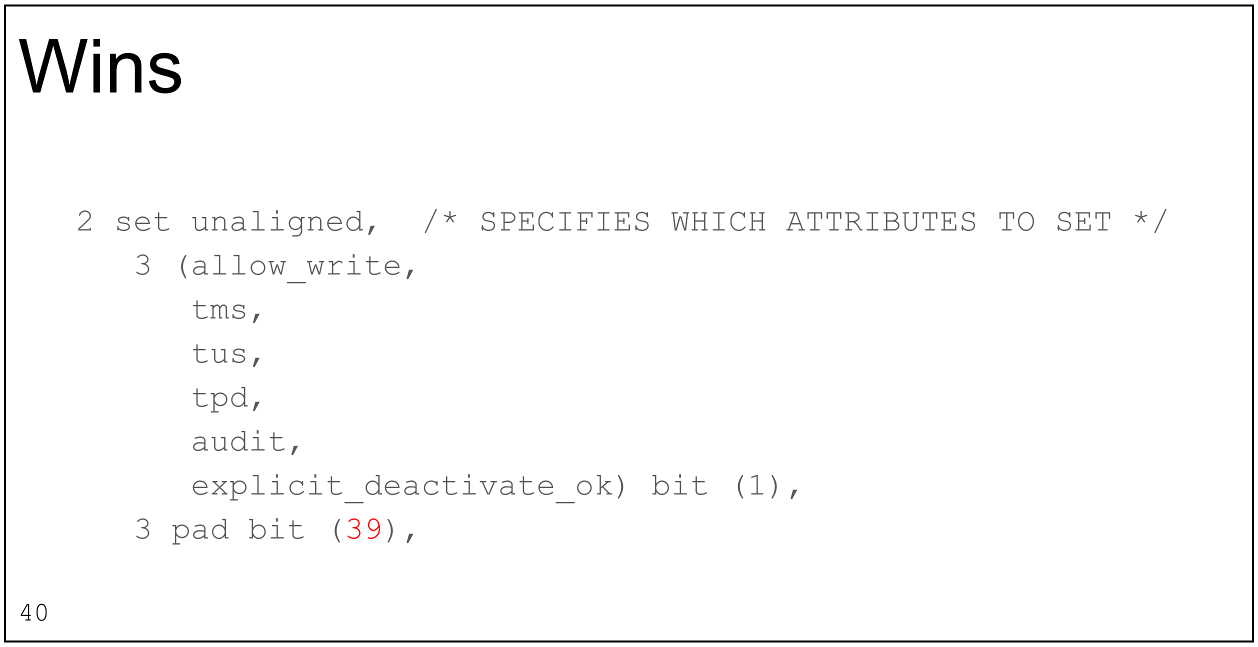
The maintenance panel

Systems

Tipping Points

39

Back in June of 2014, I discovered a bug in Multics; it was a minor bug that I found through code examination. I was tracking down a bug in the emulator by following some data moving through the Multics data structures. Examining the contents of memory usually meant looking up the PL/1 variable declaration and and counting bits to extract fields out of words. When analyzing the kst_attritbutes declaration, I noticed that the declaration was not padded out to a word boundary as was the usual practice, and that the last field in the declaration crossed a word boundary, which was quite atypical.



Wins

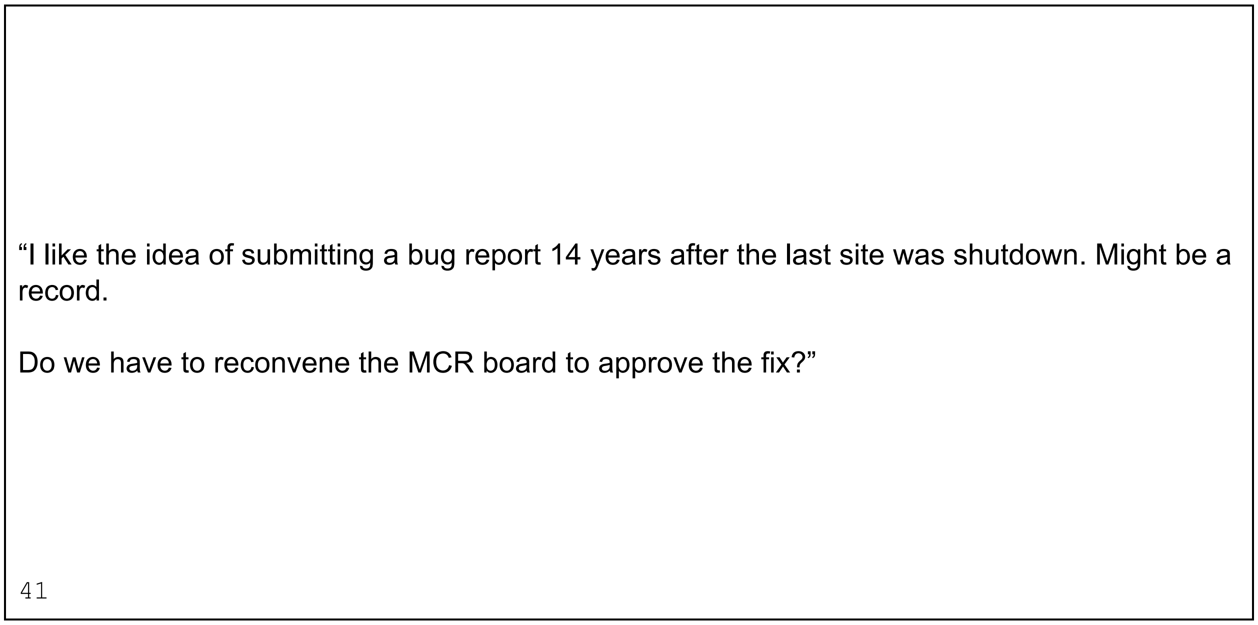
```
2 set unaligned,  /* SPECIFIES WHICH ATTRIBUTES TO SET */
   3 (allow_write,
      tms,
      tus,
      tpd,
      audit,
      explicit_deactivate_ok) bit (1),
   3 pad bit (39),
```

40

I decided that the last field “pad bit 39” was an typing error and should be pad bit 30. This bug was minor; a data declaration was one word longer than it needed to be, causing a small waste of storage and no functional or performance impact.

On a lark, I wrote my analysis and put it out the Multicians mailing list as a bug report.

This got the response:



“I like the idea of submitting a bug report 14 years after the last site was shutdown. Might be a record.

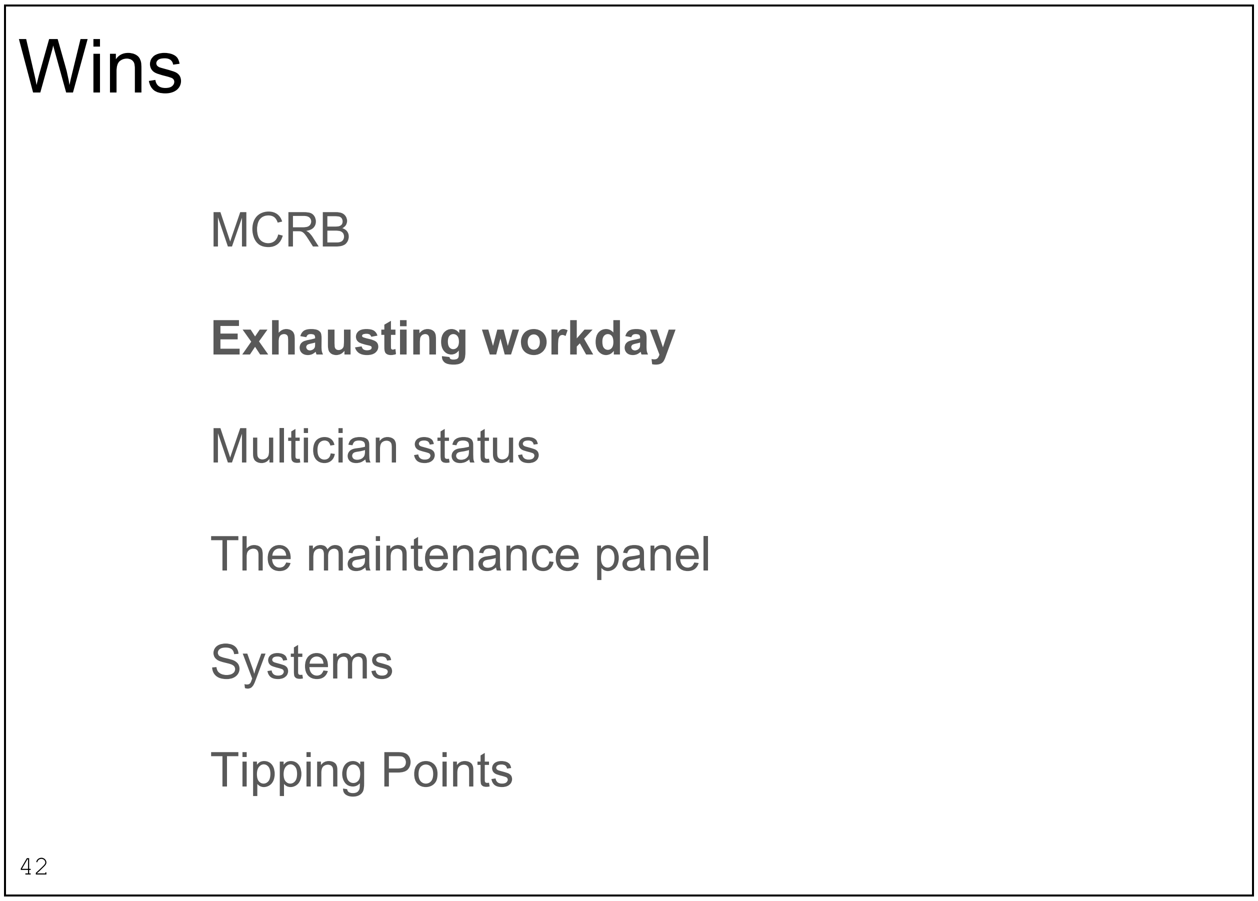
Do we have to reconvene the MCR board to approve the fix?”

41

“I like the idea of submitting a bug report 14 years after the last site was shutdown. Might be a record. Do we have to reconvene the MCR board to approve the fix?”

The MCR board was the Multics Change Review Board. They reviewed and approved changes to Multics; they had a formal process of change submittal, review, approval, auditing and installation.

On January 6th, 2017 a new MCRB composed of Multicans and DPS8/M emulator developers approved MCR10021 fixing the issue; the fix was incorporated into MR12.6f.



Wins

MCRB

Exhausting workday

Multician status

The maintenance panel

Systems

Tipping Points

42

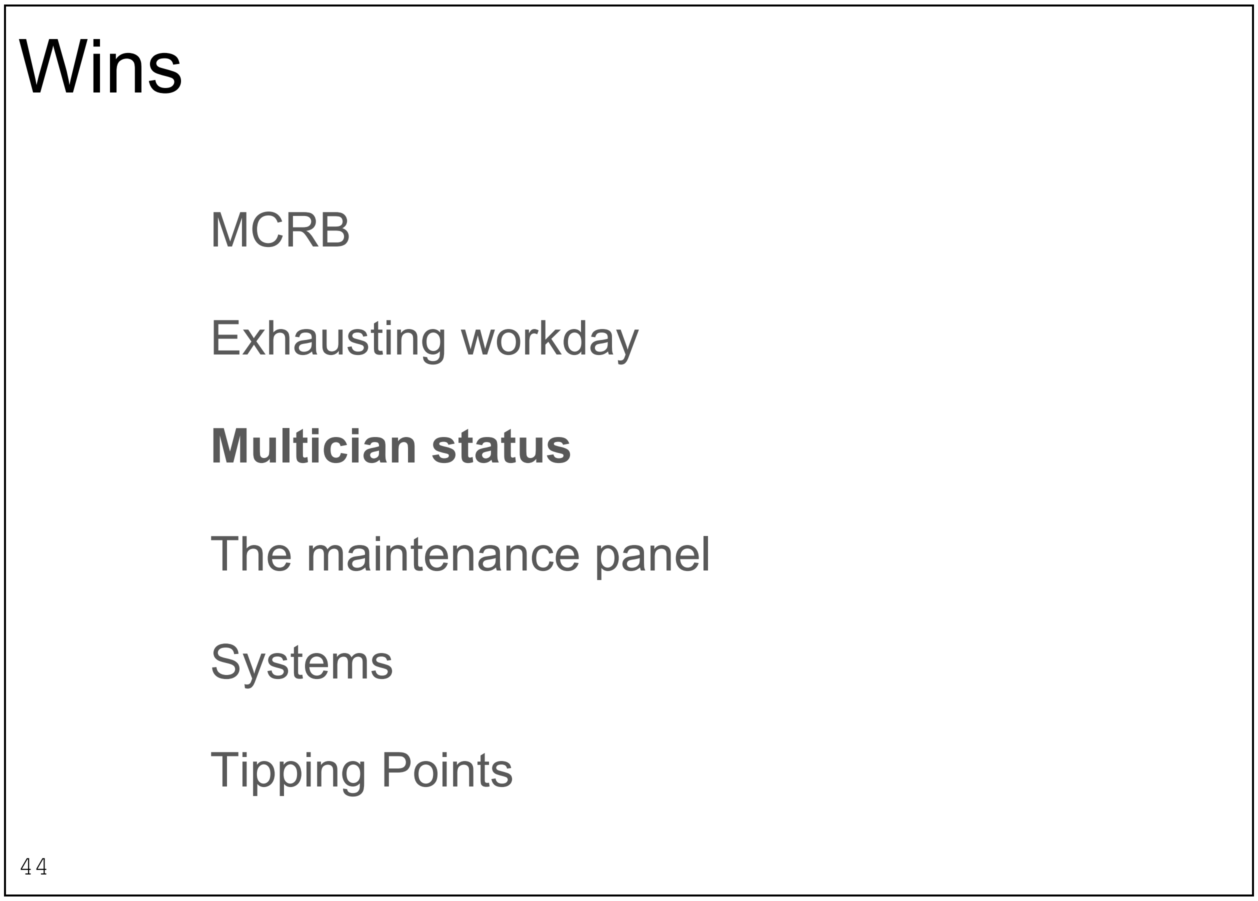
In July of 2015 I found an actual logic error in Multics; the program “acquire_resource” used incorrect logic to check an error return status, causing an a gibberish error message. I duly filed a bug report on the Multicians mail list, eliciting this response:

"Super. Nothing like coming home from an exhausting workday to find a bug in my 35-year-old code has come back to haunt me.   :-("

43

"Super. Nothing like coming home from an exhausting workday to find a bug in my 35-year-old code has come back to haunt me. "

My proposed fix was accepted as MCR10010 and shipped in MR12.6e.



Wins

MCRB

Exhausting workday

Multician status

The maintenance panel

Systems

Tipping Points

44

Some time in 2015, there was a discussion on the Multicians mailing list about the volume of emulator discussions on the list, and if moving technical discussions to a separate mailing list, and inviting those Multicians who were interested in those details to join that list. The consensus was that this was a Good Idea; that the Multicans list would like to be kept up to date on progress, but detailed discussions were not of sufficient general interest.

During that discussion the creator and maintainer of the Multicians web site and mailing list stated: “I think the emulator development team are just as much Multicians as everybody else on the list, and just as welcome.” About that time the names of the developers appeared on the web site’s List of Multicians page.

Speaking for myself, and I hope for the other developers, I was gobsmacked at that honor; I carry that title with great pride.

Wins

MCRB

Exhausting workday

Multician status

The maintenance panel

Systems

Tipping Points

In August of 2016, I got it into my head to take a road trip down to San Jose for the Vintage Computer Festival West and do an emulator exhibit. I was going up against lots of impressive hardware and blinking lights; all I had was some software that ran some software that no one had ever heard of.

I set up my computer and display materials and talked to people who wandered up. Most of them were looking for their parents' gaming consoles and had no idea what I was going on about. (Back away slowly from the crazy person.) A handful of people understood something about mainframe computing and emulation and listened for a bit. A very small handful said "Multics? Really? Show me!", and had a great time. A couple of Multicians showed up and introduced themselves, and one dps8m developer showed up. And a curious thing happened: someone from the Living Computer Museum handed me a business card and asked me to contact them.

It turns out the the Museum had a CPU maintenance panel from a Honeywell 6180, in very good shape other than the fact that it had been separated from the CPU with wire cutters. The museum wanted a Multics exhibit and thought that hooking the emulator up to the panel and having it drive the lights would make an excellent centerpiece to a Multics exhibit.

The Museum tasked one of their hardware people to sort through the over 600 snipped wires (all of which were color coded “white”) and figure out how the 576 lights and 96 switches worked, and to build an interface that the emulator could talk to. I tackled tracking down the meaning of the often cryptic labels of the buttons and switches, and added code to the emulator to drive the lights and respond to the controls on the panel.

In August of 2017, I again went to the Vintage Computer festival, this time in the company of a six foot tall panel with more than 500 blinking lights. This time we got a lot of traffic; the software people would stand in front of the panel and talk to me, the hardware people would circle around to back of the panel and talk to Jeff.

Wins

MCRB

Exhausting workday

Multician status

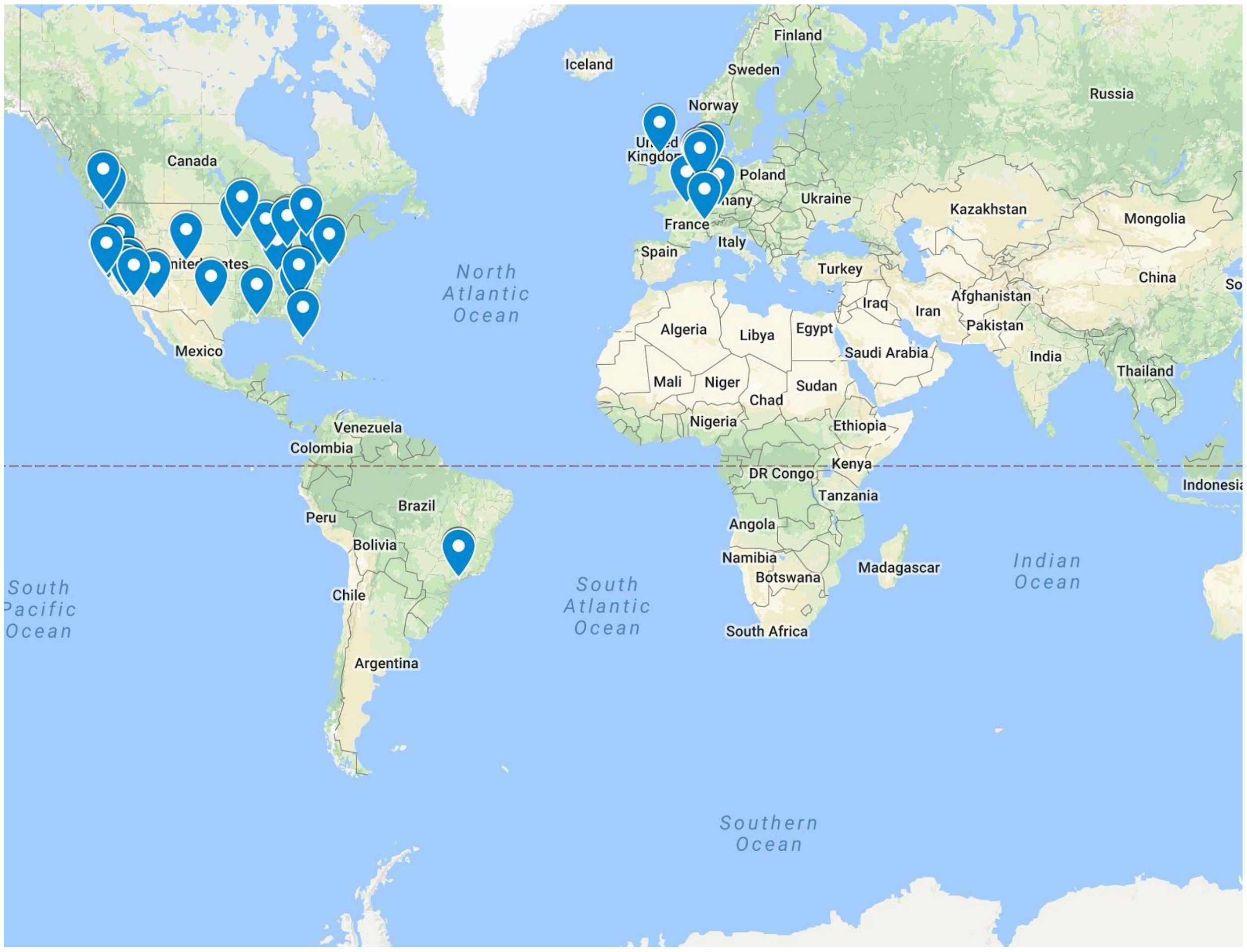
The maintenance panel

Systems

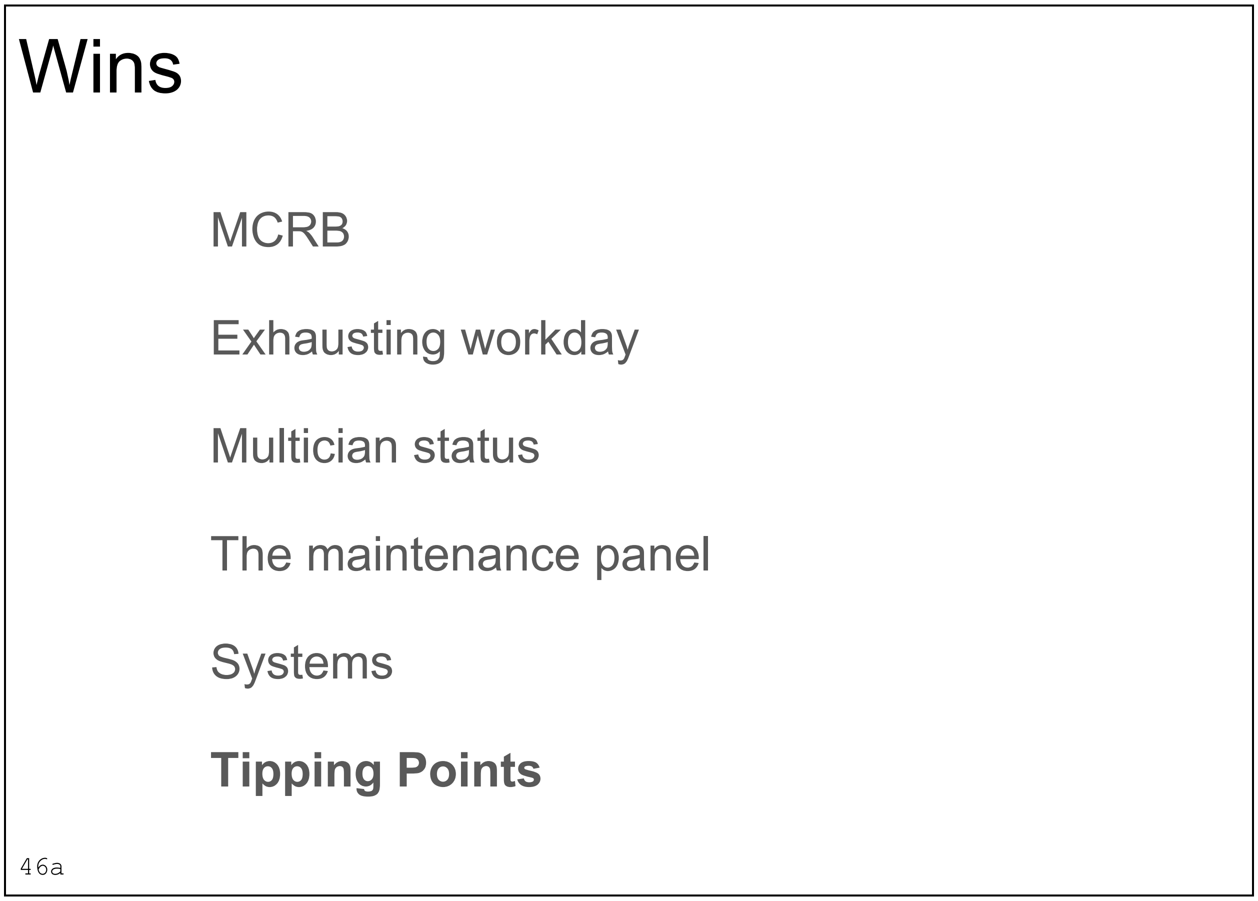
Tipping Points

Historically, Multics was installed at about one hundred sites.

Having no idea how many copies of the emulator were out there, we started asking people who were running Multics to register their systems.



As of July 2017, 45 registered systems.



Wins

MCRB

Exhausting workday

Multician status

The maintenance panel

Systems

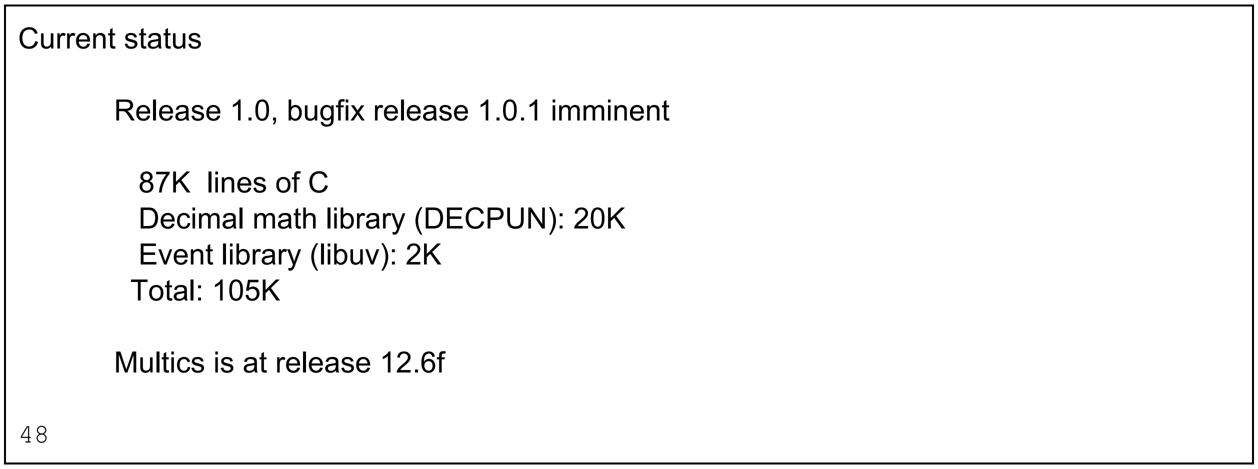
Tipping Points

46a

Two interesting “tipping points” occured.

The Multics engineers were seriously interested in reliability; one of the important reliability engineering tools is failure analysis. To this end, when Multics crashes, it carefully saves on disk data about the system state -- a structured core dump. When the system is restarted, the saved data can be analyzed with “azm”, the “analyze Multics” tool. Azm does stack and system state analysis, allowing the engineers insight into the crash. Early on in the emulation development, Multics crashed a lot. At one point the emulator started working well enough that azm started to produce usable analyses, and was helping debug the emulator.

Multics also has similar “core dump” tools for user level crash analysis. I received an email from a user with core dump printer output attached. Pages and pages of 132 columns of octal numbers. From that, I was able to determine what program crashed where, and was able to track down the failing instruction and fix an emulator bug.



Current status

Release 1.0, bugfix release 1.0.1 imminent

87K lines of C
Decimal math library (DECPUN): 20K
Event library (libuv): 2K
Total: 105K

Multics is at release 12.6f

48

Current status: Release 1.0, bugfix release 1.0.1 imminent

87,546 lines of C
Decimal math library: 20,235
Event library: 2,145
Total: 105,781

As a comparison, simh Alpha runs about 7K lines of C, simh PDP-10 about 13K and simh VAX about 63K. Hercules runs a whopping 235 KLOC, however it supports a vast array of CPU models from the 360 up to the z/Architecture, is multithreaded and has a spectacular feature set.

Multics is at release 12.6f

The Future

Threaded support for Multiple CPUs
FNP emulation
Fixing Append Cycle/Computed Address Formation
Pl/I compiler
Networking
Code improvement

49

What is on the slate for the future:

Fixing bugs, of course.

Work is proceeding on a threaded emulator; the current threaded code base has been known to run a dual CPU configuration for up to ten minutes under load before crashing horribly.

The FNP is the communications processor; it handles asynchronous serial lines, X.25, btsync, HDLC and other acronyms. Currently, it is emulated at the protocol level; the FNP and Multics exchange messages like “write the data to the line” and “here is some data that came in over a line”, these messages are parsed by the FNP emulation code and acted upon.

The actual FNP was a 18 bit minicomputer with an architecture remarkably similar to the DPS8/M. We are in possession of seemingly good documentation of the architecture, the source code of the software that ran on it and a good understanding of its behavior. It would be an interesting project to implement the FNP as an instruction level emulation.

There is an outstanding bug in the emulator -- the GTSS GCOS simulator crashes. The crash occurs in code taken from a GCOS library that we do not have the source for and identifying the issue remains elusive.

Examination of the Multics source leads to the observation that most of Multics is written in PL/I, and that the Multcs PL/I compiler generates a limited set of memory addressing code sequences. This implies that only a few paths are taken through the memory addressing portions of the emulator and that those paths are well exercised and debugged.

The GCOS code was written by a different team of engineers, who in all probability used different memory addressing patterns, and that the GCOS code takes different, un-debugged code paths through the emulator.

We have stared long and hard at the Append Unit and the Computed Address Formation documentation and have come to the conclusion that they are incomplete and incorrect.

The missing bits of the logic need to be deduced and coded.

We have identified two bugs in the Multics PL/I compiler; one was traced back to an error in the last compiler update, which we had the before and after source code for, and was a relatively simple fix. The second bug is well understood; the compiler fails to generate code to save a register across the second part of a computation, we don’t understand the workings of the compiler well enough to design a fix; nor do we have a PL/I compiler validation test suite to test fixes.

The ARPAnet and TCP/IP stacks were unbundled products and have been lost. Implementing a network interface in the emulator is simple enough, but we need to rewrite the stacks.

The code can always be improved. Performance on small ARM systems is very poor, mostly due to a low cache hit rate; the emulator data flow needs to be simplified and reorganized with regard to caching issues. The decimal math is handled by a generic decimal math library; this means that every decimal instruction converts the operands from DPS8/M format to the library format, performs the operation and converts the result back to DPS8/M format. A bespoke decimal math library would be much more efficient.

Acknowledgements

My fellow DPS8/M developers past and present
The Multicians
Bull HN Information Systems
SIMH
SourceForge and Wikidot
Classic Computers mailing list members
Living Computer Museum + Labs
Sultan Library, Gathering Grounds Coffeehouse and Galaxy Chocolates
Damon and Lily
Neil, Justin and Gerald

This open source online collaboration thing is like totally awesome.

About a year and a half back, I got an email from someone nine time zones from here who requested anonymity. He had seen that I was struggling with porting the emulator to Windows and sent me some suggestions and patches. He then inquired about some of the outstanding ISOLTS issues and asked for instructions on running ISOLTS. I sent those off to him, and then he proceeded to send me a detailed analysis of the undocumented ISOLTS command shell, hundreds and hundreds of lines of patches, detailed disassembly of ISOLTS tests; single handedly doing more to bring the emulator into ISOLTS compliance than the rest of us combined, despite the fact that Multics, DPS8/M and even mainframe computers was new territory for him.

My bucket list now includes traveling to Europe and buying him a beer.

All of the DPS8/M developers, past and present. This was very much a team effort, all of them have made important contributions.

The Multicians for their patience, contributions of knowledge and insight, and their support.

Bull for releasing the Multics source code, without which this would have not been possible.

The simh emulator framework.

Sourceforge and Wikidot for hosting code and documentation.

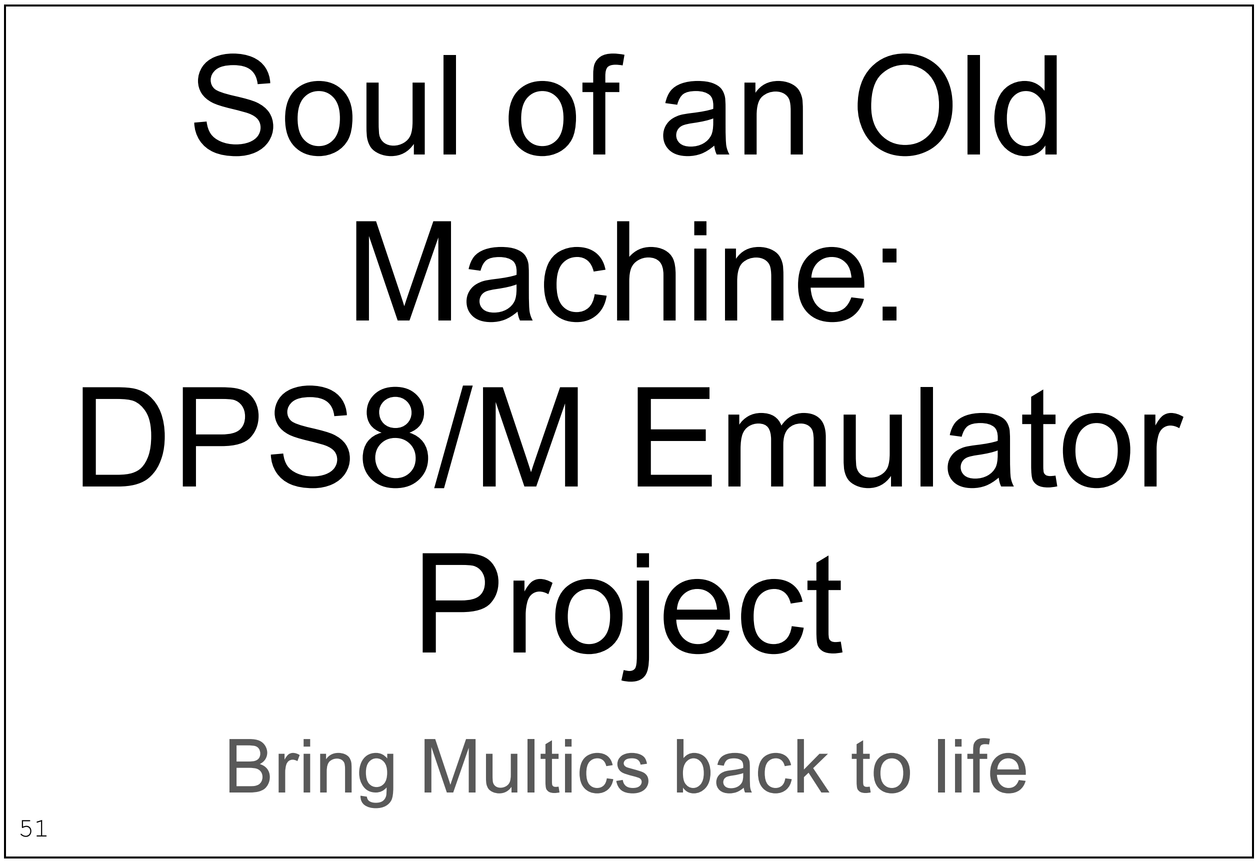
The classic computers mailing list is a valuable source of insight into mainframe technology, especially peripheral interface protocols and floating point implementations.

The Living Computer Museum + Labs for the fantastic opportunity to help bring actual hardware to life.

Over the years, the libraries and coffee houses that have tolerated my sitting all day and typing away.

Damon and Lily for graphics, editorial and personal support.

To my friends who have supported (or at least tolerated) me.



Soul of an Old Machine: DPS8/M Emulator Project

Bring Multics back to life

51

This has been my journey with the dps8m emulator; it has been an amazing experience. I have met some incredible people, learned much about emulation technology, mainframe computing, different approaches to operating systems, and participated in the achieving of the believed impossible goal of restoring Multics.

Thank you for listening to me talk about it.

Questions?

Multics wiki: http://swenson.org/multics_wiki/

DPS8/M Emulator: http://ringzero.wikidot.com/

Multicians: http://multicians.org/

XXXX QR codes for the links? XXXX

52

XXXX Notes to myself, not part of the talk XXX

Links:

Mondy wiki
https://github.com/MichaelMondy/multics-emul/wiki

Multics systems map
https://www.google.com/maps/d/viewer?hl=en&mid=1bBcdkHAE6K-om4qmH0RTAHID_rY&ll=1.2303740432473407%2C-130.73094509999999&z=1

Orangesquid
https://sourceforge.net/p/h6180/code/HEAD/tree/