MALCOLM A. MUMME
P.O. Box 85062
Tucson, AZ, 85754-4004
(951) 808-5580
http://www.cs.ucr.edu/~mummem/
mamumme@acm.org

SUMMARY of QUALIFICATIONS

- Software engineer since 1980 with experience in simulations, embedded real-time systems, microcoding, DSP, processor design, systems programming, theory.
- Many programming languages, applications and operating systems (see SKILLS).
- Committed to high technical standards.
- Likes math, physics, parallel processing, skiing, skating, sailing, kayaking.
- Can work autonomously, independently or in teams. Self-motivated.

EDUCATION

University of California at Riverside, Riverside, California (UCR)

PhD in Computer Science (specialized in formal methods/verification)	(Expected) 2020
Master of Science degree in Computer Science (MSCS)	December 2008
Bachelor of Science degree in Computer Science (BSCS)	June 2004
University of California at Riverside (UCR)	90 quarter units, GPA 3.461
Pasadena City College	transferred 50+ semester units

OTHER EDUCATION

Many continuing education classes including: 1980-1993

- Transistor Electronics: JFETs, MOSFETs, BJTs, opamps, linear circuit analysis &c.
- VLSI design: (Mead-Conway book): Stick diagrams, nMOS, cMOS, layout; RISC
- Fault Tolerant Systems Design: Arithmetic codes, TMR, nMR, D-algorithm &c.
- AI
- Expert Systems
- C++ version 2
- Hatley-Pirbhai OOD
- CMI/IPT

Frequent attendance at various IEEE and ACM conferences such as: 1984-2006

- ICPP (International Conference on Parallel Processing)
- IEEE Compcon, Wescon, Southcon
- ACM LFP (Lisp and Functional Programming), FPCA (Functional Programming and computer Architecture), OOPSLA (Object-Oriented

Programming, Systems, Languages, and Applications), Siggraph (Special Interest Group in Computer Graphics)

Continued reading of academic association publications: 1986-2005

- Association for Computing Machinery (ACM) member.
  - Sigplan (Special Interest Group in Programming Languages) notices
  - Siggraph (Special Interest Group in Computer Graphics)
  - Sigops (Special Interest Group in Operating Systems) Operating Systems Review
  - Sigsam (Special Interest Group in Symbolic and Algebraic Manipulation)
- American Association for the Advancement of Science (AAAS) member (until 2005).
  - Science magazine (weekly)

COMPUTER SKILLS

• More at resume end	• Ada	• Haskell
• Advanced data structures/algorithms	• Assembly	• Java
	• C / C++ / C++11	• VHDL

GRADUATE SCHOOL PROJECTS

- [2007] Research for potential advisors during advisor search
  - Research in graph drawing.
    - Algorithms for drawing planar graphs on grids.
      - For the special case of onion graphs with at most 4 vertices per layer, I found a general solution that is optimal in grid space usage for a given number of vertices. (see http://www.cs.ucr.edu/~mummem/onion4/).
  - Research in automatic variable ordering for compact symbolic (decision diagram) representation of discrete models.
    - I devised and implemented a variable ordering heuristic that produces variable orderings very similar to the orders that were manually constructed by modelers.
  - Reconfigurable computing project
    - Sparse matrix multiplication.
    - Did high-level design for a super-scalar-like sparse matrix multiplier to implement in VHDL for single-FPGA on board implementation. Potential advisor then told me not to use VHDL, but to use his C-language translator.
    - Wrote C implementation of a very simplified sparse matrix multiply algorithm. This was structured as pre-processing, followed by a sequence of on-FPGA and off-FPGA steps, followed by post processing, due to the tool limitation of only allowing processing of a single input stream of a single numeric type.
    - Found about 15 bugs in C-language VHDL translator.
- [2008-2011] Research in symbolic methods (using decision diagrams) for bisimulation on discrete models (minimization of transition systems (automata)).

- [2008] MS Project.
  - Motivated by the desire to produce an improved symbolic lumping (minimization of stochastic automata) algorithm, my advisor assigned me the project of first producing an improved bisimulation algorithm. I devised and implemented two symbolic bisimulation algorithms and did a comparison.
  - The first algorithm (B) was designed to use a more compact representation of transition systems than state-of-the-art lumping algorithms.
  - The second algorithm (H) was designed to mimic a state-of-the-art lumping algorithm using a more compact representation of the transition relation only.
  - Algorithm B turned out to be the same as a well-known classical symbolic bisimulation algorithm, which was unknown to us because it apparently had never been adapted for the lumping problem. It did not produce improved bisimulation run-times, although in some cases it had improved memory usage.
- [2008-2009] PhD Qualifier Project.
  - I found a way to apply the saturation heuristic to the special case of the bisimulation problem where all transition relations are both visible and deterministic, producing a new symbolic bisimulation algorithm (SD).
  - The new algorithm SD delivers reasonable run-time and space performance compared with algorithm H when applied to automata with small minimizations, while delivering significantly better performance with large automata having large minimizations.
- [2009-2011] ~~Current~~ Incremental Research.
  - I refined my implementations of algorithm SD and of the fastest state-of-the-art symbolic bisimulation algorithm (W), and compared performance using various models.
  - I found a generalization SN of algorithm SD, which handles the more general case of the bisimulation problem where some transition relations are non-deterministic. This algorithm can also be generalized to handle the special case of weak bisimulation, where some transitions are invisible. Implemented and characterized performance of algorithm SN, improving state-of-the-art in symbolic bisimulation algorithms for asynchronous labeled transition systems.
  - ~~We are currently preparing a modernized decision diagram manipulation library with improved performance and functionality. I expect to also make substantial API improvements, which will allow a wider variety of researchers to participate in the design and implementation of symbolic algorithms~~. Never Happened! Sad!
  - ~~I expect to generalize algorithm SN to apply to the symbolic lumping problem~~. Never Happened! Sad!
  - Published paper [1] on algorithm SD.
  - Wrote invited paper [2] on algorithms SD and SN.

❑ [2012-2013] Research in programming language design for first PhD proposal.

- [2012] Designed formal specification language to support metaprogramming with higher-level logic programming.
- [2012] Designed new type system for Ephemeral programming language to overcome previous limitations.

- [2012] Wrote 83-page research proposal.
- [2013] Finished detailed proof that f x x = x implies (in a certain theory of equivalence of lambda-terms) either f=(λx. λy. x) or f=(λx. λy. y), i.e. f ∈Boolean.
- [2013] Showed that Simon Payton-Jones's "free theorems" method provides basic axioms about simple types, as duals of the above kind of result.
  - For Booleans, Boolean = t → (t → t) implies ∀f∈Boolean: f x x = x
  - For Naturals, Natural = (t → t) → (t →t) implies ∀n∈Natural: n+1 = 1+n
- [2013] Presented dissertation proposal to committee
- [2013] Changed topic after proposal not approved

❑ [2012-2020] Research in symbolic methods (improving advanced data structures) for final PhD proposal.

- [2012] Invented (Generalized Decision) Diagram ((GD)D), prototype to support expected effort to improve decision diagram manipulation library.
- [2013-2014] Changed topic to symbolic methods.
  - Invented (Generalized Decision) Diagrams ((GD)Ds).
  - Proved canonicity for (GD)Ds.
  - Wrote 68-page dissertation proposal, which was approved after my committee changed the task list to:
    1. Write and submitted short paper on (GD)Ds
    2. Implement (GD)Ds
    3. Test performance of (GD)D implementation for algorithm SN
    4. Write dissertation describing results from tasks 2 and 3
  - Immediately wrote and submitted short paper on (GD)Ds, as required by committee. It which was not accepted due to lack of results.
- [2015] Research on Variable-Labeled Decision Diagrams (VLDDs). VLDDs are defined very similarly to (GD)Ds, but the definition is simpler in part.
  - Implemented VLDDs by writing 10000^+ lines C++ code in 5 months.
  - Implemented algorithm SN on VLDDs and on MDDs.
  - Compared VLDDs with MDDs by performance on algorithm SN.
- [2015-2017] Wrote 517-page dissertation
  - Proved canonicity for VLDDs
  - Proved VLDDs and (GD)Ds are the same.
- [2017-2018] Occasional dissertation updates.
- [2017-2020] Waiting for advisor to finish reading dissertation.
- [2018-2020] Proceeding with activities in 'Future Work' section of dissertation.

TEACHING EXPERIENCE

❑ [2017-2018] Adjunct instructor at California Baptist University, Computer Software and Data Systems Department.

- [2017 Fall] CSC522: Software Development Methodology.
  - Official Course Description: This course gives detailed coverage to significant software development methodologies including traditional plan driven methods, lean methodologies and a particular emphasis on agile methods.

Comparison and discussion of traditional and newer lightweight methods will be made throughout the course. The course may include one or more projects to gain experience practicing software development methods. (3 units; Fall)

- [2018 Spring] CSC524: Web Application Development.
  - Official Course Description: The design and development of data driven web applications. The integration and exploitation of HTML, JavaScript, server-side programming languages and database technology. (3 units; Spring)
- [2018 Spring] CSC526: Software Systems Design.
  - Official Course Description: An in-depth look at software design. Study of design patterns, frameworks, and architectures. Survey of current middleware architectures. Component based design. Measurement theory and appropriate use of metrics in design. Designing for qualities such as performance, safety, security, reusability, reliability, etc. Measuring internal qualities and complexity of software. Evaluation and evolution of designs. Basics of software evolution, reengineering, and reverse engineering. Prerequisite: CSC527 (3 units; Spring)
- [2018 Spring] EGR326: Software Design and Architecture.
  - Official Course Description: An in-depth look at software design. Study of design patterns, frameworks, and architectures. Survey of current middleware architectures. Component based design. Measurement theory and appropriate use of metrics in design. Designing for qualities such as performance, safety, security, reusability, reliability, etc. Measuring internal qualities and complexity of software. Evaluation and evolution of designs. Basics of software evolution, reengineering, and reverse engineering. Prerequisite: EGR 327. (3 units; Spring)
- [2018 Spring] EGR424: Web Applications Development.
  - Official Course Description: The design and development of data driven web applications. The integration and exploitation of HTML, JavaScript, server-side programming languages and database technology. Prerequisite: EGR 325. (3 units; Spring)
- [2018 Spring] EGR506: Engineering Research and Development Methods.
  - Official Course Description: This course is an introduction to research and development in the fields of engineering. Methods for properly researching a topic, collecting and processing data, drawing conclusions and presenting results are discussed. Special attention is paid to the process of technical development as opposed to research. Co-requisite: EGR501 (3 units; Spring/Summer)

❑ [2016-2017] Tutor at Sylvan Learning Systems of Riverside

- Tutoring K-12 students in Reading, Math, and AP subjects.

❑ [2005-2014] Assistant teaching at University of California at Riverside

- [2014] Taught lab for CS008: Introduction to computing.
- [2013] Taught lab for CS010: Intro to programming.
- [2013] Graded for CS246: Advanced Verification Techniques in Software Engineering.
- [2011] Taught lab for CS012: Intro to Programming II.

- [2011] Taught lab for CS008.
- [2010] Taught lab for CS061: Machine Organization and assembly language programming.
- [2010] Taught lab for CS008.
- [2010] Taught lab for CS181: Principles of Programming Languages.
- [2009] Taught lab for CS180: Introduction to Software Engineering.
- [2007] Taught lab for CS179E: Senior Project in Computer Science: Compilers.
- [2006] Taught lab for CS152: Compiler Design.
- [2006] Taught lab for CS008.
- [2005] Taught lab for CS008.

PERSONAL PROJECTS

- ❑ [1979-] Research in computer graphics and parallel processing algorithms.
- ❑ [1983-] Research in programming language design and formal specification for parallel processing.
  - [1983-] Project for the design of a general-purpose programming language supporting formal specification and parallel processing (among other things).
    - [1983-1985] I studied lambda calculus, graph theory, set theory, and class theory, and did a fair amount of research in lambda calculus, for the purpose of providing simple and powerful foundations for the programming language.
    - [1986-1988] I decided to use a set theoretic foundation, and, in 1988, decided it was time to start implementing something.
    - [1988-1988] I then started the process of obtaining approval (from employer Hughes Aircraft) to proceed independently with this research on my own time, with the product remaining my own property.
    - [1989-1989] After some waiting and reflection, I decided to instead use a pure class theoretic foundation (TGF Classes!).
    - [1988-1997] I held back on development, while waiting for IP waivers. I spent some of that time on computer graphics algorithms for affine fractal rendering.
    - [1998-1998] I finally obtained the desired approval in 1998 (shortly after acquisition by Raytheon Systems).
    - [1998-1999] After a "plant site closure" layoff in late 1998, I held back development another year, due to layoff package agreements.
    - [1999-2007] After starting school again in 1999, development has been mostly on hold except for an attempt in 2012 to do this as my PhD research.
    - [2008] ~~Research is finally resuming, due to my advisor allowing me to do this as my main project, allowing me to start it before being advanced to PhD candidacy.~~ Never Happened! Sad!
  - [2003] Simple low level programming language design for parallel processing, for details see the links listed below: "CS179e Project in compilers" on my web page (http://www.cs.ucr.edu/~mummem/index2004.html).
    - The Ephemeral language is intended to be a low-level ISA-independent topology-independent target language for parallel processing compilers.

- It is also intended to be operable on machines with unusually simple processors that are incapable of supporting operating systems as we know them.
- By supporting the programming of massively parallel aggregations of extremely simple machines, it should be able to support wafer-scale parallel processors (when such things become available).
- During this project, I was able to run some toy problems, but it turned out that the type system was insufficient to support serious programming. Fortunately, I have found a reasonably workable fix, which I hope to implement soon.

❑ [1988] Combinator-reduction based macro expander on *Apple][e* in Pascal

- I implemented this in compiled Kyan™ pascal. The small size of the machine required dividing this program into about 10 passes.
  - Tokenizing/ include file processing
  - Token sorting/numbering
  - Parsing
  - Usage analysis
  - Serialized combinator tree production
  - List space construction and initialization with combinator tree
  - Combinator reduction in list space
  - Result serialization from list space
  - Unique token generation
  - De-tokenization/output file generation
  - Or, error message generation
- In the 3 passes involving list space, the memory available for storing the combinator tree was so small that I had to implement a sort of virtual memory, with the combinator space stored in a file, and the memory used as a cache for the file.
- I later ported this program to Turbo Pascal for PC DOS.

❑ [1989] Wrote CGI code (ray tracing, radiosity) on *Apple][e* in assembly/Pascal.

- Used compiled Kyan™ pascal and 6502 assembly language.

❑ [1990] Custom memory card to support macro expander on *Apple][e*

- The *Apple][e* is an 8-bit machine with 16-bit addresses. The resulting space crunch results in considerable "virtual memory" thrashing when running my macro expander.
- The memory card I build provides convenient access to an additional 32K list cells, each of which is 8 bytes wide. It all fits on a standard sized *Apple][* daughterboard, which fits in an I/O slot and is accessed through 16 bytes of memory mapped I/O.
- The user program puts the desired cell number into the upper 2 bytes of the I/O space, and the circuitry provides read/write access to the cell through the lower 8 bytes of the I/O space.
- I wrote the memory test program for this card, which found a problem. After changing one wire, everything worked perfectly. This was good, since I had no logic analyzer available at the time.

❑ [2002] Multi-user client-server flight simulator in Java

- PowerPoint presentation accessible from html resume at my web page

- Host runs simulation server, which will accept multiple clients through TCP sockets
- The simulation server runs a buffering thread for each client.
- The simulation server simulates all objects and provides all inter-client communication.
- Each user runs a client applet, which provides the GUI for that user and opens TCP connection to the server.
- The client applet runs a buffering thread and a rendering thread. The buffering thread accumulates simulation updates while the rendering thread creates a display of the pilot's front window view and a radar display. User controls activate callbacks which send messages to the server.

❑ [2006] Blackboard tool for explaining special theory of relativity.

- This is a (stand-alone) Java program designed to act as a chalkboard for explaining resolution of special relativity paradoxes.
  - The GUI contains 2 parts, (1) drawing area on which user can draw space-time diagrams (2) control area with sliders with which user can adjust the viewers reference frame, and buttons for selecting functions.
  - The GUI is implemented using the old AWT.
  - The operation is approximately according to MVC, with all state stored in a simple database.
- I wrote this because I wanted a relatively easy to explain special relativity to my non-technical friends.
  - The program allows one to draw a space-time diagram of some situation.
  - The user can adjust the scale parameters of the drawing so that relativistic effects become significant.
  - The user can adjust the velocity of the viewer's reference frame, causing the program to perform the Lorenz transform graphically on the diagram.

INDUSTRIAL EXPERIENCE

Other See TEACHING EXPERIENCE and PROJECTS for years: 1999-present

Software Engineer:Hughes Aircraft Company / Raytheon, El Segundo CA 1980-1998
Performed all aspects of software development including concept evolution, system requirements, design, documentation, coding, test, integration, delivery, and maintenance.

❑ Progressively given more complex technical duties and promoted to MTS 2. Tasks here are listed in reverse chronological order.

❑ Improved real-time display simulator.

- Coded simulated pilot cursor operation. This is about simulating the pilot's use of the controls to make the cursor go to a particular place on the screen. This would be trivial, except that in this case, the controls must be manipulated by a program and must work with a variety of different systems. The different systems provide different agility of cursor movement and dependence on the controls. The control program must

experiment with how to control the cursor on a particular system and learn to manipulate the cursor by observing it on a simulated display.
- Made various proprietary simulated display improvements

❑ Simplified H/W design for the target echo simulation processor (DTG).
- My knowledge of digital logic design and the previous generation DTG allowed me to make suggestions that saved gate array space and complexity. My knowledge of programmable processor design principles allowed me to oppose an improper flexibility limitation which would have made it nearly impossible to implement one of the important DTG functions of which I was unaware at the time. At first, the design committee resisted and I looked like a fool because I knew of no specific case where the limitation would be fatal, but I eventually constructed an example of something the previous architecture could not handle well, and then the chief scientist recognized that it applied to the required DTG functionality.

❑ Analyzed, debugged and patched legacy microcode.
- Manually calculating vector lengths from bits displayed on a logic analyzer is not convenient, but it works for debugging simulator microcode.

❑ Translated legacy microcode to C code using a tool I implemented.
- When you have to debug something with 128-bit wide instructions and about 20 fields and about 7 functional units, a disassembler just isn't enough. It actually took about a month to get this working well enough, since there were various timing dependencies among the various functional units, and some things were partially pipelined (and 7 functional units means a lot of opcode and operand tables).

❑ Optimized and coded radar clutter profile simulation algorithm.
- A radar clutter profile is a function that indicates clutter magnitude at any given distance. As perceived by the radar, the effective clutter magnitude is also influenced by clutter distance itself. For a variety of performance and cost reasons, a simple direct calculation of clutter is not used in clutter simulation. Instead it is necessary to determine a partially processed version of radar clutter to be inserted into the simulation at an appropriate point. Although this can increase coding complexity, this can improve performance. The details are proprietary, but I did manage to make an improvement while recoding the algorithm.

❑ Recoded major parts of a Systolic Cellular Array Processor (SCAP) instruction level (IL) simulator according to SCAP design changes.
- The Hughes SCAP underwent some interesting design changes on its way to physical realization. I was called to take over this simulator after the original engineer made a career optimization. The SCAP is a fairly complex processor with many proprietary parts. Its GUI is similarly complex, with an amusingly large number of window panes. I made various changes to the simulator and its X-windows GUI corresponding to the SCAP design changes. These included changes to the control pipeline, IEEE arithmetic exception handling and various proprietary things.
- I also went through the entire program and explicitly distinguished the simulated machine state from the rest of the program state. I then used

  this information to implement a state save-and-restore function, which allowed simulations to be suspended and restarted conveniently. It was not convenient to extract this information after the simulator was already written, but having a working save-and-restore function greatly simplified my design of the test suite, which I also wrote.
  - I went on to propose a project, which would allow state save-and-restore function to be implemented in a scan-chain compatible fashion, allowing a simulation state to be saved and then loaded to an actual processor (and in reverse). This would allow actual processors to be used as simulation accelerators, and simulators to be used to inspect actual processor states. You can guess whether or not this plan was ever approved.

- Inspected SCAP array chip logic diagrams and discovered numerous errors, allowing the designer to correct them before extensive logic simulation began; implemented new parts of IL simulator from corrected logic diagrams.
  - I want to point out here that my analysis and inspection skills are not limited to program code, but also extend to digital logic design blueprints. The SCAP details are proprietary.
  - I converted the corrected logic diagrams into Haskell formulas and wrote a special Haskell function set to convert these to C code. I then integrated the generated C code into the existing simulator. C++ probably would have sufficed instead of Haskell, but it was not conveniently available to me at the time.
- Integrated and tested real-time interrupt driven embedded C and assembly code I designed and coded, in time for contractual acceptance testing and delivery.
  - This was an instrumentation/monitoring package implemented in a Multibus box in C and 80286 assembly without an OS. I primarily designed and coded the part of this package that is required to periodically send data on a dual 1553B bus to a collection process on another system.
  - I made a design to meet the requirements that worked in an asynchronous buffer-management style and then converted the design to operate (with more limitations) in the "cyclic executive" style with double buffering.
  - I wrote the code that generated I/O command lists to be processed by the 1553B bus interface processor. I also wrote the interrupt handler which handles 1553B bus interface interrupts. This code deals with bus failures and controls switching between the two redundant 1553B buses.
  - I (and the rest of the team) integrated, debugged, tested and certified this and other code in accordance with contractual schedules.
- Designed algorithms and performed coding and timing studies for signal and image processing algorithm cores for each of several proposed high-performance processors of various proprietary designs.
- Implemented mission reliability calculations, screen design, and database access for fault-tolerant systems mission reliability tool on HP3000 and VAX.
  - The VAX part of this involved VAX FORTRAN coding, VAX FMS for user input of parts descriptions, reliability data, redundancy descriptions &c., and VAX RMS for data base access.

    - The primary effort here was the conversion of reliability databases among various formats between various machines.
    - Designed various efficient probability calculations (secondary effort).
- Delivered second version of Data Flow Analyzer (DFA) to special programs after designing, coding and debugging the data-flow-analysis code and debugging the integrated DFA.
    - Background on the DFA:
        - The DFA helps DSP application coders deal with the complexity of allocating resources (registers and memory) on the Hughes Programmable Signal Processor (PSP). The PSP was a proprietary high-performance custom computer used in embedded avionic DSP applications. The PSP had a very complex design with hundreds of special signal processing instructions, several banks of various kinds of memory, several explicit pipeline stages, several ways of moving data from one kind of memory to another. Due to the difficulty of constructing a usable compiler to efficiently target such a machine, the PSP was hand-coded in assembly language. Programmers had to choose the location of each variable, manually reorder instructions to properly use various proprietary processor resources, and synchronize their code with the asynchronous I/O subsystem. One of the difficulties with PSP programming is that systems are provided with just enough resources for the computation. Sometimes the memories were just barely sufficient to hold necessary data as long as they were used *carefully*. It was difficult to use them well, since there were various constraints relating various instruction parameters to various memories, constraints between instructions in various pipeline stages, constraints requiring certain variables to be placed in memories corresponding to certain other variables etc.
        - Using the DFA improved the programming practice through input language improvements and by providing useful services, such as: allocation of memory and register locations for storing variables (scalar and vector), analysis of code to automatically identify resource hazards caused by instruction ordering errors, as well as generation of some control opcodes and I/O control opcodes.
        - The DFA input language allows a better coding style to be used without compromising code performance. It improved upon plain assembly coding by providing language features such as "block-structured" control constructs (i.e. if-then-else, do-while, etc.), nested blocks for control of variable scopes, and code pre-processing.
        - The PSP application programmer leaves the most difficult variable allocation decisions open, by means of special syntax. The DFA then attempts to determine if these remaining allocations can be chosen in a way that also meets the special PSP constraints and satisfies the storage requirements of the program. Data flow analysis often finds that the actual scope of a variable is much smaller than its syntactical scope, reducing constraints on where it may be stored. Many times,

    - the results of data flow analysis provide the flexibility necessary to allow a feasible allocation to be found for all program variables.
  - The module I wrote (Data flow analysis) supported automated allocation of memory and register locations by determining the exact scope of variables and determining aggressive but correct space-sharing constraints between pairs of variables. My data flow analysis code performed what was then the "state of the art" in flow analysis. Now we just call it set-use analysis with some proprietary augmentations (data flow analysis also modeled the interaction of the program with the asynchronous I/O subsystem).
  - I have done maintenance and debugging in all modules of the DFA.
- ❑ Debugged, modified, maintained and customized incremental assembler (SG) software on an HP1000 and provided CM and user support and education.
  - o Background on SG: Radar application programs were written in assembly language for a custom proprietary radar application processor. SG originated when it took many mainframe hours to assemble and load the complete radar application program, so that small changes were easier to test by hand-assembling the code into an object patch. The heavily-used manual patching process was tedious and error-prone, resulting in unwanted surprises when the patches were integrated into the source code. SG virtually eliminates these surprises by preparing patches in a way that simulates a source code update. SG accepted a list of source code updates, as well as special files generated in the previous full build by the full assembler. SG's output is a set of object code patches, including branches to special spare patching areas, new code, branches back, as well as branches around "deleted" code. Using SG, small changes to the radar application program could be loaded and tested in only a few minutes (until too many changes accumulated).
  - I became responsible for maintaining this program shortly after it was first delivered. It was mostly written in HP1000 assembly language. As there were still a few adjustments to be made, I quickly became skilled at deciphering the intent of assembly code and debugging assembly code. I made the following enhancements over time:
    - Proper handling of multiple patching areas with various overlay rules.
    - Enlargement of the symbol table capacity, which required virtualizing it to use "extended memory area", while it was previously accessed from register pointers.
    - Proper handling of multiple module/file updates from multiple source code updates. This required a fair amount of update file pre-processing and checking, as well as the addition of many command line options and some utility programs.
    - Handling intermediate level updates, where enough changes had accumulated to prevent practical direct usage of the patch assembler, but not enough updates to justify going back to the mainframe for a full source code re-assembly. This involved having the incremental assembler accumulate information, which it used to update the special

files received from the assembler. Additional updates would be made to a special revision of the object code, which was already partially patched. The idea was to create the illusion that an actual source code update and re-assembly had occurred, so that the incremental assembler only had to process newer updates. This program was apparently a little too successful. Fortunately, after about a decade, the mainframe build process came to take only a few minutes. At that point the patch assembler was only needed in remote locations where radar application program changes had to be tested without convenient mainframe access.

- Maintained “real-time” Special Test Equipment control program (STE)
  - Background on STE:
    - The STE controlled the debugging device attached to embedded radar processors. The device could set breakpoints and watchpoints, snoop memory accesses, modify memory, load programs etc.
    - The STE also came to be the primary controller for various other devices and simulated devices as the overall radar systems and their test environments had become more complex after the mid 1970s.
    - The STE ran on an HP1000 RTE system and had several tens of medium sized Fortran modules maintained by about five programmers.
  - My tasks were:
    - Implementing (possibly nested) file input for STE commands, instead of input from only the user input device (like implementing a “source” command for a shell, but not in Unix). Bonus: Simple looping.
    - Enhancing capabilities of various commands per user requests.
    - Implementing commands to control yet another simulated device:
      - Implementing a new STE command with various options, and manually modifying the command parser tables.
      - Implementing a low level “driver” to access the I/O card that was connected to the simulated device.
      - Implementing the in-between code.
      - Test and integration with the device simulator.
    - Attending various reviews and analyzing code.
    - Repairing various commands according to user requests.

PATENTS

- Processor architectures with patents: (see: http://patft.uspto.gov/netahtml/PTO/srchnum.htm)
- #5,379,444 “simplified synchronous mesh processor and array” (http://www.google.com/patents/US5379444)
  - This processor design is nearly the most extreme example of a minimal SIMD processor design. Here, I’ll just describe the preferred variant of the design. There are four main parts to the design:
    - 1 The control processor: a “standard” SISD processor sends commands to all other parts.

- 2 The main processor cell array: a 2D (rectangular) array of very simple processor cells. Each processor cell receives a single bit input from each neighbor cell on the x axis and on the y axis, for a total of 4 single bit inputs. All cells in the array also receive the same opcode from the control processor. Each processor also receives an x-enable signal and a y-enable signal from parts 3 and 4 respectively. The processor cells internals are described later.
- 3 An x-enable generator, generates a vector of signals "along the x axis", to be distributed to the processor cells.
- 4 A y-enable generator, similar to the x-enable generator, but on the other axis.

- Part 2, the main processor cell array, is where most of the data processing takes place. Consider each processor cell to be associated with a pair of natural numbers n,m. The single bit inputs to this processor cell are from its four nearest neighbors: n-1,m and n+1,m and n,m-1 and finally n,m+1. At the boundaries of the array, some inputs may be absent, or may be connected to system inputs.
- Each processor cell in the array contains (exactly) one single bit of storage. This is the most unique feature of this processor design and produces most of the simplicity. The content of the single bit of storage is output from each cell and is provided as the input to its four nearest neighbors. The array is synchronous, and, on each clock cycle, a new value for the bit may be computed by the cell from the neighbor's inputs and from the "old" value of the bit. This computation depends on the opcode received from the control processor and is conditioned on the x-enable and y-enable signals both being ones.
- In parts 3 and 4, each signal in the vector of generated signals is associated with a natural number on the relevant axis. For example, each signal supplied by the x-enable generator is associated with a natural number n, while each signal from the y-enable generator is associated with a natural number m. In this case, x-enable n and y-enable m are the enable signals provided to processor cell n,m. This is necessary in such a simple design, to provide a rectangular array of enable signals to the processor cells. The internal design of the enable signal generators is not specified in the patent but is assumed to be sufficiently versatile to provide any needed patterns.
- Given that each cell has only 4 data inputs and the "current" value of its bit with which to compute results, the set of possible instruction sets for the cell is relatively small compared with most other processors. The most powerful possible instruction set would be as follows:
  - A 32-bit opcode provided by the main processor would designate a 5-input binary function. This can be implemented easily as a 32-1 mux, with the 5 data inputs serving as the selector control inputs of the mux, and the opcode connected to the 32 data input positions of the mux. Although simple and powerful, this design has two main problems:
    - 1: size: The relatively large size of a 32-1 mux compared to the size of the flip-flop that implements the stored bit means that the processor cell will be mostly occupied by the one large mux. The data storage density of the array will be extremely small.
    - 2: size: A 32-bit wide instruction path must be present throughout the array. This means that the array chip would be occupied mostly by instruction

    wiring, leading to a surprisingly low processor density, further lowering the data storage density. The cells' logic would be farther from neighbors, increasing the length of data paths and decreasing speed.
    - For these reasons, the following design is preferred:
  - A 6-bit opcode provided by the main processor would be divided into two fields,
    - The 2-bit selector field will be used to select one of the four inputs from nearest neighbor cells. This is easily implemented with a 4-1 mux.
    - The 4-bit function field will designate a binary function of 2 inputs, one of which will be the "current" bit, and the second of which will be the selected bit from one of the four nearest neighbors. This also is easily implemented with a 4-1 mux.
    - Although not as powerful, the preferred design seems to work well for providing a reasonable processor density, while still being sufficiently powerful. Convenient algorithms exist to implement multi-bit arithmetic on groups of cells at reasonable speed.
- The primary advantage of this type of array is that the simplicity of the cells allows one to have a great many of them in a small space. The nearest neighbor data connection scheme assures that all data lines/wires will be short (too bad about the opcode/enable lines). The multitude of cells almost completely makes up for the minimal data/cell ratio. The short wires and simple logic allows the clock to be very fast, compensating for the fact that even "primitive" arithmetic operations must be programmed explicitly.
- At the time of the invention, a major disadvantage was the necessity of using a great many cells on a single chip to obtain the necessary advantages. Custom logic at that time would only permit an array of about 1K-2K cells on a chip. This was enough to do Winnograd-style 31-point FFT's at better than average rates but could not handle larger amounts of data. Fortunately, with Moore's law, and the passage of time, a single custom chip should now be able to accommodate [many more than] 32K-64K processor cells per chip. This design would be very hard to compete against. Presently, the primary disadvantage of this design is that the long control lines, which must be driven with very high-power buffers, given the high clock rate desired, would potentially require a very large power and cooling budget.

❑ #5,815,728 "processor array" (http://www.google.com/patents/US5815728)

- This is a pin reduction scheme for 2D mesh connected processors that have the property that the x-axis inter-processor connections are never used at the same time as the y-axis connections. The main idea is to not allocate processor cells to chips in the usual rectilinear manner (a rectangular group of processors placed on each chip). Instead, an almost diamond shaped region of processors from the array is placed on each chip. The proposed shapes have the property that the perimeter comprises regions that alternately cut through x-axis connections and y-axis connections, so that the x and y axis connections are paired adjacently along the perimeter. Each pair of connections can be implemented with only a single physical pin/pad, by using multiplexers. This can significantly reduce the number of required pins/chip for such processors. The patent (online, see above) has various diagrams that clarify the improvement.

PAPERS

[1] A fully symbolic bisimulation algorithm. In: Delzanno, G., Potapov, I. (editors) Proceedings of the Fifth International Workshop on Reachability Problems, pages 218–230. Springer-Verlag

[2] An efficient fully symbolic bisimulation algorithm for nondeterministic systems. In: International Journal of Foundations of Computer Science, Volume 24, Issue 2, pages 263-282.

ADDITIONAL COMPUTER SKILLS

- Ada
- Advanced data structures and algorithms (decision diagrams)
- Apple Final Cut Pro / Compressor / DVD Studio Pro / DVD scripting / iMovie / iDVD / Quicktime Pro / Hypercard / MacDraw / MacPaint
- Assembly for: AMD 29116 / DEC Alpha / HP 3000, 1000, 21MX / Intel x86, 8080 / WDC 658xx, 6502
- Basic
- C / C++ / C++11
- C shell csh scripting
- DEC RSTS
- DEC VMS usage
- Forth
- FORTRAN
- Haskell
- Hughes SCAP
- Java
- Linux
- Macromedia Flash 5 Animation and Actionscript scripting
- Microsoft Word / Excel / PowerPoint / Access / Windows usage
- NCR GAPP
- Oracle SQL
- Pascal and UCSD
- UIMX Motif X11
- UNIX / Linux
- VHDL
- Z formal specification

Ordered by approximate currency (most recent first):

1. C / C++ / C++11
2. Advanced data structures and algorithms (decision diagrams)
3. ML (class only)
4. UNIX / Linux
5. Quicktime Pro / iMovie
6. Final Cut Pro / Compressor / DVD Studio Pro / DVD scripting
7. Microsoft Word / Excel / PowerPoint / Access / Windows usage
8. Java
9. iMovie / iDVD
10. VHDL
11. Perl (class only)
12. MAYA (class only)
13. Oracle SQL (class only)
14. Macromedia Flash 5 Animation and Actionscript scripting
15. Basic
16. Before year 2000:
17. UIMX Motif X11
18. Ada
19. DEC VMS usage
20. DEC Alpha
21. AMD 29116
22. C shell csh scripting
23. Apple MacDraw
24. Apple HyperCard
25. Apple MacPaint
26. Pascal and UCSD
27. HP 1000 & 21MX
28. FORTRAN
29. WDC 658xx and AMD 6502
30. Forth
31. Hughes SCAP
32. Z Formal Specification
33. Haskell
34. IBM MVS usage
35. Intel x86 and 8080
36. NCR GAPP
37. HP 3000 and MPE
38. DEC RSTS