

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

INTEL CORPORATION,
Petitioner

v.

BiTMICRO LLC,
Patent Owner.

IPR2023-00787
U.S. Patent No. 10,120,694

**DECLARATION OF DONALD ALPERT, PH.D.
IN SUPPORT OF *INTER PARTES* REVIEW OF U.S. 10,120,694**

TABLE OF CONTENTS

I.	INTRODUCTION	1
II.	QUALIFICATIONS AND EXPERTISE	3
III.	COMPENSATION	6
IV.	PREVIOUS EXPERT WITNESS EXPERIENCE.....	6
V.	MATERIALS CONSIDERED AND BASIS OF OPINIONS.....	6
VI.	LEVEL OF ORDINARY SKILL IN THE ART.....	8
VII.	UNDERSTANDING OF RELEVANT LEGAL PRINCIPLES.....	10
VIII.	OVERVIEW/TUTORIAL REGARDING TECHNOLOGY.....	13
	A. Embedded Systems and Their Components.....	13
	B. Power-On Reset and Boot	15
	C. System Registers	17
	D. Direct Memory Access (“DMA”)	18
IX.	THE CHALLENGED ’694 PATENT	25
	A. Summary of the ’694 Patent.....	25
	B. Prosecution History of the ’694 Patent	29
	C. Claim Construction	32
X.	CLAIMS 1, 5, 6, 9, AND 12 ARE OBVIOUS OVER THE COMBINATIONS OF POST, WILSON, AND KAO AND POST, WILSON, KAO, AND SO	33
	A. Post	33
	B. Wilson.....	42
	C. Kao	47
	D. So.....	51
	E. Overview of the Combinations	53
	F. Independent Claim 1	53
	1. Claim 1 Preamble.....	53
	2. Claim 1, Limitation a	54
	3. Claim 1, Limitation b.....	75

a.	a power-on reset (POR) sequencer	75
b.	configured to detect, in the nonvolatile memory, a descriptor used in a boot process.....	92
c.	and configured to copy the descriptor from the nonvolatile memory to the RAM.....	98
4.	Claim 1, Limitation c	101
a.	Wherein the POR sequencer facilitates a boot process.	101
b.	And verifies an integrity of loaded data and code in the RAM.	106
5.	Claim 1, Limitation d.....	109
6.	Claim 1, Limitation e	113
a.	wherein the POR sequencer descriptor includes register information	113
b.	wherein the POR sequencer descriptor includes ... DMA controller descriptors.....	114
7.	Claim 1, Limitation f.....	119
G.	Dependent Claim 5	125
H.	Independent Claim 6	128
1.	Claim 6 Preamble.....	129
2.	Claim 6, Limitation a	129
3.	Claim 6, Limitation b.....	129
4.	Claim 6, Limitation c	130
5.	Claim 6, Limitation d.....	130
I.	Dependent Claim 9	130
J.	Independent Claim 12	131
1.	Claim 12 Preamble.....	131
2.	Claim 12, Limitation a	132
3.	Claim 12, Limitation b.....	134
4.	Claim 12, Limitation c	135

5.	Claim 12, Limitation d.....	135
6.	Claim 12, Limitation e	135
7.	Claim 12, Limitation f.....	136
XI.	CONCLUSION.....	136

I. INTRODUCTION

1. My name is Donald Alpert, and I have been retained by Perkins Coie LLP on behalf of Intel Corporation (“Intel” or “Petitioner”) in the above-referenced *inter partes* review proceeding to evaluate United States Patent No. 10,120,694 to Manlapat et al. (“the ’694 Patent”) against certain references that predate the earliest possible priority date of March 17, 2014 (the “Earliest Priority Date”) for the ’694 Patent. The ’694 Patent is attached as Exhibit INTEL-1001 to the Petitioner’s Petition for *Inter Partes* Review of U.S. Patent No. 10,120,694 based on United States Patent Nos. 8,799,555 (“Post”), 7,861,073 (“Wilson”), 7,490,177 (“Kao”), and 7,526,679 (“So”). I am informed that the Petitioner seeks review of claims 1, 5, 6, 9, and 12 of the ’694 Patent (collectively, the “challenged claims”). I have also been informed that the ’694 Patent is currently assigned to BiTMICRO LLC (“BiTMICRO” or “Patent Owner”). As detailed in this declaration, it is my opinion that each of the challenged claims is rendered obvious by prior art references that predate the Earliest Priority Date.

2. I am also providing a declaration in support of Petitioner’s Petition for *Inter Partes* Review of U.S. Patent No. 9,858,084, which shares the same specification as the ’694 Patent and has highly similar claims. The Petition for *Inter Partes* Review of U.S. Patent No. 9,858,084 relies on the same INTEL-1005, INTEL-1013, and Kao prior art references as discussed in the ’694 Patent Petition.

My declarations for these matters may contain similar content given this similarity of subject matter and, where appropriate, I have incorporated materials from one of my declarations into the other.

3. I am over 18 years of age. I have personal knowledge of the facts stated in this Declaration and if requested by the Patent Trial and Appeal Board (“PTAB” or “Board”), I am prepared to testify competently about my opinions expressed herein.

4. I have reviewed and am familiar with the specification and the claims of the ’694 Patent. In general, I will cite to the specification of a United States patent using the following formats: (Patent No. at Col:Line Number(s)) or (Patent No. at Paragraph Number(s)). For example, the citation (’694 Patent at 1:1-10) points to the ’694 Patent specification at column 1, lines 1-10. Also, for convenience, I use italics to denote limitations from the challenged claims.

5. All of the opinions contained in this Declaration are based on the documents I reviewed and my knowledge and professional judgment. In forming the opinions expressed in this Declaration, I reviewed the documents listed below. I have also reviewed and am familiar with the other materials referred to in this Declaration. To the best of my knowledge, the documents mentioned above are true and accurate copies of what they purport to be and are the kinds of materials that an individual

with expertise in this field at the relevant time period would reasonably rely on in formulating opinions, such as those set forth in this declaration.

6. I have been asked to provide my technical opinions regarding how a person of ordinary skill in the art (“POSITA”) would have understood the claims of the ’694 Patent at the time of the Earliest Priority Date. For purposes of whether the teachings of the prior art anticipate or render obvious the claims of the ’694 Patent, I have been asked to assume the date of March 17, 2014, the filing date of the earliest application in the priority chain of the ’694 Patent, for the analysis in this document. I have also been asked to provide my technical opinions on how concepts in the ’694 Patent specification relate to claim limitations of the ’694 Patent. In reaching the opinions provided herein, I have considered the ’694 Patent, its prosecution history, and the references cited above and have drawn as appropriate on my own education, training, research, knowledge, and personal and professional experience.

II. QUALIFICATIONS AND EXPERTISE

7. I am an independent consultant and the president of Camelback Computer Architecture LLC, through which I have provided computer architecture consulting services for twenty years. In formulating my opinions, I have relied on my knowledge, training, and experience in the relevant field, which I will summarize briefly. A more detailed summary of my background, education, experience, and publications is set forth in my curriculum vitae (“CV”), which is provided as

Appendix A. My CV also includes a list of all other cases in which, during the previous 4 years, I testified as an expert at trial or by deposition.

8. I have 45 years of academic and industrial experience in applying, designing, studying, teaching, and writing about microprocessors and computer systems. I received an Electrical Engineering Ph.D. degree in 1984 from Stanford University. I earlier received an Electrical Engineering B.S. degree from MIT in 1973 and an Electrical Engineering M.S. degree from Stanford University in 1978. I have taught classes in computer architecture at Stanford, Tel Aviv, and Arizona State Universities.

9. From 1976 to 1977, I worked at Burroughs Corporation, where I designed peripheral interface controllers, including those for serial data communications based on Intel 8080 microprocessor components. From 1980 to 1989, I was the lead architect for the design of three high-performance microprocessors at Zilog and National Semiconductor. Later, at Intel, I was the lead architect of the Pentium® Processor from 1989 to 1992 and of the 815 chipset from 1999 to 2000, both of which became the most widely used PC components of their time. The 815 chipset comprised two components: (1) a memory controller hub (MCH) that included a graphics controller and memory controller with interfaces to the CPU, 133 MHz SDRAM system memory modules, an optional, external graphics controller and (2) an I/O controller hub (ICH) that included various I/O controllers

(e.g., network, hard drive, USB) for system peripheral devices, as well as Direct Memory Access (DMA) controllers. Additionally, I served as co-manager for the Itanium processor design from 1993 to 1997.

10. I am a Senior Member of the Institute of Electrical and Electronics Engineers (IEEE), and served as the chair of the IEEE Technical Committee on Microprocessors and Microcomputers from 1999 to 2000. I was the keynote speaker at the first Cool Chips conference, dedicated to the study of low-power microprocessors and systems. I have given invited lectures at several universities, and published ten papers in various professional journals and conference proceedings. My paper entitled “Architecture of the Pentium Processor” was selected as best paper in IEEE Micro for 1993. I am a named inventor on over 30 U.S. patents that pertain to microprocessors, computer systems, and related technology.

11. I have reviewed the '694 patent, and I am familiar with its subject matter, which is within the scope of my education and professional experience. Based at least on my background in academia, industry, and consulting, I am familiar with the issues and technology relating to processors, chipsets, memory, and peripheral devices for computer systems. I have personally analyzed, developed, and tested such computer components and systems. More specifically, the Pentium®

Processor and 815 chipset for which I was the lead architect at Intel implemented DMA controllers and various features for system initialization.

12. My curriculum vitae, including my publications and patents, is submitted herewith as Appendix A.

III. COMPENSATION

13. In connection with my work as an expert, I am being compensated at an hourly rate of \$650 for consulting services including time spent testifying at any hearings that may be held. I am also being reimbursed for reasonable and customary expenses associated with my work in this case. I receive no other form of compensation related to this case. My compensation is not dependent or otherwise contingent upon the specifics of my testimony or the outcome of this action.

IV. PREVIOUS EXPERT WITNESS EXPERIENCE

14. My CV attached as Appendix A to this Declaration contains a list of cases in which I have testified at trial, hearing, or by deposition within the preceding four years.

V. MATERIALS CONSIDERED AND BASIS OF OPINIONS

15. In forming the opinions set forth in this Declaration, I have considered and relied upon my education, experience, and knowledge of the relevant fields. I have also reviewed and considered the '694 Patent and its prosecution history. The '694 Patent and its prosecution history are attached as exhibits INTEL-1001 and

INTEL-1002 to the above-referenced IPR petition. Additionally, I have relied on and reviewed the following documents and materials cited in this Declaration, all of which are attached as exhibits to the above-referenced IPR petition.

Exhibit	Description
INTEL-1001	U.S. Patent 9,858,084 to Manlapat (“the ’084 patent”)
INTEL-1002	File History of U.S. Patent 9,858,084
INTEL-1003	Declaration of Dr. Donald Alpert (this document)
INTEL-1004	U.S. Patent 7,490,177 to Kao (“Kao”)
INTEL-1005	U.S. Patent 8,799,555 to Post et al. (“Post”)
INTEL-1007	<u>Microsoft Computer Dictionary</u> (5th Ed.)
INTEL-1008	J. Catsoulis, <u>Designing Embedded Hardware</u> (2005)
INTEL-1009	J. Ganssle et al., <u>Embedded Hardware</u> (2008) (“Ganssle”)
INTEL-1010	Q. Li, <u>Real-Time Concepts for Embedded Systems</u> (2003)
INTEL-1012	P. Barry et al., <u>Modern Embedded Computing</u> (2012)
INTEL-1013	U.S. Patent 7,861,073 to Wilson (“Wilson”)
INTEL-1014	U.S. Patent No. 7,921,237 (“Holland”)
INTEL-1015	M. Mintz et al., <u>Hardware Verification with SystemVerilog</u> (2007)
INTEL-1016	T. Noergaard, <u>Embedded Systems Architecture</u> (2005)
INTEL-1017	AHB Example AMBA System Technical Reference Manual, ARM, 1999
INTEL-1018	J. Lenk, <u>Simplified Design of Microprocessor Supervisory Circuits</u> (1998)
INTEL-1019	U.S. Patent App. No. 2005/0273588 (“Ong”)
INTEL-1020	Power-On Reset and Related Supervisory Functions, Application Note 327, Maxim Integrated, July 8, 2004

Exhibit	Description
INTEL-1021	U.S. Patent 7,526,679 (“So”)

16. I reserve the right to supplement this Declaration and rely upon any additional information, deposition testimony, documents, or materials that may be provided to me or that are relied upon by any of Respondent’s experts or witnesses.

17. I also reserve my right to supplement this Declaration and rely upon additional information that becomes available to me.

18. In connection with live testimony in this proceeding, should I be asked to provide it, I may rely on visual aids and demonstratives to demonstrate the bases of my opinions, such as claim charts, patent drawings, excerpts from patent specifications, file histories, interrogatory responses, deposition transcripts and exhibits, as well as charts, diagrams, videos and animated or computer-generated video. The demonstrative exhibits I may use to accompany any testimony I may give may also include documents, testimony, and other evidence cited in my Declaration, portions of such evidence, and other portions of my Declaration and the attached exhibits.

VI. LEVEL OF ORDINARY SKILL IN THE ART

19. I am told that certain issues relating to the validity of the ’694 Patent must be judged from the perspective of a person of ordinary skill in the relevant art, as I discuss below. I have been asked to define the level of a “person of ordinary

skill in the art” or “POSITA” at the time the alleged invention as claimed was made, that is, as of the Earliest Priority Date (March 17, 2014). In deciding the level of ordinary skill, I have considered the following factors, which I have been informed are relevant to the determination of ordinary skill in the art:

- i) the levels of education and experience of persons working in the field;
- ii) the types of problems encountered in the art;
- iii) the prior art solutions to these problems;
- iv) the rapidity with which innovations are made; and
- v) the sophistication of the technology.

20. In my opinion, for the purposes of the '694 Patent, a POSITA, at the Earliest Priority Date, would have had a bachelor's degree in electrical engineering or computer engineering and at least two years of experience working in the field of embedded system design, including booting of such systems, or an equivalent combination of education, work, and/or experience. My opinion as to this level of ordinary skill in the art would be the same even if the Board were to determine that the time of the alleged invention is March 15, 2013, which is the filing date of the provisional application to which the '084 Patent asserts priority.

21. My opinion below explains how a POSITA would have understood the technology described in the references I have identified in this declaration as of the Earliest Priority Date.

VII. UNDERSTANDING OF RELEVANT LEGAL PRINCIPLES

22. I am not a lawyer and will not provide any legal opinions. Although I am not a lawyer, I have been advised that certain legal standards are to be applied by technical experts in forming opinions regarding the meaning and validity of patent claims.

23. I understand that a patent claim is invalid if it is anticipated or obvious in view of the prior art, and that a claim can be unpatentable even if all of the requirements of the claim cannot be found in a single prior-art reference. I further understand that invalidity of a claim requires that the claim be anticipated or obvious from the perspective of a person of ordinary skill in the art at the time the invention was made.

24. I have been informed that a patent claim is invalid if it would have been obvious to a person of ordinary skill in the art (“POSITA”). In analyzing the obviousness of a claim, I understand the following factors may be considered: (1) the scope and content of the prior art; (2) the differences between the prior art and the claims; (3) the level of ordinary skill in the art; and (4) any so called “secondary considerations” of non-obviousness if they are present. Such secondary considerations include: commercial success of products covered by the patent claims; a long-felt need for the invention; failed attempts by others to make the invention; copying of the invention by others in the field; unexpected results

achieved by the invention as compared to the closest prior art; praise of the invention by the infringer or others in the field; the taking of licenses under the patent by others; expressions of surprise by experts and those skilled in the art at the making of the invention; and the patentee proceeded contrary to the accepted wisdom of the prior art. I am not aware of any evidence of secondary considerations of non-obviousness relevant to the '694 patent. I reserve the right to supplement this Declaration if Patent Owner introduces evidence of secondary considerations of non-obviousness.

25. I understand that to prove that prior art or a combination of prior art renders a patent obvious, it is necessary to:

- i) identify the particular references that, singly or in combination, make the patent obvious;
- ii) specifically identify which elements of the patent claim appear in each of the asserted references; and
- iii) explain why a person of ordinary skill in the art would have combined the references, and how they would have done so, to create the inventions claimed in the patent.

26. I further understand that exemplary rationales that may support a conclusion of obviousness include:

- i) combining prior art elements according to known methods to yield predictable results;
- ii) simple substitution of one known element for another to obtain predictable results;
- iii) use of known technique(s) to improve similar devices (methods or products) in the same way;
- iv) “obvious to try” – choosing from a finite number of identified, predictable solutions with a reasonable expectation of success;
- v) known work in one field of endeavor may prompt variations of the work for use in either the same field or a different field based on design incentives or other market forces if the variations are predictable to a person of ordinary skill in the art; and
- vi) some teaching, suggestion, or motivating in the prior art that would have led a person of ordinary skill in the art to modify the prior art reference or to combine prior art reference teachings to arrive at the claimed invention.

27. I have been informed that, in considering obviousness, hindsight reasoning derived from the patent-at-issue may not be used.

VIII. OVERVIEW/TUTORIAL REGARDING TECHNOLOGY

A. Embedded Systems and Their Components

28. The phrase “embedded system” is typically used to refer to a microprocessor that is integrated with other equipment to perform a specific function. The Microsoft Computer Dictionary (5th Ed.) defines embedded system as:

Microprocessors used to control devices such as appliances, automobiles, and machines used in business and manufacturing. An embedded system is created to manage a limited number of specific tasks within a larger device or system.

(INTEL-1007, at 190-191.) Other definitions are similar: “embedded systems represent a class of dedicated computer systems designed for specific purposes.”

(INTEL-1010 at 1.) The '694 patent focuses on the inclusion of a processor as the critical component of an embedded system: “The term embedded system will be used to refer to a single CPU or a multiprocessor system in this paper.” (INTEL-1001 at 1:26-28.)

29. Because an embedded system is based on a processor, they also naturally included components found in general-purpose computers, such as memories and storage devices. The prior art identifies processors, volatile RAMs and non-volatile memories as essential components of any embedded system. (INTEL-1009 at 195-209; INTEL-1012 at 61-72.) The treatises described the use of

controllers for RAM and non-volatile memories. (INTEL-1009 at 110, 196-199, 204-205, 208; INTEL-1012 at 62, 66, 72.) They also detailed the use of the standard Serial ATA (SATA) interface to connect hard disks and SSDs (INTEL-1009 at 209-214; INTEL-1012 at 72-73) and USB and PCIe interfaces suitable for connecting storage devices generally (INTEL-1009 at 214; INTEL-1012 at 74-89). In its discussion of the prior art, the '694 patent acknowledges that embedded systems that incorporated RAM memory, non-volatile memories, and mass storage devices were conventional:

A conventional embedded system contains a nonvolatile memory which is used to store firmware that is read and executed by a CPU to initialize the system. The nonvolatile memory can be a read-only memory (ROM), one-time programmable memory (OTP), programmable read only memory (PROM), electrically erasable PROM (EEPROM), or a flash PROM (FPROM). The nonvolatile memory can be internal (on-chip) or external (off-chip) to the controller chip. The firmware stored in the nonvolatile memory can be a boot code loader, a basic input/output system (BIOS), an operating system (OS), or an application firmware. The boot code loader is a small code executed by a CPU to do minimum initialization of the system. After initialization, the boot code loader copies the BIOS or OS or application firmware from a mass storage device external to the embedded system to the random access memory (RAM) of the system.

(INTEL-1001, 1:29-45.)

30. Other conventional features of embedded processors include DMA controllers, storage controllers, a CPU bus connecting the processor and memory

devices, reset controllers, and interfaces for debugging. Each of these was already documented in the prior art. (K. Arnold, Embedded Controller Hardware Design (2001), at 117-124 (CPU bus), 175-178 (DMA controllers); INTEL-1009 at 137-138 (controllers for optical and magnetic storage devices), 166-168 (processor bus), 214-235 (DMA controllers); INTEL-1005, at 556 (reset controller); INTEL-1008 at 157-59 (debugging interface); INTEL-1010, at 51-52 (debugging interfaces); INTEL-1012, at 72-73 (host storage controllers), 469 (debugging interface).)

B. Power-On Reset and Boot

31. In a processor-based system, a *boot process* is used to start the system up. System boot-up takes place after a power-on or reset condition has occurred. (INTEL-1016, 555 (“System boot-up means that some type of power-on or reset source ... has occurred.”); INTEL-1019, ¶0001 (“Upon being powered up, or in response to a reset condition, conventional microcomputers, and other devices controlled by a programmable microprocessor or microcontroller, undergo an initialization process known as “booting.”).) When the system first powers on, it enters a power-on reset state in which internal hardware are reset to a known starting point. (INTEL-1018, 2 (“Power-On Reset. When power is applied to a microprocessor, the internal registers ‘come up’ in arbitrary states that contain random data. Applying a reset command to the microprocessor overcomes this chaos by resetting all internal circuits at a predictable starting point.”).)

32. Start-up firmware called “boot code” is then loaded, typically from system Read-Only Memory. (INTEL-1016, 555 (“When power is applied to an embedded board (because of a reset), start-up code, also referred to as boot code... in the system’s ROM is loaded and executed by the master processor.”); INTEL-1009, 5 (“The primary purpose of ROM within a system is to hold the code (and sometimes data) that needs to be present at power-up. Such software is generally known as firmware and contains software to initialize the computer by placing I/O devices into a known state.”), 16 (“At the lowest level, there are programs that are run by the processor when the computer first powers up. These programs initialize the other hardware subsystems to a known state and configure the computer for correct operation”).)

33. Boot code “length and functionality” differs depending on the particulars of the system. (INTEL-1016, 555.) Regardless of these particulars, however, boot code typically performs the “same (minimal) general functions” that include “initializing the hardware, ... initializing buses, setting the master and slave processors in a specific state, and initializing memory.” (*Id.*; INTEL-1013, 1:25-28 (“The boot-up process allows a microcontroller to perform various actions to prepare the microcontroller for ordinary operation, such as load initialization values in its registers, load instructions in its instruction memory, etc.”).) Once the hardware is initialized, software, including an operating system, may be loaded to complete

system setup. (INTEL-1016, 555 (“After the hardware initialization sequence ... the remaining system software, if any, is then initialized.”); INTEL-1008, 3 (“The bootloader ... reads the operating system from disk ... and places it in memory so that the processor may then run it.”).)

C. System Registers

34. Processor hardware includes *registers*, which are a type of volatile memory. (INTEL-1009, 70-71 (“*Register set*. A register is simply fast programmable memory normally used to store operands that are immediately or frequently used. A processor’s set of registers is commonly referred to as the *register set* or the *register file*.”); INTEL-1016, 137 (same); INTEL-1013, 2:54-56 (“Register file 102 can include various registers which are used by the processor 101 in the ordinary operation of the microcontroller.”).) Different processors will have different register sets; “the number of registers in [processor] sets varies between very few to several hundred (even over a thousand).” (*Id.*; INTEL-1016, 137 (similar); INTEL-1008, 32 (“Registers are the internal (working) storage for the processor. The number of registers varies significantly among processor architectures ... Some processors also have one or more control registers consisting of configuration bits that affect processor operation and the operating modes of various internal subsystems.”).)

35. Registers must be initialized to a known value before they can be used by the processor. (INTEL-1020, 1 (“One task of the power-on reset (POR) is ensuring that the processor starts at a known address when power is first applied.... The POR’s second task is to keep the processor from starting its operation from that known address until ... the internal registers have been properly loaded.”); INTEL-1013, 2:56-59 (“The microcontroller can require that certain specific predefined initial values are stored in one or more of the registers of the register file before the microcontroller assumes ordinary operation.”).) Register initialization values may be included in the boot data and loaded into the register set as part of the boot process. (INTEL-1013, 2:59-63(“Since the register file is usually composed of volatile memory, the initial values can be stored in the register file as part of an initialization or boot operation ... [and] can be a part of boot data that is loaded in the microcontroller during a boot-up procedure.”).)

D. Direct Memory Access (“DMA”)

36. The core processor itself may be responsible for performing “data transfer between internal and external memory spaces.” (INTEL-1009, 214; INTEL-1008, 20 (“DMA allows data to be transferred from I/O devices to memory directly without the continuous involvement of the processor.”). These memory transfer operations require significant processing workload and divert the processor from its various other tasks, thus reducing optimum performance. INTEL-1009, 214 (“in

reality you can only achieve optimum performance in your application if data can move around without constantly bothering the core to perform the transfers”).). An example provided by the 2005 textbook Designing Embedded Hardware (INTEL-1008) explains why the processing workload for such processor-driven data transfers can be high:

Let's say you want to read in 100M from disk and store it in memory. You have two options.

One option is for the processor to read one byte at a time from the disk controller into a register and then store the contents of the register to the appropriate memory location. For each byte transferred, the processor must read an instruction, decode the instruction, read the data, read the next instruction, decode the instruction, and then store the data. Then the process starts over again for the next byte.

INTEL-1008, 20.

37. Direct Memory Access (“DMA”) techniques were one way to offload this work from the core system processor. (*Id.* (“DMA is a way of streamlining transfers of large blocks of data between two sections of memory, or between memory and an I/O device.”).) The continuation of the example provided in the previous paragraph shows how DMA offloads work from the system processor:

Let's say you want to read in 100M from disk and store it in memory. You have two options.

The second option in moving large amounts of data around the system is DMA. A special device, called a *DMA Controller (DMAC)*, performs high-speed transfers between memory and I/O devices. Using DMA bypasses the processor by setting up *a channel* between the I/O device and the memory. Thus, data is read from the I/O device and written into memory without the need to execute code to perform the transfer on a byte-by-byte (or word-by-word) basis.

(Id. (emphasis original).)

38. DMA controllers were “peripheral[s] devoted to moving data around a system” that operated independently from the core processor. (INTEL-1009, 216; INTEL-1016, 280 (“DMA (direct memory access) circuitry can be used to bypass the master CPU entirely. DMA has the ability to manage data transmissions or receptions directly to and from main memory and an I/O device. On some systems, DMA is integrated into the master processor, and on others there is a separate DMA controller. Essentially, DMA requests control of the bus from the master processor.”).) A DMA controller could be configured to transfer data or code stored in one location in memory to another, without requiring the processor to perform the work itself. INTEL-1009, 215 (“You can program the DMA controller to move data in parallel with the core while the core is doing its basic processing tasks—the jobs on which it’s supposed to be focused.”); INTEL-1015, 32 (“DMA is a common hardware feature for transferring data from one memory location to another without putting a load on the CPU.”); INTEL-1008, 20 (“Using DMA bypasses the processor

by setting up a channel between the I/O device and the memory. Thus, data is read from the I/O device and written into memory without the need to execute code to perform the transfer on a byte-by-byte (or word-by-word) basis.”).

39. The 2008 industry reference book Embedded Hardware (INTEL-1009) illustrates typical DMA flows, in which a DMA controller performs transfers between system memory and/or peripherals:

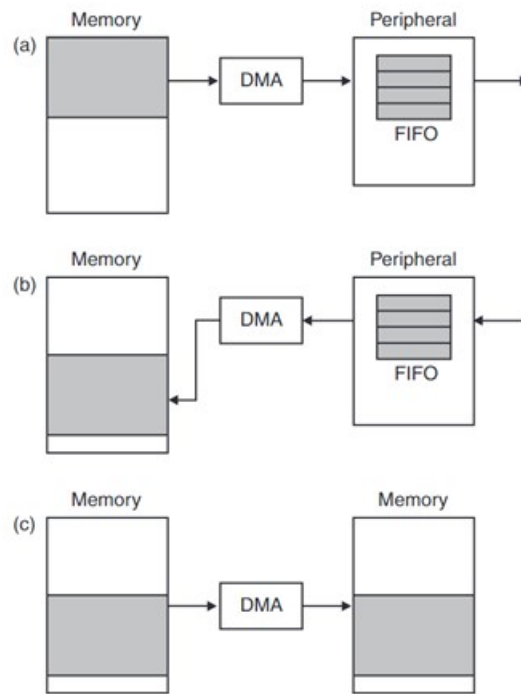


Figure 5.17: Typical DMA flows.

(INTEL-1009, 216.) DMA controllers can perform transfers of code as well as data. (*Id.* (“[A] DMA transfer doesn’t always have to involve data. We can use code overlays to improve performance, configuring the DMA controller to move code

into L1 instruction memory before execution. The code is usually staged in larger external memory.”.)

40. DMA controllers need some basic information to perform a data or code transfer, for example a source and/or destination address for the data or code to be transferred, and an indication of the size of the data or code to be transferred:

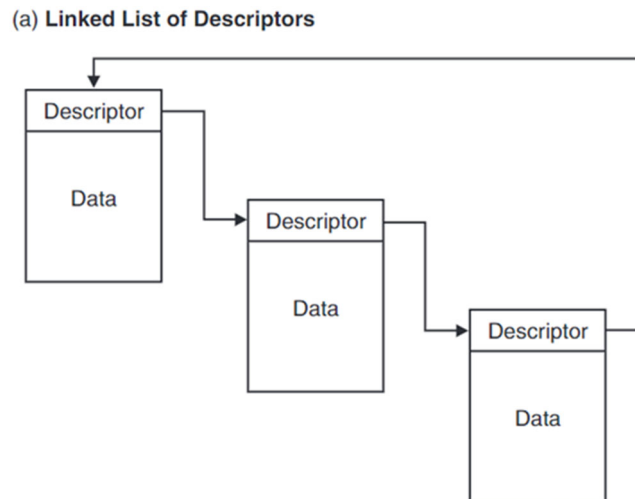
For any type of DMA transfer, we always need to specify starting source and destination addresses for data. In the case of a peripheral DMA, the peripheral’s FIFO serves as either the source or the destination. When the peripheral serves as the source, a memory location (internal or external) serves as the destination address. When the peripheral serves as the destination, a memory location (internal or external) serves as the source address.

In the simplest MemDMA case, we need to tell the DMA controller the source address, the destination address and the number of words to transfer. With a peripheral DMA, we specify either the source or the destination, depending on the direction of the transfer.

(*Id.*, 208; *see also* INTEL-1008, 21 (“A DMA transfer begins by the processor configuring the DMAC for the transfer. This setup involves specifying the source, destination, and size of the data, as well as other parameters.”))

41. DMA controllers could use descriptors stored in memory to initiate a DMA sequence. (INTEL-1009, 231 (“DMA transfers that are descriptor-based require a set of parameters stored within memory to initiate a DMA sequence.”); INTEL-1008, 21 (“Data-chaining transfers [a]llow DMA transfers to be performed as specified by a linked-list in memory. Data chaining is started by specifying a

pointer to a descriptor in memory. The descriptor is a table specifying byte count, source address, destination address, and a pointer to the next descriptor.”.) DMA “descriptors allow[ed] the chaining together of multiple DMA sequences,” where another DMA transfer could be configured to automatically start when a current DMA transfer completed. (INTEL-1009, 231; INTEL-1008, 21.) For example, DMA descriptors could be configured in a “Linked List” format, with each descriptor pointing to the address of the next:



(INTEL-1009, 233 (Fig. 5.28); INTEL-1008, 21.) A “descriptor-based” DMA model thus “provide[d] the most flexibility in managing a system’s DMA transfers.” (INTEL-1009, 231.)

42. Numerous examples of the use of DMA controllers in the boot process, and of DMA descriptors that identify data to be transferred by the DMA controller were known before the earliest filing date of the ’084 patent. For example, U.S.

Patent No. 8,125,243, published on February 28, 2012, used a DMA controller (3:45-46) to transfer device configuration data from a non-volatile memory to device configuration registers (3:62-65, 4:11-17). To carry out these transfers, the DMA controller acted on a series of DMA transaction descriptors stored in memory that identified each of the transfers of configuration data that were to be performed. (4:46-56.) U.S. Patent No. 8,996,926, filed on October 15, 2012, described a DMA controller used to transfer data (1:5-28) or boot code (5:41-44). The DMA controller used a chained set of DMA transaction control values, as illustrated in Figure 3A, that are chained together to define a sequence of transfer operations that the DMA controller was to perform.

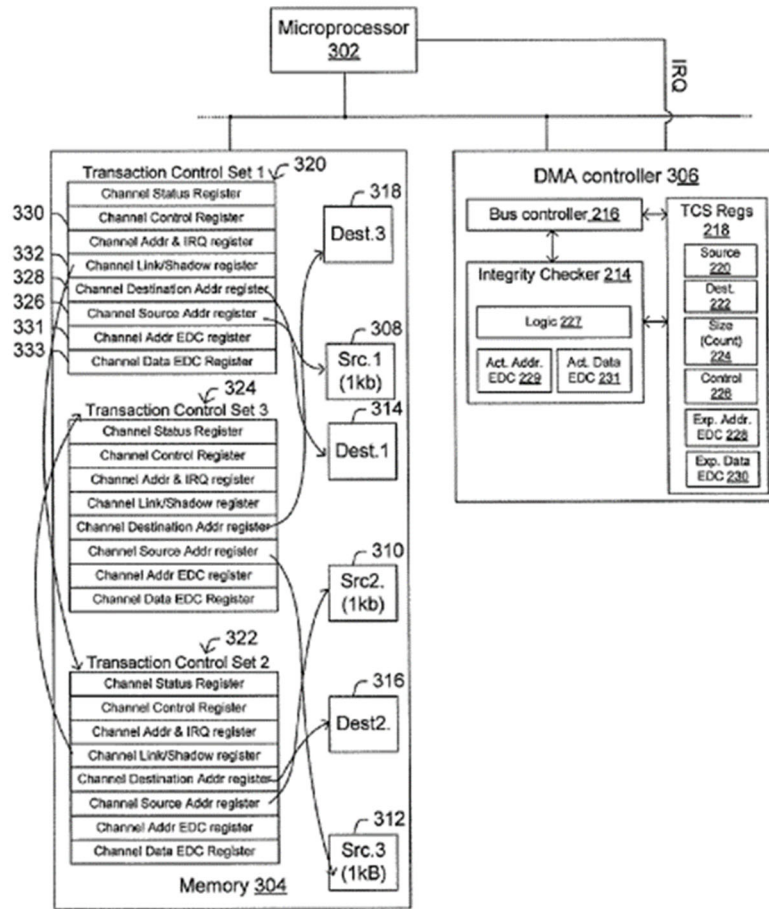


FIG. 3A

(Id.)

IX. THE CHALLENGED '694 PATENT

A. Summary of the '694 Patent

43. The '694 patent describes an embedded system with “a single or multiple Central Processing Units” and is directed to “a method and device for booting such a system directly from a storage device.” INTEL-1001, 1:14-18. As depicted in Figure 2 of the patent, the embedded system includes one or more CPUs

(in red, below), one or DMA controllers (in yellow), and a power on reset (POR) sequencer (in blue) that “makes use of a POR Sequencer Descriptor.” *Id.*, 2:47-48. The POR Sequencer descriptor is “a preassembled descriptor that is stored in the nonvolatile memory of the system.” *Id.*, 2:48-50. It “contains register information and direct memory access (DMA) controller descriptors. The register information are [sic] read and used by the POR sequencer to update the configuration registers of the system. The DMA descriptors are loaded by the POR sequencer to the DMA controllers of the system.” *Id.*, 2:51-56.

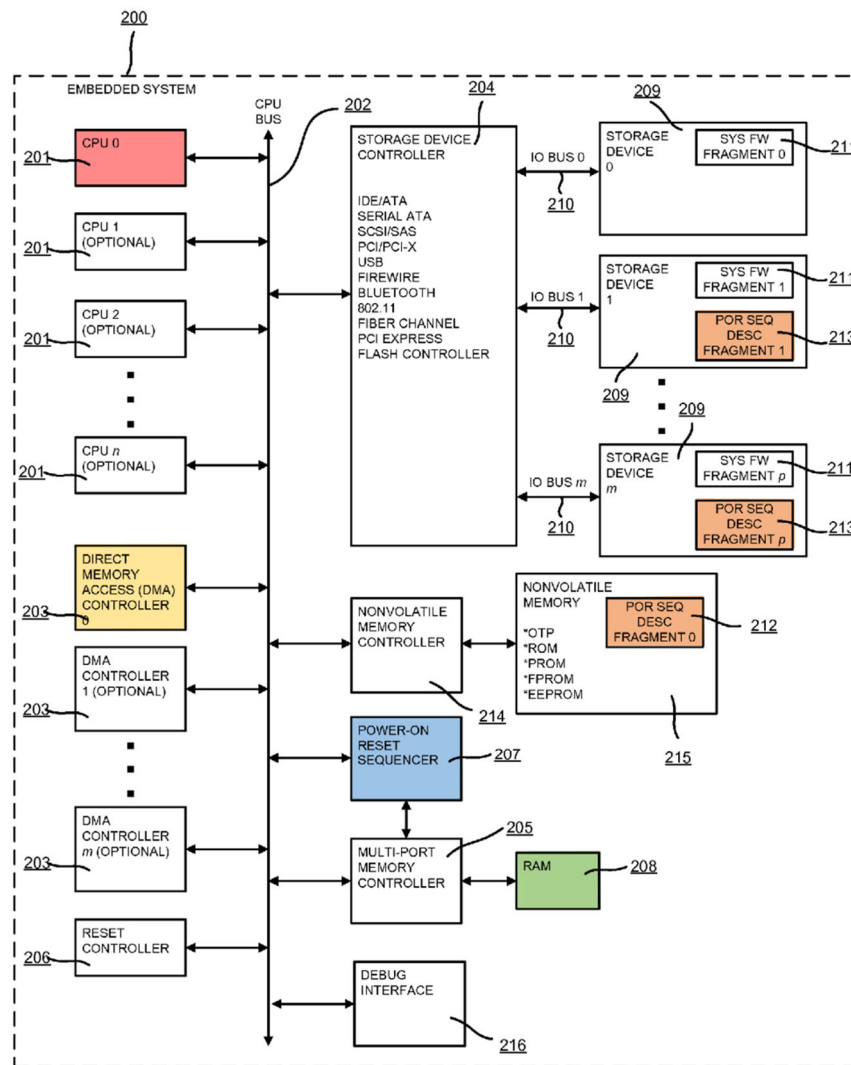


FIG. 2

44. The '694 patent proposes that “[w]hen the POR sequencer descriptor doesn’t fit in the nonvolatile memory of the system, the descriptor is divided into fragments and distributed across the nonvolatile memory and storage devices attached to the system.” *Id.*, 6:32-35.) This is illustrated in Figure 2, which depicts multiple storage devices each containing a fragment of the POR sequencer descriptor. The fragments are connected in a “linked list,” as illustrated in Figure 5,

(*id.*, 4:28-29), where each fragment contains a DMA controller descriptor (in orange) that “points to the next POR sequencer descriptor fragment.” *Id.*, 6:42-44.

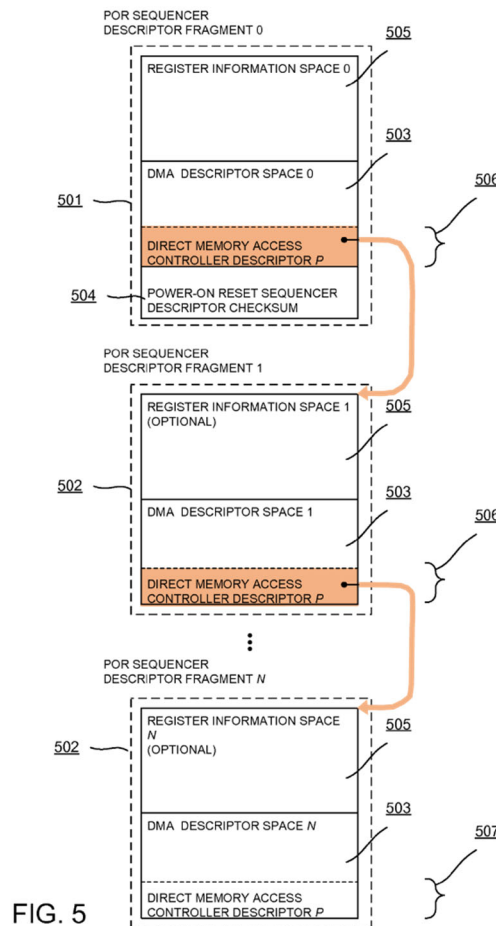


FIG. 5

45. The POR sequencer “controls the overall boot process of the embedded system.” *Id.*, 2:35-36. At system power-on, it, and other system components, are brought out of reset. *Id.*, 6:56-61. The POR sequencer copies the first fragment of the POR sequence descriptor to memory. *Id.*, 6:64-67. It then “checks the integrity of POR sequencer descriptor ... by comparing [a] computed checksum to the POR sequencer descriptor checksum” (*id.*, 6:67-7:3), which is contained, as shown in Figure 5, in the first fragment. *Id.*, 6:38-40. Using the method illustrated in Figure

7 (*id.*, 7:17-19), the POR sequencer copies the register information from the fragment in memory to update the corresponding configuration registers in the embedded system. *Id.*, 7:17-27. When all of the register configuration information has been processed, the POR sequencer writes the DMA descriptor to the DMA controller and activates it so that it will initiate transfer of the of the next POR sequence descriptor fragment from storage to the RAM. *Id.*, 7:27-35. This process continues until the configuration register data in each of the POR sequence descriptor fragments has been processed. *Id.*, 7:37-44. A similar method, illustrated in Figure 8 (*id.*, 7:48-50), is used for retrieving firmware code, which may also be stored in multiple fragments, as shown in Figure 5. *Id.*, 7:45-8:13.

46. The '694 patent states that if a CRC error is found in the configuration register data “a POR sequencer error handler is activated.” *Id.*, 7:5-7, *see also, id.*, 7:67-8:1 (if errors in the firmware are identified “an error handling routine will be executed”). The nature of these error handling routines is not specified. If the POR sequencer cannot find the primary copy of the POR sequencer descriptor “it will try to detect a second copy.” *Id.*, 7:7-9.

B. Prosecution History of the '694 Patent

47. The '694 patent application was filed on March 17, 2014 and examined under the first inventor to file provisions of the AIA. INTEL-1002, 33-36; *id.*, 44, 113. The first office action imposed a restriction requiring Applicant to select

between apparatus claims to an “embedded system” and method claims to a “method of booting up an embedded system.” *Id.*, 44. Applicant elected to pursue the apparatus claims 1-9 and cancelled claims 10-20. *Id.*, 48-51.

48. In his first action on the merits, the Examiner rejected claims 1-2 and 6-9 under 35 U.S.C. 102(a)(1) as being anticipated by Chow et al, US. Patent No. 7,676,640 and objected to claims 3-5. *Id.*, 57-58. In its response, Applicant amended the specification and claims 1 and 3 and added new claims 21-28. *Id.*, 61-66. Applicant also amended claim 1 to add the limitation, “[a power-on reset (POR) sequencer] configured to detect, in the nonvolatile memory, a descriptor used in a boot process and configured to copy the descriptor from the nonvolatile memory to the RAM.” *Id.*, 62. Applicant amended dependent claim 3 to insert the limitation, “wherein the descriptor comprises a POR sequencer descriptor, and[.]” *Id.*, 63. Applicant argued that Chow et al did not disclose the claimed “power-on reset (POR) sequencer” of amended claim 1 and that new claims 21-28 incorporated the limitations of the currently pending claims. *Id.*, 68-69. The Examiner then allowed pending claims 1-9 and 21-28. *Id.*, 76.

49. Applicant next filed a first request for continued examination (“RCE”) with an amendment. *Id.*, 79-83. The Examiner allowed the claims in the next Office Action. *Id.*, 89. Applicant then filed a second RCE without an amendment, and the Examiner allowed the claims in the next office action with an Examiner’s

Amendment and Reasons for Allowance that relied on the “power-on reset (POR) sequencer” limitations. *Id.*, 94-96.

50. Applicant next filed a third RCE with an amendment adding new claims 29-32. *Id.*, 104-107. The Examiner then rejected all claims for non-statutory obviousness-type double patenting over the co-pending application that issued as U.S. Patent No. 9,858,084, and rejected claims 1-4, 6-7, 9, 21-23, and 25-32 under 35 U.S.C. 103 as unpatentable over the combination of US 2007/0180223 A1 to Tanaka and U.S. Patent No. 7,921,237 to Holland et al. *Id.*, 114-118. However, the examiner stated that pending claim 5 (and other claims not relevant here) contained patentable material and would be allowable if presented independently. *Id.*, 118. Applicant, therefore, added this text at the end of pending claims 1 and 29 (which became issued claims 1 and 12):

[1[d]] wherein the descriptor comprises a POR sequencer descriptor, and wherein the POR sequencer uses the POR sequencer descriptor which is a preassembled descriptor that is stored in the nonvolatile memory,

[1[e]] wherein the POR sequencer descriptor includes register information and DMA controller descriptors, and

[1[f]] wherein the POR sequencer reads and uses the register information to update configuration registers of the embedded system.

Id., 122, 127. The first two clauses (1[d] and 1[e]) incorporated language from previously-pending claims 3 and 4, which the examiner had decided were

unpatentable over Tanaka and Holland. The final added clause (1[f]) contained language from previously-pending claim 5, which the examiner had found patentable. The text in the last two clauses was also added to pending claim 21 (which became issued claim 6). *Id.*, 124-125; *compare id.*, 101 (previous claims 3, 4, and 5).

51. Thus, each independent claim was modified to include the limitations about reading and using register information to update configuration registers to overcome rejections by the Examiner. The claims were then allowed, and issued, as amended. *Id.*, 138, 139.

C. Claim Construction

52. I understand that “claim construction” is the process of determining a patent claim’s meaning. I understand that the Board applies the same claim construction standard used in district courts which seeks to give claim terms their plain and ordinary meaning to a POSITA at the time of the alleged invention in view of the claims, specification, and prosecution history. I also understand that if the patent applicant gave a term a special meaning in the specification or during prosecution, then the special meaning should be used. Lastly, I understand that claim construction is a matter of law and that the Board will ultimately determine the final claim construction.

X. CLAIMS 1, 5, 6, 9, AND 12 ARE OBVIOUS OVER THE COMBINATIONS OF POST, WILSON, AND KAO AND POST, WILSON, KAO, AND SO

A. Post

53. Post described a method for booting an embedded device such as an iPod, an iPhone or other cellphone, a computer, or other electronic device. INTEL-1005, 2:60-67. An exemplary device, shown in Figure 1, includes a system on a chip (SOC) 110 and a non-volatile memory (NVM) 120. The SOC includes control circuitry, which may include one or more processors (*id.*, 3:36-38), memory, a boot ROM, and an interface to the NVM. *Id.*, 3:33-34.

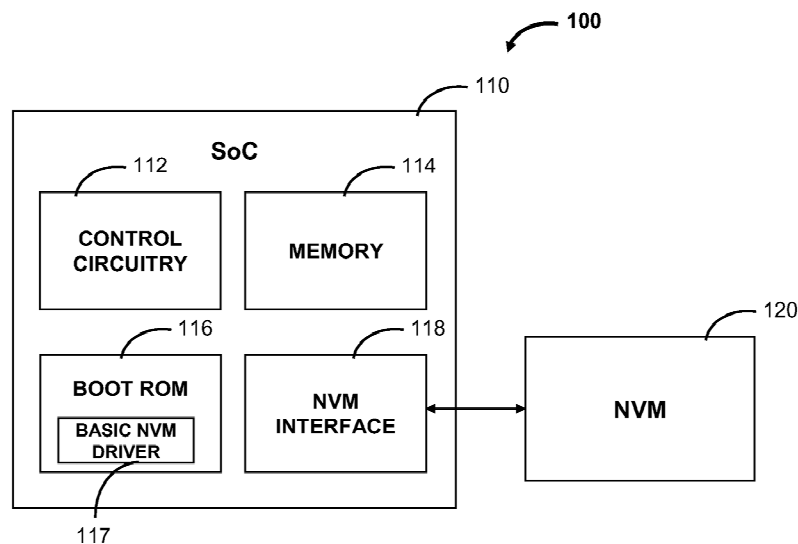


FIG. 1

SoC 110 had control circuitry 112 that could include circuitry, logic, or one or more processors for controlling the operations and functions of SoC 110 and electronic device 100:

System-on-a-chip 110 can include control circuitry 112, memory 114, boot ROM 116, and NVM interface 118. Control circuitry 112 can control the general operations and functions of SoC 110 and electronic device 100 in general. Control circuitry 112 can include any suitable components, circuitry, or logic, such as one or more processors. Control circuitry 112 may operate under the control of a program, such as an application, operating system, a NVM driver, or a bootloader loaded in memory 114. At bootup of electronic device 100, control circuitry 112 may operate under the control of program code stored in boot ROM 116.

Id., 3:33-43. SoC 110 also included memory 114 that was “any suitable type of volatile memory, such as random access memory (“RAM”). (*Id.*, 3:33-34, 3:44-51.)

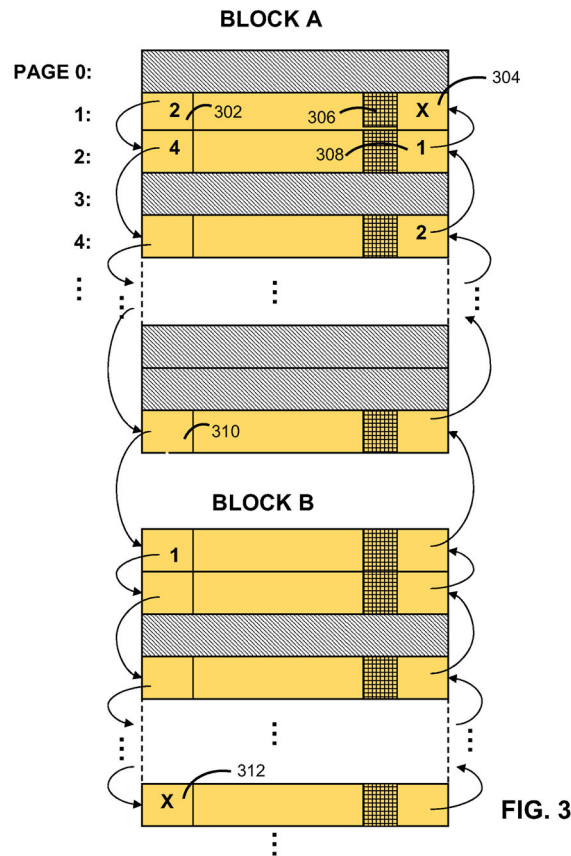
Post explained that memory 114 could take the form of several types of random access memory and that it could serve as the main processor memory:

Memory 114 can include any suitable type of volatile memory, such as random access memory (“RAM”) (e.g., static RAM (“SRAM”), dynamic random access memory (“DRAM”), synchronous dynamic random access memory (“SDRAM”), double-data-rate (“DDR”) RAM), cache memory, or any combination thereof. In some embodiments, memory 114 may act as the main memory for any processors implemented as part of control circuitry 112.

(*Id.*, 3:44-51.)

54. Post describes a process for organizing boot data in the NVM. “[T]he boot data may be stored in only a subset of the memory locations in the boot data storage area ... includ[ing] nonconsecutive pages in one or more blocks of the non-volatile memory.” *Id.*, 1:45-48; *id.*, 1:45-48 (“[T]he boot data may be stored in only

a subset of the memory locations in the boot data storage area. The subset can include nonconsecutive pages in one or more blocks of the non-volatile memory.”). Nonvolatile memory 120 included a “boot data storage area” made up of “any suitable number of integrated circuits, blocks, or pages.” (*Id.*, 4:27-35 (“The boot data storage area may be a logical or physical partition in NVM 120 dedicated to storing boot data.... The boot data storage area can include any suitable number of integrated circuits, blocks, or pages....”).) The use of nonconsecutive pages of NVM is illustrated in Post’s Figure 3, which shows a discontinuous sequence of NVM pages that contain boot data (*i.e.*, “boot data pages” (*id.*, 5:6-8)). Each page (highlighted in yellow) is linked to the next page of boot data by a next page pointer 302 and to a previous page of boot data by a prior page pointer 304. *Id.*, 6:28-33, 6:38-46. In addition to the pointers, a boot page indicator 306 is set in each page that contains boot data. *Id.*, 6:35-37, 7:61-64.) As shown in Figure 3, a “next page pointer can reference another page in a different block of the same or a different integrated circuit.” *Id.*, 6:51-53.



55. “At bootup of electronic device 100, control circuitry 112 may operate under the control of program code stored in boot ROM 116.” *Id.*, 3:41-43; *see also*, *id.*, 3:53-55. Operating pursuant to this code, the control circuitry loads boot data from the NVM 120 into the memory 114:

The program code may be executed to begin loading additional information from NVM 120 into memory 114. This additional information may include firmware, such as a first stage boot loader, that, once loaded into memory 114, is used by electronic device 100 to load a second stage bootloader or the kernel and operating system of electronic device 100. For clarity, ... the firmware (or other data) retrieved from the NVM 120 by the mask ROM code may sometimes be referred to as ‘boot data.’

Id., 3:55-65.

56. A process implemented in the boot ROM code for retrieving boot data from the NVM is illustrated in Post's Figures 4 and 5. *Id.*, 8:15-20 (Fig. 4 illustrates the process of finding first page of boot data), 9:14-18 (Fig. 5 illustrates process for loading the rest of boot data). The first step in the process is to find the beginning of the boot data. At steps 404 and 406 "the electronic device may begin executing mask ROM code [the program code]," and "read an initial page from the non-volatile memory." *Id.*, 8:27-31. "[A]t step 408, the electronic device may determine whether the initial page contains boot data." *Id.*, 8:34-35.

This determination can be performed by employing an error correcting code ... to check for a valid codeword, or by checking ... to determine whether the initial page is programmed or unprogrammed. In addition[,] or alternatively, this determination can involve checking for a boot data indicator or a non-data indicator in the metadata

Id., 8:36-42. If the first page examined does not contain boot code, in step 410 the device reads the next page from the NVM, and returns to step 408 to determine if it contains the first page of boot code. *Id.*, 8:43-49. This continues until the first page is found or the algorithm determines that it cannot be retrieved. *Id.*, 8:53-64, 9:3-7. The control circuitry loaded the first page of boot data into memory. (*Id.*, 9:3-6 ("Returning to step 414, if the electronic device determines that the current page is the first page of boot data, the electronic device can maintain the boot data in

memory at step 420.”), Fig. 4 (step 420, “Maintain the Boot Data in Memory”), Fig. 5 (step 510, same).)

57. Once the device finds a page of boot data, Post’s electronic device can use the process shown in Figure 5 for identifying subsequent pages of boot data. The electronic device then examined the next-page pointer to determine whether there were more pages of boot data. If there were, it saved the next-page pointer “for use in processing subsequent pages” and read the next page into memory:

Following step 510 [“Maintain the Boot Data in Memory”], the electronic device can determine at step 512 whether the current page contains the last page of boot data. For example, the electronic device can examine the next page pointer to determine whether the pointer references a subsequent page of the non-volatile memory, or whether the pointer indicates that there are no more pages of boot data. If the current page is the last page, process 500 may end at step 514. At this point, all of the boot data may be stored in memory, so the electronic device may transfer control of the bootup process from the mask ROM code to the boot data. If the current page is not the last page, process 500 may instead continue to step 516 [“Save the Page’s Next Page Pointer”], where the electronic device can save the current page's next page pointer for use in processing subsequent pages. Process 500 may then return to step 504 [“Read Next Page from NVM”], where the electronic device can continue reading pages from the non-volatile memory to obtain the rest of the boot data.

(*Id.*, 9:47-62; *id.*, Fig. 5.) Control circuit 112 used pointers to identify additional pages of boot data and iterate through the process of Figure 5 until “there [were] no

more pages of boot data,” until all boot data had been loaded into memory 114. (*Id.*, 9:14-15, 9:49-56, Fig. 5.)

58. The device retrieved pages from the NVM in sequence. Post’s algorithm could use several methods to determine if a page of NVM data contained boot data. It could apply an error correcting code (ECC) to the data to determine if it contains valid data, check to determine if the page has been programmed or not, examine the boot data indicator metadata in the page, and examine the prior page pointer to determine if it points to the last previous boot data page identified or not:

Then, at step 506, the electronic device may determine whether the current page contains boot data. In some embodiments, this determination can be performed by employing an error correcting code on the current page data to check for a valid codeword, or by checking a program status of the initial page to determine whether the initial page is programmed or unprogrammed. In addition or alternatively, this determination can involve checking for a boot data indicator or a non-data indicator. If the electronic device determines that the current page contains boot data, the electronic device can verify the prior page pointer of the current page data. That is, at step 508, the electronic device can confirm whether the prior page pointer points back to the expected page (i.e., the most recently read and/or processed boot data page).

(*Id.*, 9:25-38.)

59. If the page was determined to not contain boot data, the control circuitry examined the next page pointer from the prior boot data page to confirm that it does not point to the current page. *Id.*, 9:63-10:7. If a retrieved page was determined not

to contain boot data, it was not kept and was overwritten by the next page retrieved from the NVM. *Id.*, 10:3-8. By only keeping those pages determined to contain boot data, Post’s system constructed a contiguous sequence of boot data pages in the memory 114, even if the boot data was stored in a discontinuous and unpredictable manner in the NVM. This is illustrated in Post’s Figure 2, which shows how a sequence of boot data loaded only in “lower” NVM pages is copied to a continuous sequence of pages in the memory. *Id.*, 5:39-52.

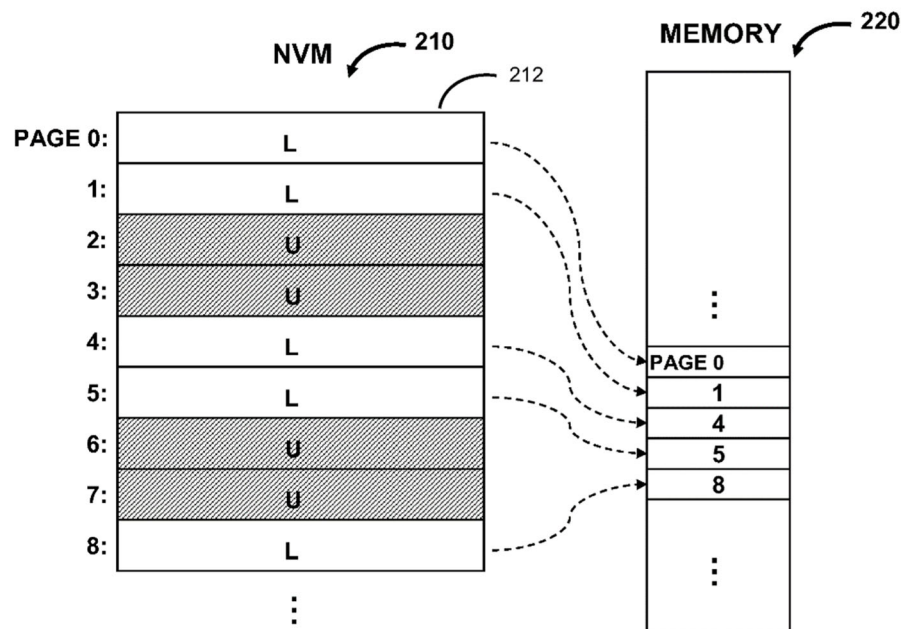


FIG. 2

60. In another boot data retrieval method described by Post, the next page pointers may be used to avoid the need to retrieve and examine each page from the NVM. *Id.*, 1:62-64 (“Using these pointers, the electronic device can load the boot data and verify that pages expected to have boot data actually contain boot data (and

vice versa).”), 10:48-49 (“pages may be selectively read such that some or all of the non-data pages are skipped over”).

61. If the boot data is corrupted and cannot be corrected using the ECC codes (*id.*, 6:60-65), or the next page pointers or back page pointers are not consistent with the addresses in the NVM where boot data pages are found (*id.*, 10:8-13), then Post’s control circuitry may determine that it has failed to retrieve correct boot data. In that case Post’s system discards the downloaded boot data and may seek to find an alternative “mirrored” copy in the NVM. *Id.*, 10:14-19. If none is available, the system may signal a boot failure. *Id.*, 10:24-31. In that case, “the electronic device may stop the bootup process and may display an error message to the user.” *Id.*, 10:30-32; *see also, id.*, 8:65-66 (error message if system cannot find a first page). The user may be able to intervene to restore or update the boot data stored in the NVM using the process shown in Post’s Figure 6. *Id.*, 10:60-11:3.

62. Post’s control circuitry incorporated integrity checking into its boot data transfer process. The prior-page and next-page pointers were used to “verify the integrity of boot data storage.” (INTEL-1005, 6:54-55 (“In some embodiments, the electronic device can use the next page pointers to verify the integrity of boot data storage.”); *id.*, 7:21-33 (“As with the next page pointers, the prior page pointers may allow an electronic device to verify and account for all of the boot data stored in the NVM.... Thus, the electronic device can use just the prior page pointers or the

prior page pointers in conjunction with other metadata (e.g., next page pointers) to verify and account for all of the boot data stored in the NVM.”.) The step of determining whether a page included boot data involved “employing an error correcting code” on the page to detect uncorrectable errors in the boot data overall. (INTEL-1005, 8:34-38 (“Then, at step 408, the electronic device may determine whether the initial page contains boot data. In some embodiments, this determination can be performed by employing an error correcting code on the initial page data to check for a valid codeword,”), 6:60-65 (“If the error correction results in uncorrectable errors, the boot data for page 2 may be unobtainable, and the electronic device may not have all of the boot data necessary for device operation.... In this case, the electronic device may signal a boot failure.”).) In this way, Post’s process provided a resilient boot up sequence that could determine if critical boot data was missing and abort the boot sequence as necessary. (INTEL-1005, 6:64-65; *id.* 8:62-66 (“Because the electronic device is missing boot data, which may be critical for device operation, the electronic device may signal a boot failure at step 416. For example, the electronic device may stop the bootup process and may display an error message to the user.”).)

B. Wilson

63. Like Post (and Kao), Wilson was also concerned with the process of retrieving and using boot data as part of the boot up sequence for an embedded

electronic device, such as a mobile phone. INTEL-1013, 1:55-2:8, 3:35-37. Wilson described a system, illustrated in Figure 1, that included a microcontroller 100 with a processor 101, a memory 103, a register file 102, a register load assist (RLA) engine 104 and an external non volatile memory 111 that stores boot data. *Id.*, 2:53-67, 3:29-31 (“An external non-volatile memory, such as external ROM 111, can also be provided to store boot data.”). Wilson described that boot-up of an electronic device generally requires device initialization data “The boot-up process allows a microcontroller to perform various actions to prepare the microcontroller for ordinary operation, such as load initialization values in its registers, load instructions in its instruction memory, etc.” *Id.*, 1:25-28. In Wilson’s system the device microcontroller includes a register file and “[t]he microcontroller can require that certain specific predefined initial values are stored in one or more of the registers of the register file before the microcontroller assumes ordinary operation.” *Id.*, 2:54-59; *see also, id.*, 4:59-61. Wilson’s microcontroller also included RLA engine 104 that decompressed register initialization values and loaded them into corresponding registers. (*Id.*, 6:4-7 (“The RLA engine is used to decompress the data field into various register initialization values and to load these values into their respective registers.”), 5:47-51 (“The boot data can be sent in a single block. FIG. 4 shows an exemplary block of boot data 400. Block 400 includes header 401 which identifies the block as a boot data block to the serial port interface. The block further includes

data field 402, which includes the compressed boot data.”), Fig. 1, Fig. 4.) A person of ordinary skill in the art would be familiar with this concept of customizing the behavior of a processing device by setting control and configuration registers to specify properties of the system and how the microprocessor was to behave.

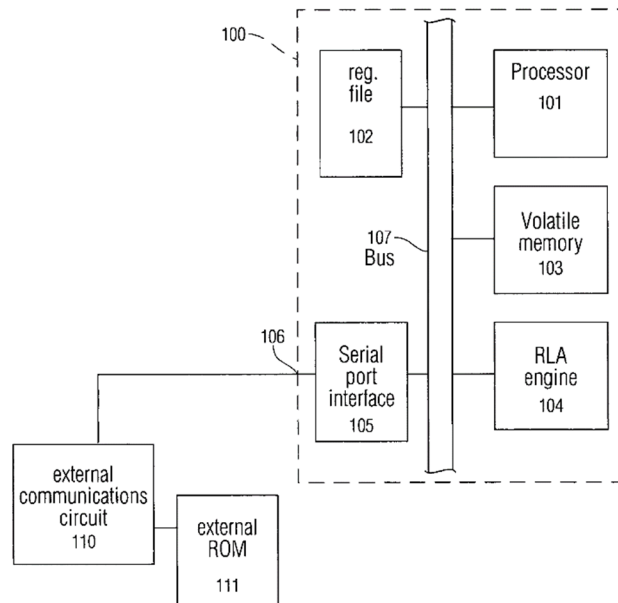


Fig. 1

64. In Wilson’s system, the initialization values for the microcontroller register file are stored in the boot data on the NVM in compressed form. *Id.*, 5:47-6:2. Wilson’s system first transferred the boot data from the NVM to the microcontroller memory. *Id.*, 1:61-67, 5:56-6:2. Wilson described a process for copying initial register values, along with other boot data, from external ROM to volatile memory 103. Compressed boot data stored in nonvolatile memory (such as ROM 111) was sent to the microcontroller in a single block:

FIG. 3 is a flowchart showing a method of operation of embodiments of the present invention. At step 300, a set of compressed boot data is sent to the microcontroller. This can be performed, for example, by the external communication circuit 110. The external communications circuit can compress the boot data itself, receive boot data that has been compressed by another entity, or obtain the compressed boot data from a non-volatile memory (such as ROM 111) where it has been saved in its compressed form.

(*Id.*, 5:31-38; 5:46 (“The boot data can be sent in a single block.”), Fig. 4.) The block included a header identifying the block as a boot data block and a data field with boot data. (*Id.*, 5:46-50, Fig. 4.)

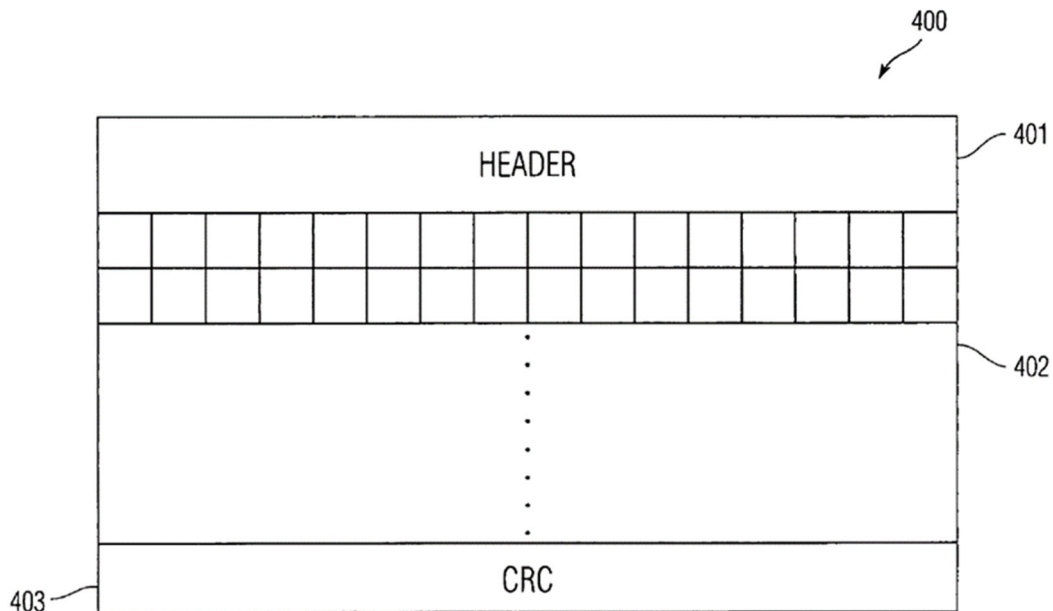


Fig. 4

65. The microcontroller received the data block and “stor[ed] data field 402 of the data block in volatile memory.” (*Id.*, 5:56-61.) The RLA engine, implemented

as a generally programmable processor, then decompressed the register initialization parts of the boot data, and stored these in the register file locations where they belonged:

After the compressed data 500 is saved in a predefined location, the RLA engine can access the predefined location and load the values 501-504 into their respective registers. To perform this, the RLA engine can load a word (such as word 520) out of the predetermined location in memory, and obtain all register values or portions thereof from that word. The RLA engine can then place each obtained five-bit register value in a 16-bit word, filling in the rest of the space with insignificant bits. Thus, the RLA engine forms a full word which is more suitable for transmission over the bus 107. The RLA engine can then consecutively send each formed word to the register file with instructions to load these words into consecutive registers. Register file 102 can receive the words, strip off the insignificant bits, and load the significant five-bit values 501-504 into their respective registers 511-514.

(*Id.*, 7:4-18, Fig. 5; *id.*, 6:3-7, 6:13-20, 6:27-29.) Wilson acknowledged that the boot data in the NVM might include not only the register initialization values, but also other data to be stored in memory and instructions for the processor. *Id.*, 6:42-44 (“[T]he compressed boot data can include other than initial register values. For example, it can include initial memory values, or initial instructions.”). This was consistent with the systems described in INTEL-1005, where the boot data contains code that is to be executed by the control circuitry processor.

66. Wilson taught that predefined initial values had to be stored in the register file before the microcontroller could begin operating.

Register file 102 can include various registers which are used by the processor 101 in the ordinary operation of the microcontroller. The microcontroller can require that certain specific predefined initial values are stored in one or more of the registers of the register file before the microcontroller assumes ordinary operation. Since the register file is usually composed of volatile memory, the initial values can be stored in the register file as part of an initialization or boot operation. Therefore, these initial values can be a part of boot data that is loaded in the microcontroller during a boot-up procedure.

(*Id.*, 2:56-63; *id.*, 4:59-61 (“In order for microcontroller 100 to commence ordinary operation, it can be required that certain predefined values be loaded in the registers of register file 102.”).)

C. Kao

67. Kao described an embedded system 100, illustrated in Figure 1, that included a host 102, a DRAM memory device 110 and a NAND flash memory device 130. INTEL-1004, 3:45-51. Kao’s external nonvolatile memory (“NVM”) 130 stored boot data that included boot codes, controller codes, and OS/Application codes for the embedded system. (*Id.*, 3:46-51, 3:60-4:3 (“[T]he NAND flash interface 122 may be used to access information stored in the NAND flash memory 130 including boot codes 132, controller codes 134, operating system and application code 136, data 138, and any other information stored in the NAND flash

memory 130.”), 1:17-21 (“An embedded computer system typically contains a computer processor (referred to as a host), non-volatile memory (such as a flash memory and/or ROM memory), and volatile memory such as a dynamic random access memory (DRAM).”), Fig. 1.) Like the Post patent, Kao’s patent was concerned with the process of loading boot code for the embedded system from an external NVM:

Embodiments of the invention provide an improved system and method for booting an embedded system. In one embodiment of the invention, boot codes and controller codes stored in a non-volatile memory may be loaded by a DMA engine located on a volatile memory chip into a boot code buffer and a controller code buffer. By utilizing the DMA engine located on the volatile memory chip to load boot and controller codes, the processing workload necessary to boot the embedded system may be offloaded from a host processor to the volatile memory chip, thereby reducing the complexity (and therefore the overall cost) of the host processor and the embedded system.

(*Id.*, 2:65-3:3.) Figure 1 of Kao, annotated to highlight the DMA engine 120, is shown below.

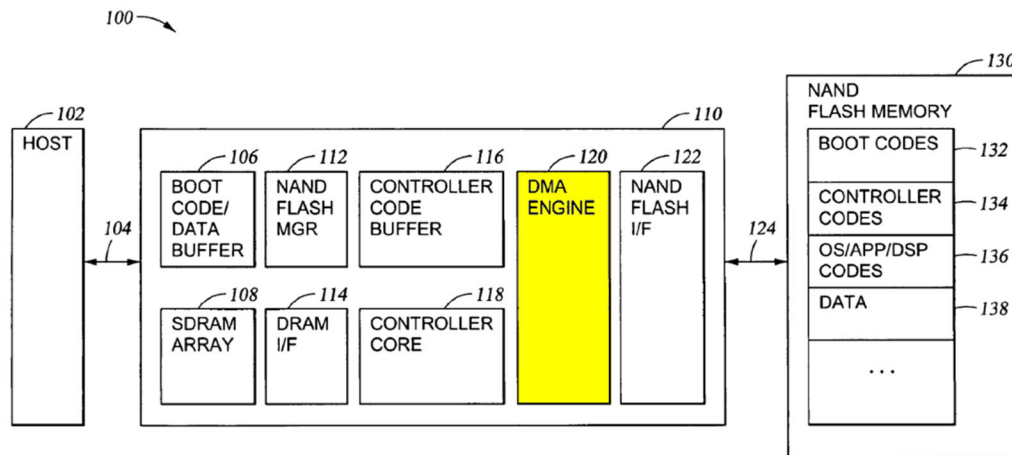


Fig. 1

68. Kao described using the DMA engine instead of the host processor to copy boot data from a nonvolatile NAND flash memory to a boot code buffer. (*Id.*, 8:26-29 (“In one embodiment of the invention, the DMA engine registers 502₀, 502₁, 502₂, 502₃ may be used to control which data is loaded from the NAND flash memory 130 into the SDRAM 108 and vice versa.”), 8:39-45 (“[W]hen a DMA transfer is initiated, the address generator and counters 504 may initialize a counter and provide source and destination addresses to the NAND flash memory 130 (via the NAND flash interface 122) and to the SDRAM array 108 (via the SDRAM interface 114, or other interfaces, e.g., for the boot code buffer 212)”)).) When the system was powered up, DMA engine circuitry loaded boot codes (and other controller or OS codes) from NAND flash memory 130 into boot code buffer 106. (*Id.*, 4:61-66 (“When the system 100 is powered up at step 310, the boot sequence

may begin at step 312 by loading the boot codes 132 ... from the NAND flash memory 130 into the boot code buffer 212 using the DMA engine circuitry 120 of the DRAM memory device 110.”.) Kao taught that this DMA transfer used techniques “known to those skilled in the art” that resulted in “automatic transfer of the boot code ... without external control or input, e.g., by the host 102 or any external controller circuitry.” (*Id.*, 4:66-5:3.) After the boot codes were copied from the NAND flash memory, the DMA circuitry copied desired code into SDRAM 108. (*Id.*, 4:24-30 (“After the boot codes 132 and controller codes 134 have been loaded into the DRAM memory device 110, commands issued to the DMA engine 120 may be used to initiate a DMA access which copies desired code from the OS ... into a code shadow 204 and other data 202 stored in the SDRAM array 108.”).)

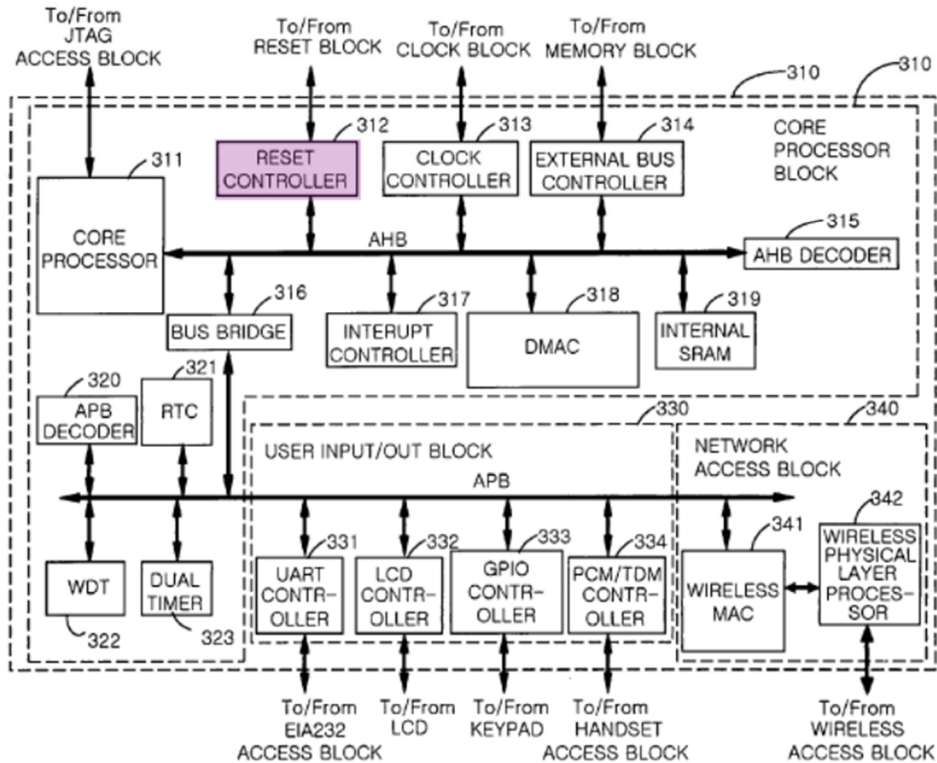
69. An important teaching in Kao’s patent was that a DMA engine should be used to transfer boot code from the NVM to a DRAM memory 110 that is accessible to the host. *Id.*, 4:20-24 (“[T]he DMA engine circuitry 120 may be used to automatically load boot codes 132 from the boot code images 232 into the boot code buffer 212 and load controller codes 134 from the controller code images 234.”); 11:17-19 (“each DMA transfer may occur between the NAND flash memory 130 and the SDRAM array 108 without storing code in separate, special purpose buffers”). As Kao described, using a DMA engine to carry out boot code transfers allows the boot code to be loaded “without interaction by the host.” *Id.*, 3:18-21.

70. Utilizing the DMA engine to load boot code from the NVM offloads the processing workload necessary to boot the embedded system from the processor and “reduc[es] the complexity (and therefore the overall cost) of the host processor and the embedded system.” *Id.*, 3:3-10. Kao described that use of a DMA engine to carry out “automatic transfer of the boot code 132 from the NAND flash memory 130 ... without external control or input, e.g., by the host 102 or any external controller circuitry” was “known to those skilled in the art.” *Id.*, 4:66-5:4.

D. So

71. So disclosed a reset controller within an embedded system. So described “a system-on-chip for an Internet phone,” (So, Abstract, 1:7-8). So’s SoC “includes a core processor block 310 performing a function of a central processing unit” (*id.*, 3:56-57) that was illustrated in Figure 3. (*Id.*, 3:53-55 (“FIG.3 is a detailed block diagram of the system-on-chip of FIG. 1 in accordance with an embodiment of the present invention.”), Fig. 3 (shown below, annotated).)

FIG. 3



72. So's SoC included a "core processor block 310" that included a core processor 311 connected to an Advanced High-Performance Bus ("AHB"). So's SoC processor block included reset controller 312 (in purple, above). (*Id.*, 3:62-63.) "The reset controller 312 connected to the external reset block initializes all controllers according to the reset signal inputted from the external reset block, generates the reset signal required in each controller and provides the signal to the [Advanced High-Performance Bus]." (*Id.*, 4:4-8.)

E. Overview of the Combinations

73. As I describe in detail below, the combinations of Post, Kao, and Wilson and Post, Kao, Wilson, and So disclosed all limitations of the challenged claims. In the combination of Post, Kao, and Wilson, Post's control circuitry 112 would be modified to include 1) Wilson's register file and circuitry to perform the register loading functionality of Wilson's RLA engine and 2) a DMA controller that performed the boot data transfer functionality of Kao's DMA engine; and the boot data would have included initial register values, as taught by Wilson. The combination of Post, Kao, Wilson, and So would have had the additional adjustment of Post's SoC 110 being modified to include So's reset controller.

F. Independent Claim 1

74. It is my opinion that claim 1 of the '694 Patent is rendered obvious by the combination of Post, Wilson, and Kao; and separately by the combination of Post, Wilson, Kao, and So.

1. Claim 1 Preamble

An apparatus, comprising:

75. Post disclosed an "electronic device 100" which was an "apparatus." Post disclosed that electronic device 100 could be any of a number of apparatuses, including a cellular telephone or computer:

FIG. 1 is a block diagram of illustrative electronic device 100. In some embodiments, electronic device 100 can be

or can include a portable media player (e.g., an iPodTM made available by Apple Inc. of Cupertino, Calif.), a cellular telephone (e.g., an iPhoneTM made available by Apple Inc.), a pocket-sized personal computer, a personal digital assistance (“PDA”), a desktop computer, a laptop computer, and any other suitable type of electronic device or system.”

INTEL-1005, 2:60-67. Electronic device 100 included a system-on-chip 110 (*id.*, 3:1-2) that included control circuitry 112 to control SoC 110 and electronic device 100. Control circuitry 112 could include circuitry or one or more processors. (INTEL-1005, 3:33-38.)

2. Claim 1, Limitation a

an embedded system comprising one or more processors, a reset controller, a storage device controller, one or more direct memory access (DMA) controllers, a random access memory (RAM) and a memory controller, a nonvolatile memory and a nonvolatile memory controller, a debug interface, and;

76. **Embedded system.** Post disclosed that the “[e]lectronic device 100 can include system-on-a-chip (‘SoC’) 110 and non-volatile memory (‘NVM’) 120.” INTEL-1005, 3:1-2; *id.*, Fig. 1:

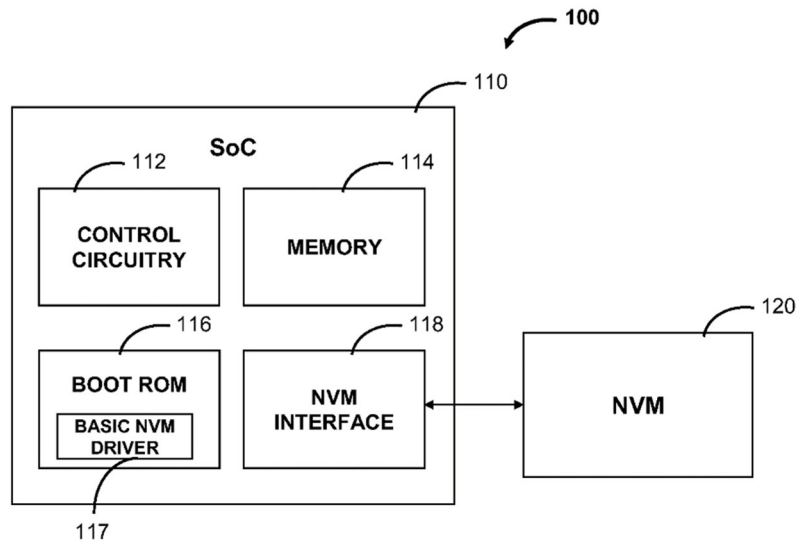


FIG. 1

The SoC 110 and NVM 120, which made up the electronic device 100, were collectively an *embedded system*.

77. Post’s system-on-chip (“SoC”) 110 and nonvolatile memory 120 were together an *embedded system*. SoC included control circuitry 112, which included one or more processors, and nonvolatile memory:

System-on-a-chip 110 can include control circuitry 112, memory 114, boot ROM 116, and NVM interface 118. Control circuitry 112 can control the general operations and functions of SoC 110 and electronic device 100 in general. Control circuitry 112 can include any suitable components, circuitry, or logic, such as one or more processors.

INTEL-1005, 3:33-38. The ’694 patent gives an example of an “embedded system” as including a system with one or more processors: “The term embedded system will

be used to refer to a single CPU or a multiprocessor system in this paper.” ’694 Patent, 1:26-28. Post disclosed an embedded system in a similar way.

78. Post’s examples of the types of devices represented by the electronic device 100 further illustrated that the device could be an *embedded system*. Post described that, for example, the “electronic device 100 can be or can include a portable media player (e.g., an iPod™ made available by Apple Inc. of Cupertino, Calif.), a cellular telephone (e.g., an iPhone™ made available by Apple Inc.), a pocket-sized personal computer, [or] a personal digital assistance (‘PDA’).” INTEL-1005, 2:61-66. Components used in devices such as portable media players, cellular telephones, and PDAs, such as SoCs, are examples of embedded systems. Thus the SoC 110 and NVM 120 disclosed by Post are further demonstrated to be *embedded systems*. Therefore Post disclosed this element of claim 1[a].

79. The rest of claim 1[a] recites conventional components of embedded systems. Each of these was either included in Post’s embedded system would have been obvious to include for the reasons discussed.

80. One or more processors. SoC 110 included one or more processors within control circuitry 112. “System-on-a-chip 110 can include control circuitry 112 ... Control circuitry 112 can include any suitable components, circuitry, or logic, such as one or more processors.” INTEL-1005, 3:33-38. Post even referred to

control circuitry 112 interchangeably as the “processor 112.” *Id.*, 4:5-6. Post disclosed this element.

81. **A reset controller.** When a system is first powered on, components of the system are initially held in a reset state. *See* Kao, 1:32-34 (“when the embedded system is powered up (e.g., when the embedded system enters a reset state)”). The reset state ensures that the system supply voltage reaches an acceptable level for normal operation and may also be called a Power-On Reset. INTEL-1018, 2 (“Power-On Reset. When power is applied to a microprocessor, the internal registers ‘come up’ in arbitrary states that contain random data. ... To ensure proper startup, the RESET input must be held low ... while the supply voltage (V_{CC}) is below the minimum level allowed for microprocessor operation.”). For example, the ’694 patent describes a “process of releasing ... CPUs 201 from their reset state” where each CPU is released from reset before executing system firmware. INTEL-1001, 8:24-39, 3:61-67. The ’694 patent further states that “[t]he method of copying POR Sequencer Descriptor from nonvolatile memory to RAM starts when the supply voltage of the system becomes stable after power-up. Then the Reset Controller, the POR Sequencer, the CPU Bus, the memory controllers, the DMA controllers, and the storage device controllers are released from reset.” (INTEL-1001, 3:12-17.)

82. Post disclosed that control circuitry 112 received a “startup or reboot command” that caused it to start the bootup of the electronic device: “Turning first

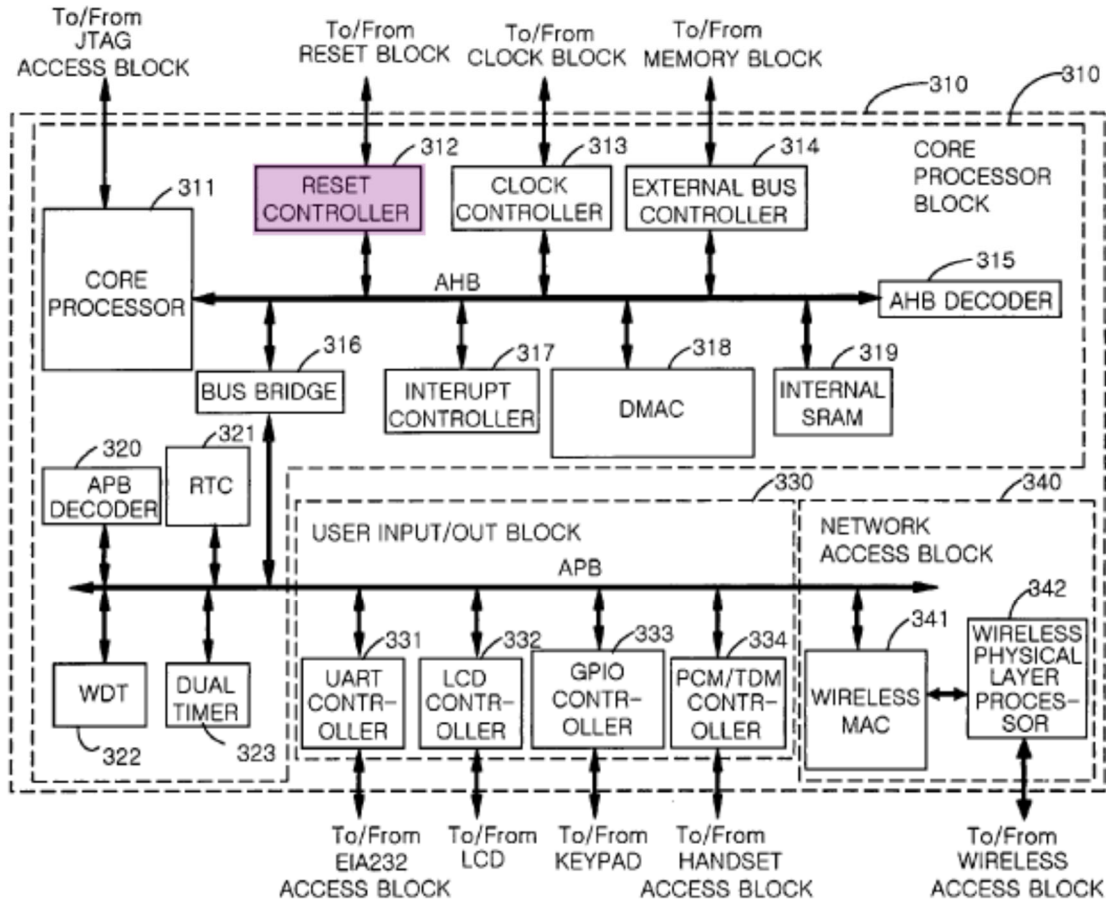
to FIG. 4, a flowchart of illustrative process 400 is shown for initiating bootup of an electronic device. ... Process 400 may begin at step 402 in response to, for example, a startup or reboot command of the electronic device.” INTEL-1005, 8:15-27; *id.*, 8:9-12 (control circuitry 112 performed Process 400 of Fig. 4). Post taught that this bootup occurred when the system was powered up. “When an electronic device is being powered up or rebooted, an operating system can be booted up from a NVM of the electronic device.” *Id.*, 1:16-18.

83. A POSITA would have understood the “startup or reboot command” received after power-up, which caused control circuitry 112 to boot the device and operating system, to indicate that the SoC included a reset controller that generated these commands. A system boot-up indicates that a power-on or reset signal has been sent to components of the system. *E.g.*, INTEL-1016, 555 (“System boot-up means that some type of power-on or reset source ... has occurred.”). A POSITA would have understood power-on reset to be one of “[t]he most common microprocess-supervisory functions.” INTEL-1018, 1. A POSITA would expect the reset signal to be generated by some component within the SoC. Otherwise, the processor and other system devices would power up in an undefined state that could render them inoperable. INTEL-1018, 2 (“Power-On Reset. When power is applied to a microprocessor, the internal registers ‘come up’ in arbitrary states that contain random data. ... To ensure proper startup, the RESET input must be held low ...

while the supply voltage (V_{CC}) is below the minimum level allowed for microprocessor operation.”). A POSITA would have been familiar with many reset controllers that were appropriate and intended for use within a SoC like that of Post. As just one example, an ARM Technical Reference Manual from 1999 described a Reset Controller for use in an ARM processor that “generate[d] the HRESETn signal ... [that] is used by all the other elements in the EASY microcontroller, primarily for power-on initialization.” INTEL-1017 at 2-3; *id.* at 4-40 (“The reset controller is used to generate the system reset signal from an external reset input as shown in Figure 4-23.”). Apple’s iPhones, as disclosed by INTEL-1005, used ARM processors since the first models.

84. Additionally, So expressly disclosed a reset controller in an SoC that was similar to Post’s SoC. So described “a system-on-chip for an Internet phone” (INTEL-1021, Abstract), similar to Post’s SoC for an iPhone. So’s SoC “includes a core processor block 310 performing a function of a central processing unit” (*id.*, 3:56-57) that was illustrated in Figure 3. *Id.*, 3:53-55 (“FIG.3 is a detailed block diagram of the system-on-chip of FIG. 1 in accordance with an embodiment of the present invention.”), Fig. 3:

FIG. 3



85. So's SoC processor block included reset controller 312 (in purple, above). "The reset controller 312 connected to the external reset block initializes all controllers according to the reset signal inputted from the external reset block, generates the reset signal required in each controller and provides the signal to the [Advanced High-Performance Bus]." *Id.*, 4:4-8. So, therefore, disclosed a reset controller.

86. It would have been obvious to modify Post in view of So's teachings. In the resulting combination, Post's SoC 110 would have included So's reset

controller 312. This would have been obvious because the systems in So and Post are similar: So described a system on chip (SoC) to go into a device like a phone, and Post similarly described SoC 110 as going into an iPhone. INTEL-1021, 1:7-8 (“a system-on-chip for the Internet phone”), Title (“Apparatus for Developing and Verifying System-On-Chip for Internet Phone”); INTEL-1005, 2:60-64 (“electronic device 100 can be or can include ... a cellular telephone (e.g., an iPhone™ made available by Apple Inc.)”), 3:33-41, Fig. 1.) So also provided a way to implement the reset functionality that Post described, but did not provide details for: Post taught that control circuitry 112 received a “startup or reboot command” that initiated the bootup process of Fig. 4 (INTEL-1005, 8:25-27, 8:9-12), which a POSITA would have understood to be substantially the same as a reset command. Post did not describe the component that generated these commands; but So specifically taught a reset controller included in the SoC that “generates the reset signal required in each controller” of the SoC according to an external reset signal (INTEL-1021, 4:4-8). A POSITA would have looked to So for teachings on how to implement a reset controller that provided Post’s startup or reboot commands that prompted the boot process.

87. A POSITA would also have viewed the combination of Post with So’s teachings as obvious because So taught a known technique to solve a known problem—specifically, a reset controller that generated reset signals used in the

process of initializing components of an embedded system. Including a reset controller, like that disclosed by So, within Post's SoC 110 was within the skill of an ordinary artisan, because reset controllers were conventional components that a POSITA would be generally familiar with. For example, the Noergaard text referenced an off-the-shelf microprocessor (the Motorola/Freescale MPC823) that included a "reset controller that is responsible for responding to all reset sources." INTEL-1016, 556. The Catsoulis text similarly referenced a reset circuit (a reset controller) in another off-the-shelf microprocessor (the Motorola 68000-series). INTEL-1008 at 340 & Figs. 18-5, 18-6 ("To reset an MC6800," RESET and HALT signal lines had to be driven low by a "reset circuit"). So also evidenced that a POSITA knew how to implement a reset controller for an SoC that was similar to Post (structured for inclusion in a cellular phone, with multiple processors/controllers).

88. **A storage device controller.** Post's SoC 110 included nonvolatile memory interface 118, which included "any suitable circuitry or components that enable[d] SoC 110 to access NVM 120 and its memory locations." INTEL-1005, 4:14-16. The "memory locations" of NVM 120 included "multiple integrated circuits": "In some embodiments, NVM 120 can include multiple integrated circuits, where each integrated circuit may have multiple blocks." Id., 3:12-13. NVM 120's memory locations could also be "configured as single level cells ('SLC') or multi-

level cells ('MLC')." *Id.*, 3:20-21. Post's disclosure that NVM 118 had suitable circuitry to "enable ... access" to NVM 120 and its diversity of memory locations indicated that NVM 118 included a storage device controller that provided an interface and access to these devices. The '694 describes a storage device controller in this same way. '694 Patent, 2:40-41 (referencing "a storage device controller which provides interface to the storage device").)

89. **One or more direct memory access (DMA) controllers.** It would have been obvious to modify Post's SoC 110 to include a DMA controller such as Kao's DMA engine 120, which performed the function of transferring boot data pages from NVM 120 to memory 114. Kao disclosed an "Exemplary Embedded System" that included a "DMA engine 120" within DRAM device 110:

FIG. 1 is a block diagram depicting an exemplary system 100 according to one embodiment of the invention. As depicted, the system 100 may include a host 102, volatile memory (DRAM device 110), and non-volatile memory (NAND flash memory 130). ...

In one embodiment of the invention, the DRAM device 110 may include an SDRAM memory array 108. The DRAM device may also contain a boot code and data buffer 106, a DRAM interface 114, a flash manager 112, a controller code buffer 116, an embedded microcontroller core 118, and DMA engine 120.

INTEL-1004, 3:46-65; *id.*, Fig. 1 (DMA engine highlighted in yellow):

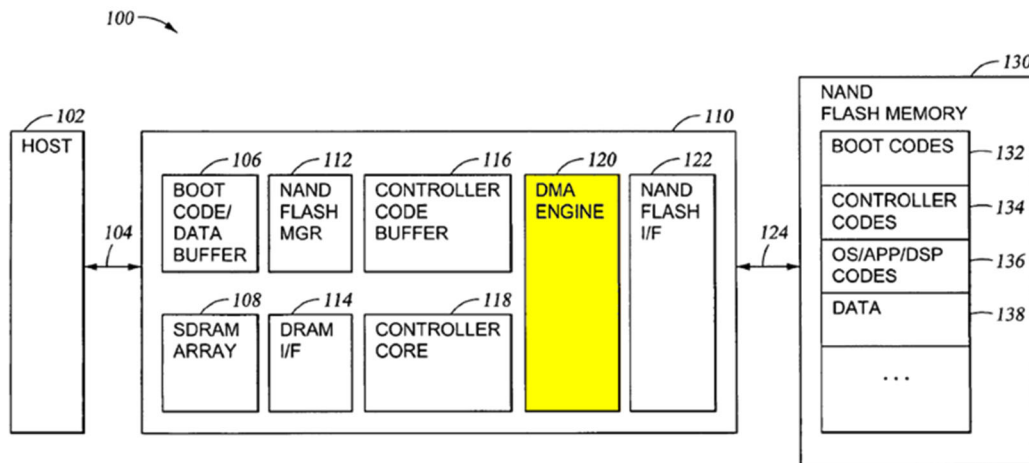


Fig. 1

90. Kao disclosed that DMA engine 120 automatically loaded boot data from NVM (NAND flash memory 130) into volatile memory (buffers in DRAM device 110): “the DMA engine circuitry 120 may be used to automatically load boot codes 132 from the boot code images 232 into the boot code buffer 212 and load controller codes 134 from the controller code images 234.” *Id.*, 4:20-24. DMA engine 120 performed the boot data transfer function without input or control by the host processor, and was thus a “*DMA controller*”: “As known to those skilled in the art, the DMA access may include automatic transfer of the boot code 132 from the NAND flash memory 130 to the boot code buffer 212 without external control or input, e.g., by the host 102 or other external controller circuitry.” *Id.*, 4:66-5:3; *id.*, 5:5-7 (“The DMA engine circuitry 120 may utilize the NAND flash interface 122 ... to access the NAND flash memory 130.”). DMA engine 120 used the source address

of the boot code, *i.e.*, its address within the NVM, along with at least size information, to perform the transfer: “the source address and size of the boot code may be provided by an external device, by data stored in an area of memory (e.g., in a ROM or in a predefined area of the NAND flash memory 130) or in another manner, as described below.” *Id.*, 5:12-16; *id.*, 10:28-47 (“the controller codes 134 may be loaded using data stored in the boot codes 132 ... The resulting address may point to the beginning of the controller code image 134.”).

91. It would have been obvious to combine Post with Kao’s teachings of using a DMA controller and DMA techniques to transfer boot data from nonvolatile memory to RAM. The combination would result in Post’s control circuitry 112 being modified to include a DMA controller (such as Kao’s DMA engine 120) that performed the functions of transferring boot data from NVM 120 to memory 114 described by Post. For example, a POSITA would have found it obvious for the resulting combination to perform the process of Figure 5 of INTEL-1005, transferring boot pages subsequent to a first identified page from NVM 120 to memory 114. Kao motivated this combination because it taught that using DMA to perform memory transfers provides the benefit of offloading work from the processor during a boot process, which reduced the complexity and cost of the overall embedded system:

By utilizing the DMA engine located on the volatile memory chip to load boot and controller codes, the processing workload necessary to boot the embedded system may be offloaded from a host processor to the volatile memory chip, thereby reducing the complexity (and therefore the overall cost) of the host processor and the embedded system.

Id., 3:3-9; *id.*, 4:31-34 (“by offloading work from the host 102 to the DMA engine 120 and embedded microcontroller 118 in the DRAM memory device 110, design and operation of the host 102 and embedded system 100 may be simplified”). Offloading the process of Figure 5 would have freed Post’s processor 112 up to perform other work associated with the boot process while the DMA controller was fetching additional boot data pages, speeding up the process by allowing work to be done in parallel.

92. A POSITA would also have been aware of substantial teachings in the art about the benefits of using DMA techniques (and DMA controllers that performed DMA techniques) generally. The Ganssle text noted that a processor would not perform optimally if it was being “constantly bother[ed] to perform transfers” between different memory locations and that a DMA controller, operating separately from the core processor, could solve that very problem: “you can only achieve optimum performance in your application if data can move around without constantly bothering the core to perform the transfers This is where a direct memory access (DMA) controller comes into play.” INTEL-1009, 214; *id.* (“[I]t is

much more efficient to employ a DMA controller that operates independently from the core.”). Reducing processor workload was a universal concern and objective within the field of embedded system design. That recognized, universal need, combined with the teachings of Kao and others on the benefits of using DMA controllers and techniques to meet that need, would have prompted a POSITA to include a DMA controller as part of control circuitry 112. A POSITA would also have been generally familiar with DMA controllers, and would have known of example implementations available in the art (*e.g.*, Kao’s DMA engine 120), and thus would have had a reasonable expectation of success in making this combination.

93. A POSITA would also have recognized that Post and Kao related to embedded systems of similar design and operation, which would have further motivated a POSITA to include Kao’s DMA controller and techniques in Post. Both Post and Kao described embedded systems with one or more processors, NVM, and volatile memory. INTEL-1005, Fig. 1; INTEL-1004, Fig. 1. Both were designed to be used in mobile phones. INTEL-1005, 1:60-64 (iPhone); INTEL-1004, 1:15-17 (cell phone). Both also described a boot process in which boot data was loaded from NVM to on-chip volatile memory and executed. INTEL-1005, 3:55-4:13; INTEL-1004, 4:20-24 (“As described below with respect to the boot sequence, the DMA engine circuitry 120 may be used to automatically load boot codes 132 from the boot code images 232 into the boot code buffer 212 and load controller codes 134 from

the controller code images 234.”). Post stated that control circuitry 112 “can include any suitable components, circuitry, or logic” (INTEL-1005, 3:36-38) and Kao affirmed that its DMA techniques were “typical” for embedded systems. INTEL-1004, 1:17-24 (“An embedded computer system typically contains a computer processor ... [that] include[s] a central processing unit (CPU) ... or direct memory access (DMA) data transmission device”). In view of all these structural and operational similarities, a POSITA would have looked to the teachings of Kao to improve Post. And a POSITA would have recognized that including Kao’s DMA controller and techniques in Post’s control circuitry 112 would have improved Post’s SoC in a manner similar to how it improved Kao’s system: by offloading processor workload and lowering processor complexity and cost.

94. Further, both Post and Kao addressed a similar problem, of how to load boot data from nonvolatile memory to volatile memory of an embedded system during boot. Kao’s DMA controller and DMA techniques supplied an established way to address that problem, and a POSITA would have been motivated to apply that solution in the context of Post. A POSITA would also have viewed this combination as just substituting Kao’s DMA controller and DMA techniques in place of using processor 112 to perform the entirety of the work of loading boot data and metadata from NVM 112 to memory 114. A POSITA would have recognized that Post’s embedded system was ready for the improvements of reduced processor

workload and system cost/complexity by, for example, using a DMA controller instead of processor 112 to perform the process of Figure 5. It would therefore have been obvious to include a DMA controller as part of the circuitry of control circuitry 112.

95. **A random access memory (RAM) and a memory controller.** Post disclosed that SoC 110 included memory 114. INTEL-1005, 3:33-34 & Fig. 2. Memory 114 was “a random access memory (RAM)”: “Memory 114 can include any suitable type of volatile memory, such as random access memory (RAM).” *Id.*, 3:44-45. Post also disclosed that memory 114 could act as “the main memory for any processors implemented as part of control circuitry 112” (*id.*, 3:50-51) and that control circuitry 112 was able to write data to memory 114—for example, control circuitry 112 “load[ed] additional information into memory 114” (*id.*, 3:55-57) and loaded an “operating system ... and additional NVM drivers ... in memory 114 (i.e., from NVM 120)” (*id.*, 4:5-8).

96. A POSITA would have understood, in view of Post’s disclosure that processors within control circuitry 112 used memory 114 as their “main memory” and that control circuitry wrote information to memory 114, that SoC 110 included a memory controller that allowed this interoperability. At the very least, Post’s disclosure suggested the need and benefit of including a memory controller in SoC 110 to enable control circuitry 112 to load information to memory 114 and use it as

“main memory.” A POSITA would have been familiar with this type of basic controller functionality and would have viewed it as a standard and expected component of an SoC that, like INTEL-1005, was designed for inclusion in an electronic device like an iPhone. INTEL-1009-110, 110 (describing memory controller as a “common type[] of memory manager ... integrated into the master processor ... A memory controller (MEMC) is used to implement and provide glueless interfaces to the different types of memory in the system, such as cache, SRAM, and DRAM, synchronizing access to memory and verifying the integrity of the data being transferred.”). Post thus disclosed a random access memory (RAM) (memory 114) and further disclosed a memory controller (circuitry that allowed control circuitry 112 to interface with memory 114); and it would separately have been obvious to incorporate a memory controller for memory 114 into SoC 110.

97. **A nonvolatile memory and a nonvolatile memory controller.** Post’s “[e]lectronic device 100 can include ... non-volatile memory (“NVM) 120.” INTEL-1005, 3:1-2 & Fig. 1. Nonvolatile memory 120 included a “memory controller” that could be a “flash controller”: “some of the components and functions of NVM interface 118, as well as some of the functionality of any NVM drivers, may be implemented and performed by a separate memory controller (e.g., flash controller) included, for example, in NVM 120.” *Id.*, 4:19-23. Post recognized that flash memory was a type of nonvolatile memory. *Id.*, 1:6-8 (“NAND flash memory,

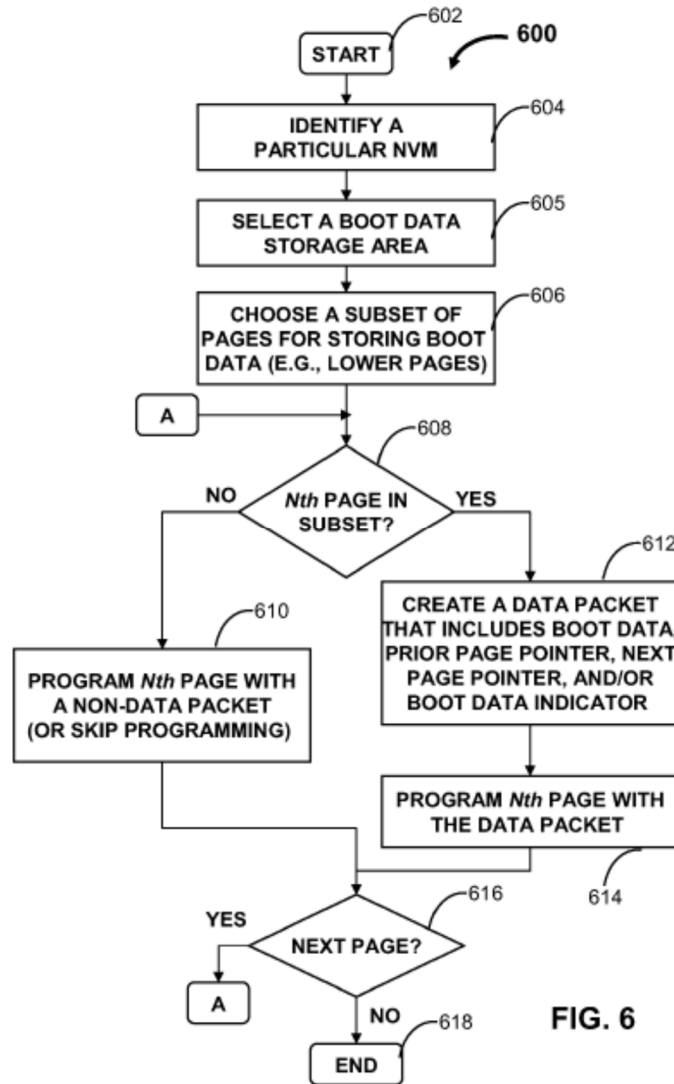
as well as other types of non-volatile memories (NVMs), are commonly used in electronic device for mass storage.”); INTEL-1009, 207 (“The two main types of nonvolatile Flash memory widely used today are NOR-based and NAND-based”).) Post thus disclosed a nonvolatile memory (NVM 120) and a nonvolatile memory controller (flash controller in NVM 120).

98. **A debug interface.** A debug interface facilitates debugging of embedded systems by, among other things, rewriting or updating firmware. Post disclosed that control circuitry 112 performed a “firmware update as part of a restore process” that allowed boot data to be “changed or updated”:

Referring now to FIG. 6, a flowchart of illustrative process 600 is shown for storing boot data in a reprogrammable non-volatile memory. Because, unlike a mask ROM, the nonvolatile memory is reprogrammable by an electronic device’s operating system and NVM interface, the boot data may be changed or updated without the inconvenience of switching boot ROMs. Process 600 may be performed during a firmware update as part of a restore process. The restore process may be performed at a factory during manufacture or repair of the electronic device, or may be initiated by the end user to update the electronic device.

INTEL-1005, 10:60-11:3.

99. Figure 6 illustrated this “process for storing boot data in a non-volatile memory” (*id.*, 2:54-56):



100. Process 600 (which was performed by control circuitry 112, *id.*, 8:9-12), began with control circuitry 112 identifying one out of multiple NVMs in which to store boot data:

Process 600 may begin at step 602, and at step 604, a particular non-volatile memory may be identified. The NVM may be identified responsive to a user selection of a NVM installed in the electronic device, or the electronic device may automatically detect the presence of an installed non-volatile memory. The particular non-volatile

memory may be identified from a variety of potential NVMs.

Id., 11:4-10.

101. Control circuitry 112 identified “a boot data storage area in the non-volatile memory” and optionally “eras[ed] outdated boot data or perform[ed] other garbage collection procedures to clear space for new boot data.” *Id.*, 11:21-30. Control circuitry 112 then stored boot data, pointer data, and non-data packets into the nonvolatile memory:

The electronic device may then store (or not store) boot data in each page of the boot data storage area by iterating through the following steps. At step 608, the electronic device may determine whether the current page in the iteration is in the subset of pages. If not, the electronic device may not program boot data into the current page. The electronic device may instead program a non-data packet in the current page or may leave the current page unprogrammed. If the current page is in the subset, the electronic device may prepare some of the boot data for programming into the page. In particular, at Step 612, the electronic device may create a data packet that includes boot data. The data packet may also include metadata, such as a prior page pointer, a next page pointer, a boot data indicator, or a combination thereof. Then, at step 614, the electronic device may program the current page with the data packet.

Id., 11:41-56. Control circuitry 112 iterated through this process until there was no “additional boot data to be stored.” *Id.*, 11:57-62. At that point, “all of the boot data may be stored in the non-volatile memory.” *Id.*, 11:62-63.

102. Post's disclosure of control circuitry being able to update the stored boot data as part of a firmware update or restore process, performed by manufacturer or user, indicated that the SoC 110 had a debug interface. The '694 patent describes a "debug interface" in this same way, as including, for example, a component "used to preprogram the nonvolatile memory and storage device with data and code." INTEL-1001, 2:43-45. At a minimum, a POSITA would have viewed Post as suggesting the inclusion of a debug interface in Post's SoC 110 to perform these functions, because a debug interface (such as, *e.g.*, a JTAG port) was a standard component of embedded systems and would have provided a way to implement Post's disclosed function of updating stored boot data. INTEL-1008, 157-158 ("A JTAG (Joint Test Action Group) port provides access to the internals of the processor and, through it, the rest of the computer system. ... The JTAG port allows for real-time debugging of hardware and software. ... The JTAG port allows you to examine and modify registers and memory locations."), 159 ("JTAG is becoming increasingly common due to the many advantages it brings to the system designer. Most new processor releases are including JTAG in their design"); INTEL-1009, 138 (common system I/O components include "Debugging I/O (BDM, JTAG, serial port, parallel port, etc.)"). Post's SoC 110 therefore included a "debug interface" and alternatively, a POSITA would also have been motivated to include one in SoC 110.

3. Claim 1, Limitation b

a power-on reset (POR) sequencer configured to detect, in the nonvolatile memory, a descriptor used in a boot process and configured to copy the descriptor from the nonvolatile memory to the RAM;

a. a power-on reset (POR) sequencer

103. The asserted combinations disclosed *a power-on reset (POR) sequencer*. Post's control circuitry 112, modified to include Wilson's register file and circuitry to perform the register loading functionality of the RLA engine, was a "*POR sequencer*." Control circuitry 112 (as modified) detected boot data and associated pointer metadata, which was a "*descriptor used in a boot process*," in NVM 120 (*in the nonvolatile memory*"). In the combination of Post and INTEL-1013, a POSITA would have understood that the boot data would have included initial register values, as taught by Wilson.

104. I am not aware of any specific definition provided in the '694 Patent, or any understanding in the art, for the term "power-on reset (POR) sequencer." This appears to be a term coined by the applicant that a POSITA would identify as a component by way of the claimed functionality it is configured to perform:

- detecting a descriptor (having certain characteristics, as recited in Claims 1[d]-[e]) in nonvolatile memory (*see* Sec. X.F.3.b);
- copying that descriptor to RAM; facilitating a boot process (*see* Sec. X.F.3.c);

- verifying integrity of information in RAM (*see* Sec. X.F.4); and
- updating register information (*see* Sec. X.F.7).

See claims 1[b], 1[c], 1[f]. Post’s control circuitry 112, as modified to include Wilson’s register file and circuitry to perform register loading functionality of Wilson’s RLA engine, included this recited functionality.

105. To the extent there is any independent significance to the “POR sequencer” label, a POSITA would have viewed this as indicating a component that facilitates a boot process after power-on. When a system is powered on (including when it is restarted/reset), the different components of the system are initially held in reset to allow the supply voltage to stabilize. The term “Power-On Reset” generally refers to this scenario. The Lenk text describes this scenario:

1.1 Power-On Reset When power is applied to a microprocessor, the internal registers “come up” in arbitrary states that contain random data. Applying a reset command to the microprocessor overcomes this chaos by resetting all internal circuits at a predictable starting point. To ensure proper startup, the RESET input must be held low for a typical 20 to 120 ms. Although simple, the external reset-timer circuit must keep the RESET signal low while the supply voltage (V_{CC}) is below the minimum level allowed for microprocessor operation.

INTEL-1018 at 2; *id.*, at 1 (“The most common microprocessor-supervisory functions include power-on reset...”); INTEL-1008, 171 (“The flash incorporates an inbuilt power-on reset that will put the device into a known state”).

106. In the context of an embedded system, a Power-On Reset refers generally to the known state of system components when they have been powered on (or restarted/reset), before booting up. *E.g.*, INTEL-1008, 281 (“Many processors require an external power-on reset (POR) circuit to bring them to a known state and to commence the execution of software.”). Post’s control circuitry 112 would be a Power-On Reset *Sequencer*, then, in that it sequentially brought components out of this known state by starting the boot process. The ’694 Patent appears to use this term in a consistent manner. INTEL-1001, 3:14-16 (“The method of copying POR Sequencer Descriptor from nonvolatile memory to RAM starts when the supply voltage to the system becomes stable after power-up.”).

107. Post disclosed that control circuitry 112 facilitated a boot process for an electronic device 100 after power-on. “When an electronic device is being powered up or rebooted, an operating system can be booted up from a NVM of the electronic device.” INTEL-1005, 1:16-19. Post’s boot sequence was performed by control circuitry 112:

At bootup of electronic device 100, control circuitry 112 may operate under the control of program code stored in boot ROM 116. ... The program code may be executed to begin loading additional information from NVM 120 into memory 114. This additional information may include firmware. Such as a first stage bootloader, that, once loaded into memory 114, is used by electronic device 100 to load a second stage bootloader or the kernel and operating system of electronic device 100. For clarity, the

program code stored in boot ROM 116 may sometimes be referred to as “mask ROM code,” and the firmware (or other data) retrieved from NVM 120 by the mask ROM code may sometimes be referred to as “boot data.”

INTEL-1005, 3:41-43, 3:55-65; *id.*, 1:19-23 (“For example, the electronic device can be configured to first execute code from a mask read-only memory (“ROM). The code can include instructions to load a bootloader, such as a first stage bootloader, from a flash memory to a volatile memory.”). Post’s Figures 4-6, which I describe in more detail below, illustrated the processes for loading and/or updating boot data performed by control circuitry 112. (*Id.*, 8:9-12 (“The steps of these illustrative processes may be performed by ... control circuitry 112 of FIG. 1.”), 8:15-16 (“illustrative process 400 ... for initiating bootup of an electronic device”), 9:14-15 (“illustrative process 500 ... for loading boot data into memory”), 10:60-62 (“illustrative process 600 ... for storing boot data in a reprogrammable non-volatile memory”).)

108. After the operating system was loaded, “control of control circuitry 112 [w]as transferred to the operating system.” (*Id.*, 4:5-7.) This marked the “completion of the bootup procedure” for the electronic device:

The task of booting up a second stage bootloader and/or the kernel and operating system of the electronic device may then be transferred from the mask ROM code to the first stage bootloader. The successful completion of this bootup procedure is typically critical for the electronic device to function properly.

Id., 1:19-26. Post's control circuitry 112 thus facilitated a boot sequence for electronic device 100.

109. A POSITA would have found it obvious to include the register file and functionality of a Register Load Assist engine described by Wilson in Post's control circuitry. Wilson disclosed a microcontroller that included a processor 101, volatile memory 103, and register file with registers that the processor used during ordinary operation: "Microcontroller 100 can include a programmable processor 101. Register file 102 can include various registers which are used by the processor 101 in the ordinary operation of the microcontroller." INTEL-1013, 2:53-56. Figure 1 showed the components of microcontroller 100:

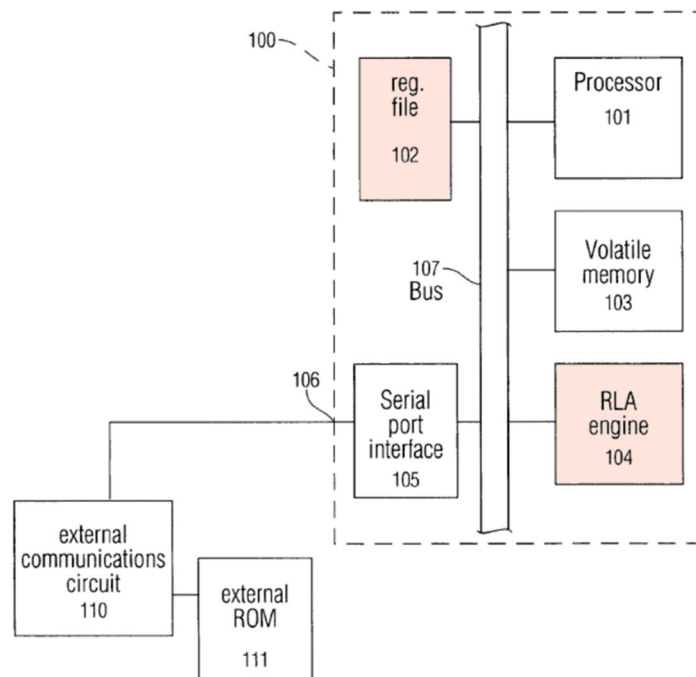


Fig. 1

110. Wilson's microcontroller (like Post's SoC 110) was intended for use in a mobile phone with a touchscreen. "In some embodiments the microcontroller is used to control a portable device, such as, for example a mobile phone featuring a multifunction touch screen. Since the microcontroller can be used to power a small device, it can be custom designed to perform specific predefined functions." INTEL-1013, 3:35-39. The microcontroller could be "custom designed to perform specific predefined functions" for a phone with a multifunction touch screen and included an "unorthodox register file" with registers for configuring the touch screen display and screen pixel colors. INTEL-1013, 3:41-44 ("custom designed microcontroller as featured in certain embodiments of the present invention can include a more unorthodox register file."). These registers included, for example, registers that configured colors of screen pixels:

[T]he size of the various registers can differ from the processor word size. This is the case because registers of unorthodox sizes may be chosen as best Suited for certain specific predefined operations the microcontroller is to perform. For example, the microcontroller can be used to display images on a screen and each five-bit register of the register file 200 can represent a color value for a pixel on a screen.

Id., 3:56-62.

111. Initial values had to be stored in the registers of the register file for microcontroller 100 to operate. "The microcontroller can require that certain

specific predefined initial values are stored in one or more of the registers of the register file before the microcontroller assumes ordinary operation.” INTEL-1013, 2:56-59. Wilson thus disclosed that the boot data included these required initial register values, which were then loaded into the registers of the register file during a boot process:

Register file 102 can include various registers which are used by the processor 101 in the ordinary operation of the microcontroller. ... Since the register file is usually composed of volatile memory, the initial values can be stored in the register file as part of an initialization or boot operation. Therefore, these initial values can be a part of boot data that is loaded in the microcontroller during a boot-up procedure.

Id., 2:53-63; *id.*, 5:1-4 (“The predefined values can be sent from an element external to the microcontroller (such as, for example, external communication circuit 110) to the microcontroller at time of boot up.”); *id.*, 5:30-45, 3:26-34.

112. The functionality of decompressing the boot data into register values and loading them into registers was performed by the Register Load Assist (“RLA”) engine 104 within microprocessor 100. “The RLA engine is used to decompress the data field [of block of boot data 400] into various register initialization values and to load these values into their respective registers.” *Id.*, 6:4-7; *id.*, 5:47-51 (“The boot data can be sent in a single block. ... The block further includes data field 402 which includes the compressed boot data.”). Figure 3 illustrated the steps of a

process that started with “send[ing] [a] set of compressed boot data” to the microcontroller and ended with the RLA engine “decompress[ing] data and load[ing] it into registers” and “activa[ting] processor”:

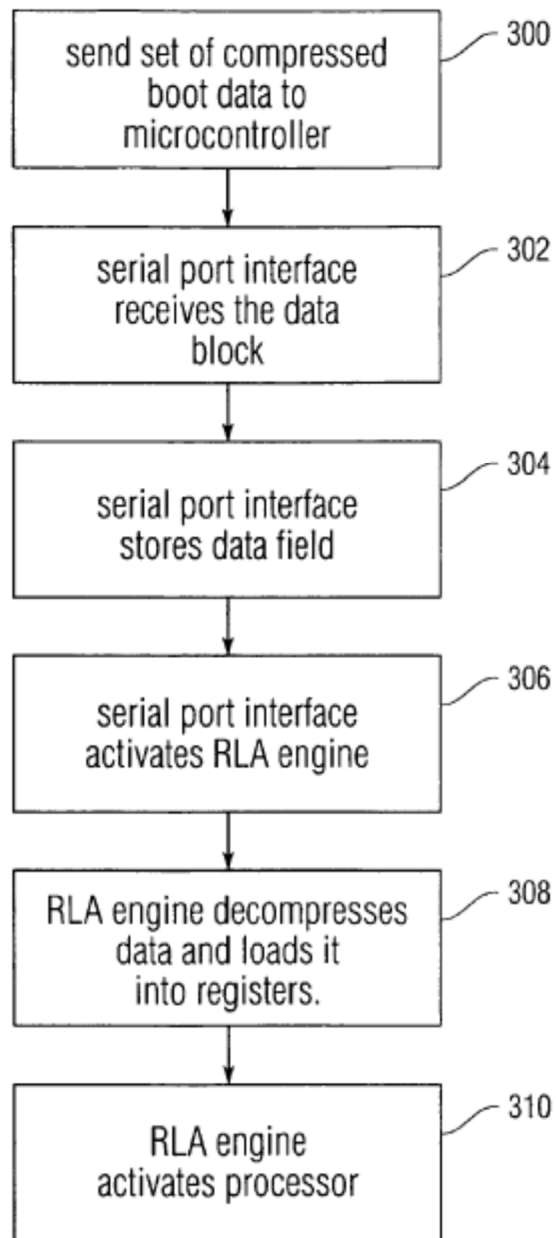
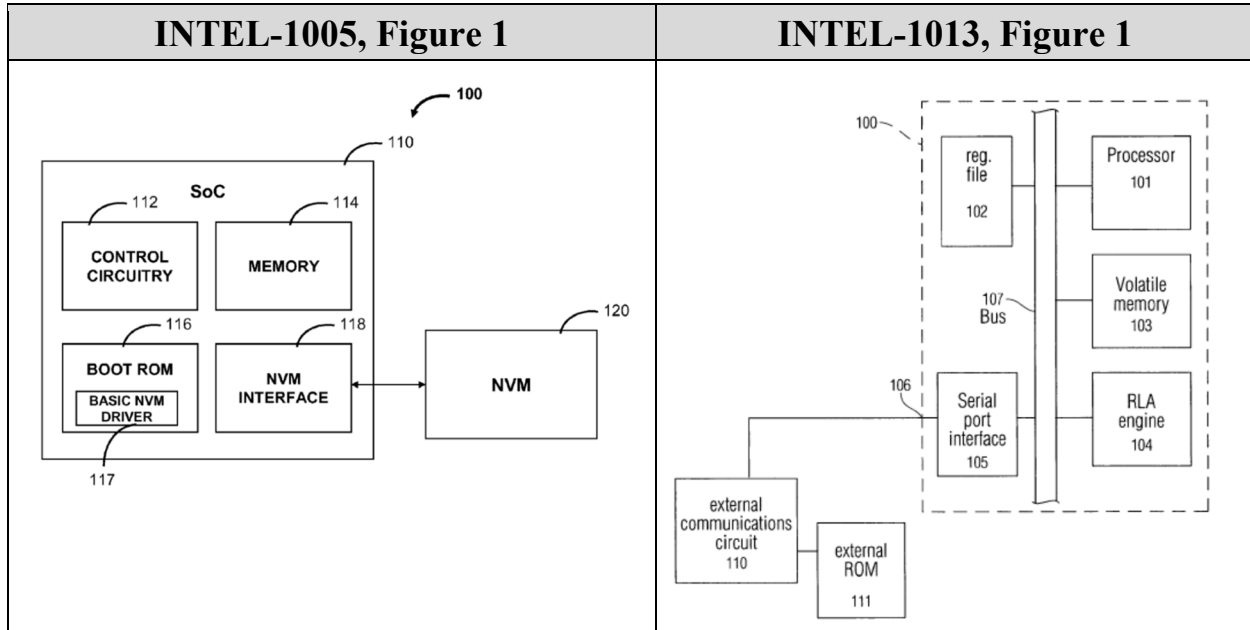


Fig. 3

113. A POSITA would have found it obvious to combine Post with Wilson (such that Post's control circuitry 112 included Wilson's register file and circuitry

to perform the Register Load Assist functionality of RLA engine) for a number of reasons. First, Wilson and Post described systems that were similar in structure, components, and functionality. Wilson's microcontroller was "on a chip" and meant to be included in a mobile phone. INTEL-1013, 1:11-17 ("A microcontroller is often referred to as 'a computer on a chip.' In other words, it is usually a single chip which includes a generally programmable processor as well as various memories and interfaces required to allow the generally programmable processor to operate."), 3:35-37 ("the microcontroller is used to control a portable device, such as, for example a mobile phone featuring a multifunction touch screen"). In exactly the same way, Post's SoC 110 was a "system on chip" to be included in a phone. INTEL-1005, 3:33-38 ("System-on-a-chip 110 can include control circuitry 112, memory 114, boot ROM 116, and NVM interface 118. Control circuitry 112 ... can include any suitable components, circuitry, or logic, such as one or more processors."), 1:61-64 ("electronic device 100 can be ... a cellular telephone (e.g., an iPhoneTM made available by Apple Inc.)").) Comparing Figure 1 of both systems shows their similar components (including processors, on-chip volatile memory, and external NVM):



A POSITA would have viewed Post and Wilson as relating to the same field of endeavor and would have viewed their teachings as interrelated and instructive of design choices that, having improved one system, could be applied in the context of the other in a similar way.

114. Wilson also taught that the boot process for many microcontrollers including initializing the microcontroller's registers with values. "Many microcontrollers require a boot-up process when initially powered up. The boot-up process allows a microcontroller to perform various actions to prepare the microcontroller for ordinary operation, such as load initialization values in its registers, load instructions in its instruction memory, etc." INTEL-1013, 1:24-28.

Indeed, Wilson taught that this initialization was *required* for registers that the microcontroller used during its “ordinary operation”:

Register file 102 can include various registers which are used by the processor 101 in the ordinary operation of the microcontroller. The microcontroller can require that certain specific predefined initial values are stored in one or more of the registers of the register file before the microcontroller assumes ordinary operation.

Id., 2:54-59. A POSITA would have recognized that Post’s SoC, like the similar microcontroller described in INTEL-1013, also included registers used in ordinary operation and would have needed to initialize those registers during the boot process. But Post does not provide express details about how to load initialization values into its registers during boot. A POSITA would have understood Wilson as disclosing one way to implement a process (including initial register values in the boot data) and structure (RLA engine and register file) for loading initial values into registers of Post’s SoC during a boot.

115. Additionally, Wilson taught a benefit to be realized from its techniques and structure for loading initial register values that applied to the specific context of a mobile phone: the benefit of using “unorthodox” registers to custom design the microcontroller registers to perform specific phone-related functions. Wilson described configuring colors of pixels on the phone screen or configuring the multi-touch panel. INTEL-1013, 3:35-62 (“registers of unorthodox sizes may be chosen

as best suited for certain specific predefined operations the microcontroller is to perform. For example, the microcontroller can be used to display images on a screen and each five-bit register of the register file 200 can represent a color value for a pixel on a screen.”); 11:11-39 (host processor 728 can “receiv[e] outputs from multi-touch panel processor 702 and perform[] actions based on the outputs that may include, but are not limited to, moving an object such as a cursor or pointer, scrolling or panning, adjusting control settings, opening a file or document, viewing a menu ...”). “[R]egisters of unorthodox sizes” could be chosen as best suited for these types of “specific predefined operations [of] the microcontroller (*id.*, 3:57-60), and initial values of custom sizes could be included in the boot data (*id.*, 6:49-7:3). Wilson’s disclosure would have prompted a POSITA to include registers of similar “unorthodox sizes” in Post’s SoC 110, for example, to configure screen color and multitouch panel operation in similar ways; and Wilson taught techniques and structure for achieving this. A POSITA would have viewed Post’s expansive teaching that control circuitry 112 “can include any suitable components, circuitry, or logic” (INTEL-1005, 3:36-38) as promoting expansions of this type.

116. A POSITA would have viewed Wilson as providing a teaching, suggestion, or motivation to include registers and Register Load Assist functionality in Post’s control circuitry 112 because Wilson taught that register values must be loaded before a microcontroller can begin normal operation. INTEL-1013, 4:59-61

(“for microcontroller 100 to commence ordinary operation, it can be required that certain predefined values be loaded in the registers of register file 102”); 2:56-59 (“The microcontroller can require that certain specific predefined initial values are stored in one or more of the registers of the register file before the microcontroller assumes ordinary operation”).

117. A POSITA would also have viewed Wilson’s described techniques and structure to provide an additional operational benefit—improving the speed of the boot process. Wilson taught that by including initial register information along with the other boot data, such as initial memory values and instructions, and *compressing* all of that data, the entire boot package could be loaded quickly, as a single block, from external NVM. INTEL-1013, 5:10-26 (“the bootup communications can take up too much time. ... a slow boot up process can create a negative impression in the mind of a consumer. Therefore, a faster scheme for loading predefined values into the registers on boot up is required. For this purpose, embodiments of the present invention provide for sending compressed boot data into microcontroller 100.”); 5:46-47 (“The boot data can be sent in a single block. FIG. 4 shows an exemplary block of boot data 400.”), 6:42-44 (“the compressed boot data can include data other than initial register values. For example, it can include initial memory values, or initial instructions.”). The compressed boot data, including register values, was then decompressed by the RLA engine. *Id.*, 6:27-29 (“At step 308, the RLA engine

accesses the compressed boot data from the volatile memory, decompresses it and uses it to load the various registers in register file 102.”), 6:45-48 (“the RLA engine can decompress and load instructions in an instruction memory (which can be part of volatile memory 103), or can decompress other initial memory values and load them in their intended memory locations”). A POSITA would have recognized that the compression and consequent speed benefits described by Wilson—and achieved through the compressed boot data including register values, and the RLA engine functionality—would apply to and improve Post’s system in the same way.

118. A POSITA would also have had a reasonable expectation of success in making the combination of Wilson’s register load assist functionality and register file with Post’s control circuitry 112. Processor registers were well known. *E.g.*, INTEL-1016, 137 (“A register is simply fast programmable memory normally used to store operands that are immediately or frequently used. A processor’s set of registers is commonly referred to as the register set or the register file.”), 156 (“Registers are simply a combination of various flip-flops that can be used to temporarily store data or to delay signals.”). Techniques for initializing registers during a boot process were also well known. INTEL-1013, 1:24-28 (“Many microcontrollers require a boot-up process when initially powered up ... [which] allows a microcontroller to perform various actions to prepare the microcontroller for ordinary operation, such as load initialization values in its registers ...”); INTEL-

1016, 555 (“The same (minimal) general functions are performed by boot code across the various platforms, which [include] initializing the hardware ...”). And Wilson’s disclosure shows that POSITAs knew how to custom-design a microcontroller (one that was similar to Post’s SoC in design and operation) to include configuration registers and circuitry that initialized these registers with values included in boot data. Wilson even gave implementation suggestions such as using a “generally programmable processor” or “an application specific circuit ... within a larger application specific integrated circuit (ASIC)” to implement the RLA engine functionality. INTEL-1013, 6:14-24. Therefore, a POSITA would have been motivated by Wilson to implement Wilson’s register load assist functionality and register file as an application specific circuit within control circuitry 112, and would have had a reasonable expectation of success in doing so.

119. A POSITA would also have been motivated to make this combination because Wilson showed that there was a known way to solve the known problem of initializing processor register information: include initial register values in the boot data, and use circuitry within a microcontroller/SoC to decompress and load that information into registers. Doing so would have been within the level of ordinary skill, because a POSITA would have been generally familiar with processor registers and techniques for booting processors. (INTEL-1013, 1:25-28, 2:54-63, 6:27-41, Figs. 1, 3; INTEL-1016, 570~~555~~; INTEL-1009, 137; 1:14-26.) Further, Wilson

provided a concrete example of a POSITA doing exactly this for a microcontroller similar to Post's SoC.

120. A POSITA would have been prompted by similarities between Post and Wilson to look to Wilson for additional teachings in order to improve Post's boot process. Post and Wilson both taught storing boot data in an external nonvolatile memory of an embedded system. INTEL 1005, 1:32-34 ("Systems and methods are provided for storing and retrieving boot data, such as a bootloader, in and from a NVM (e.g., NAND flash memory); INTEL-1013, 1:57-60 ("The microcontroller can include zero or minimal non-volatile memory which stores boot data. Thus, most of the boot data can be stored at a non-volatile memory external to the microcontroller."). The focus of both Post and Wilson was improving the boot process in an embedded system. INTEL-1013, 2:4-8 ("Thus, by allowing the remote receipt of compressed boot data, the RLA engine provides for a microcontroller which can receive a large amount of boot data from an external source, but nevertheless feature a relatively fast boot-up sequence."); INTEL-1005, 1:35-38 ("The disclosed embodiments provide, among other things, techniques to ensure the reliable storage and retrieval of such boot data, thereby allowing for consistent and successful startup of an electronic device.".) Both Post and Wilson described techniques for booting an embedded system within "a portable device, such as, for

example a mobile phone featuring a multifunction touch screen.” (INTEL-1013, 3:35-37; INTEL-1005, 2:60-67 (“electronic device 100 can be ... an iPhone”).

121. For these illustrative reasons, it would have been obvious to combine the teachings of Post and INTEL-1013, such that Post’s control circuitry 112 would have been modified to include Wilson’s register file and circuitry to perform the register loading functionality of Wilson’s RLA engine. The control circuitry 112 of this combination was a “Power-on Reset Sequencer.”

b. configured to detect, in the nonvolatile memory, a descriptor used in a boot process

122. As discussed, in the asserted combinations, control circuitry 112 would be modified to include Wilson’s register file and circuitry to perform the register loading functionality of Wilson’s RLA engine control circuitry 120. This control circuitry 112 was configured to detect boot data and associated pointer metadata, which was together a “*a descriptor used in a boot process*,” in NVM 120 (“*in the nonvolatile memory*”). As discussed in connection with Sec. X.F.6, in the asserted combinations Post’s boot data would include register information (as taught by Wilson), and Post’s pointers would be DMA descriptors.

123. Control circuitry 112 loaded boot data that included firmware, such as a first stage bootloader, or other data, from NVM 120 into memory 114 at bootup of electronic device 110:

At bootup of electronic device 100, control circuitry 112 may operate under the control of program code stored in boot ROM 116.

...

Boot ROM 116 may be configured to store program code that is executed by control circuitry 112 on bootup of electronic device 100. The program code may be executed to begin loading additional information from NVM 120 into memory 114. This additional information may include firmware, such as a first stage bootloader, that, once loaded into memory 114, is used by electronic device 100 to load a second stage bootloader or the kernel and operating system of electronic device 100. For clarity, ... the firmware (or other data) retrieved from NVM 120 by the mask ROM code may sometimes be referred to as “boot data.”

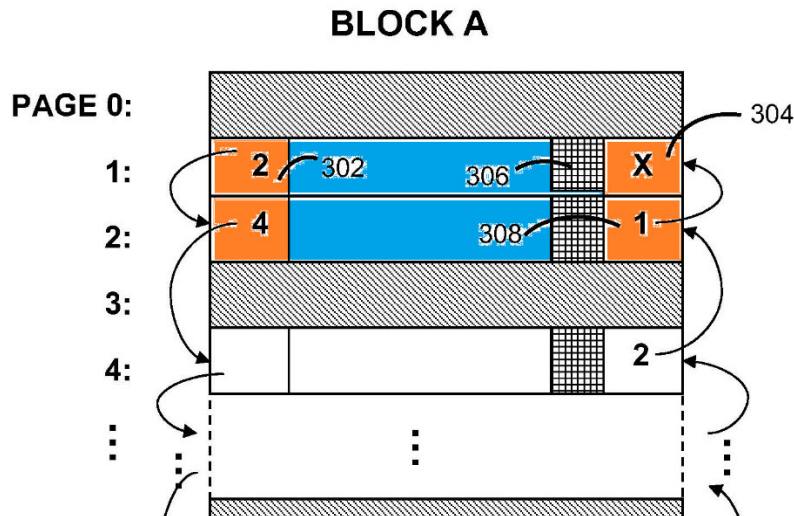
INTEL-1005, 3:41-43, 3:52-65.

124. Post described boot data being stored in a linked-list format in pages (units) of NVM 120. *Id.*, 6:19-24 (“FIG.3 illustrates ways in which a linked-list format may be used to store a map of which pages are currently used as boot data pages. FIG. 3 is a graphical view of two illustrative blocks, blocks A and B, of a non-volatile memory. The graphical view of blocks A and B may represent the state of two blocks in the boot data storage area of NVM 120 (FIG. 1).”). Next-page and prior-page pointer metadata were also stored in NVM 120 in the boot data pages.

In blocks A and B, each of the boot data pages can store both data (e.g., boot data) and metadata. The metadata for page 1, for example, can include a next page pointer 302 and a prior page pointer 304. The boot data pages of blocks A and B may form a first linked list using the next page

pointers, including next page pointer 302. The boot data pages can also form a second linked list using the prior page pointers, including prior page pointers 304 and 308. The metadata in each boot data page can further include a boot data page indicator, such as boot data page indicator 306.

Id., 6:28-38. Figure 3 showed pages of nonvolatile memory 120. As shown in the highlighted version of Figure 3, boot pages 1 and 2 included boot data (in blue) as well as next-page and prior-page pointer metadata (in orange):



(*Id.*, Fig. 3.)

125. A POSITA would have understood that in the asserted combinations, the boot data stored in NVM 120 would have included initial register values. Wilson disclosed that register values were in the boot data loaded during a boot up procedure, and that the register file (which the asserted combinations include) needed these register values to be initialized during the boot process. “The

microcontroller can require that certain specific predefined initial values are stored in one or more of the registers of the register file before the microcontroller assumes ordinary operation. ... [T]hese initial values can be a part of boot data that is loaded in the microcontroller during a boot-up procedure.” INTEL-1013, 2:56-63; *id.* 6:50-54 (“Compressed boot data 500 is shown. ... The compressed data may include a plurality of initial register values intended for the register file.”).

126. Given that the combination already includes Wilson’s register file, and that this register file must be initialized with values before the microcontroller assumes ordinary operation, and that it also includes circuitry to load register values into registers, the boot data of the combination would have included these initial register values disclosed by Wilson. At the very least, a POSITA would have found it obvious for the boot data to include initial register values (as described by Wilson) for the same reasons I discussed with respect to the “POR sequencer” element of claim 1[b]. *See* Sec. X.F.3.a.

127. The asserted combinations also disclosed “*detecting, in the nonvolatile memory, a descriptor,*” because Post disclosed various ways of searching for and identifying boot data and metadata in the nonvolatile memory. First, after control circuitry 112 read a page from NVM, it detected whether that page contained boot data (part of the “*descriptor*”):

At step 406, the boot ROM data may instruct the electronic device to read an initial page from the nonvolatile memory. ... Then, at step 408, the electronic device may determine whether the initial page contains boot data. In some embodiments, this determination can be performed by employing an error correcting code on the initial page data to check for a valid codeword, or by checking a program status of the initial page to determine whether the initial page is programmed or unprogrammed. In addition or alternatively, this determination can involve checking for a boot data indicator or a non-data indicator in the metadata of the current initial page.

INTEL-1005, 8:29:42. If the current page was not boot data, control circuitry performed this process on additional pages to “determine whether storage of the boot data begins at a subsequent page.” *Id.*, 8:43-49.

128. Control circuitry 112 also detected the case where a first page of boot data could not be found and tried to detect a backup copy of the boot data in NVM:

The electronic device may use the prior page pointers, in particular, to account for the first boot data page. ... Thus, the electronic device can read and/or process the sequence of pages from page 0 to page 1 to page 2, and so on, looking for the first boot data page. ... In this case, because prior page pointer 308 points to page 1 instead of indicating that page 2 is the first page, the electronic device is able to determine that boot data from at least one boot data page has been lost. The electronic device may therefore signal a boot failure or may try loading a redundant copy of the boot data from another location in the NVM.

Id., 7:34-50; *id.*, 8:53-9:2:

In particular, at Step 414, the electronic device can determine whether the prior page pointer indicates that the

current page is the first page of boot data. If the prior page pointer points to a previous page instead of showing that the current page is the first page, this is an indication that the electronic device was previously unsuccessful at reading the first boot data page. That is, the current page may be a second or subsequent page used to store boot data, and the first page of boot data may have been lost due to error. Because the electronic device is missing boot data, which may be critical for device operation, the electronic device may signal a boot failure at step 416. For example, the electronic device may stop the bootup process and may display an error message to the user. Process 400 may then end at step 418 so that the user can perform a restore operation to update the firmware (including the boot data) stored in the non-volatile memory.

The '694 patent described “detecting” a POR sequencer descriptor in this same way. INTEL-1001, 6:64-7:12 (“[i]f the first copy [of the POR sequencer descriptor] is unsuccessfully detected in step 604, in block 609, the POR sequencer will try to detect a second copy of the POR sequencer descriptor ... [in] the nonvolatile memory.”).

129. The boot data and metadata of the asserted combinations was also *used in a boot process*. Control circuitry 112 used pointer metadata to load more boot data and to verify the boot data. It used boot data to load a second stage boot loader, and the kernel and operating system. INTEL-1013., 3:55-61, 9:47-62.) This loading and processing of the boot data was “critical for the electronic device to function properly” and “allow[ed] for consistent and successful startup of an electronic

device.” *Id.*, 1:26-28, 1:34-37. Wilson also disclosed that initial register values (part of the boot data of the asserted combination) were used to initialize system registers and that this was “required” before ordinary operation could proceed. INTEL-1013, 2:56-63. The ’694 patent describes using the POR sequencer descriptor in a boot-up sequence in a similar manner. *E.g.*, INTEL-1001, 2:50-54 (“The POR Sequencer descriptor contains register information and direct memory access (DMA) controller descriptors. The register information are read and used by the POR Sequencer to update the configuration registers of the system.”).

c. and configured to copy the descriptor from the nonvolatile memory to the RAM

130. The control circuitry 112 of the asserted combinations was *configured to copy the descriptor* (the boot data, including initial register values, and pointer metadata) from the *nonvolatile memory* (NVM 120) *to the RAM* (memory 114). Post disclosed that control circuitry 112 copied boot data from nonvolatile memory 120 into memory 114. INTEL-1005, 3:53-65. Figure 2 showed this copying operation:

FIG. 2 is [a] graphical view of memory 220 and of a portion of a block 212 from illustrative NVM 210. In particular, FIG. 2 provides a graphical view of the operation performed to load boot data from block 212 to memory 220 during bootup of an electronic device. Block 212 includes a sequence of pages (e.g., pages 0, 1, 2, and so on), ... [and] may illustrate the state of a block in the boot data storage area of NVM 120 (FIG. 1), and memory 220 may illustrate the state of memory 114 (FIG. 1) after

electronic device 100 (FIG. 1) has finished loading the boot data from memory 114.

Id., 5:11-23; *id.*, Fig. 2:

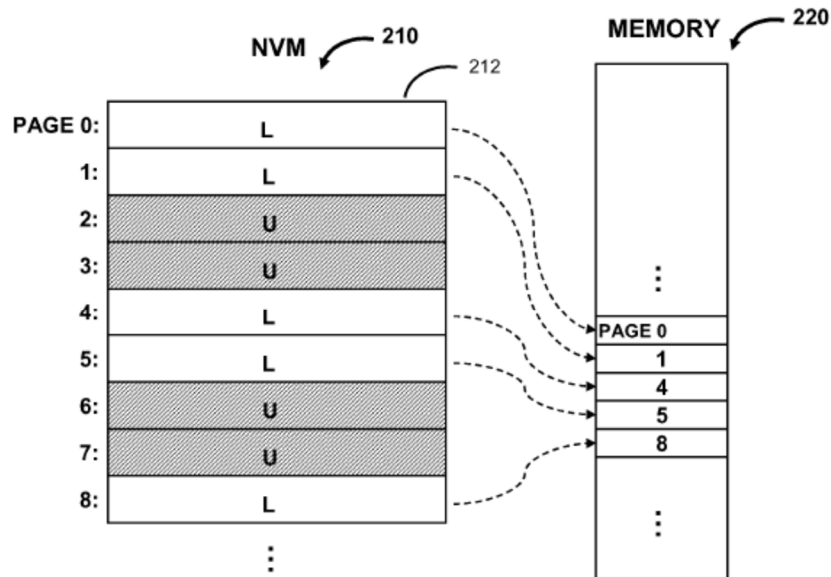


FIG. 2

Memory 114 was “any suitable type of volatile memory, such as random access memory (RAM).” *Id.*, 3:44-49.

131. Post also disclosed that pointer metadata was saved in memory 114 as part of the process of copying boot data from NVM 120: “If the current page [of boot data] is not the last page, process 500 may instead continue to step 516, where the electronic device can *save the current page’s next page pointer for use in processing subsequent pages.*” *Id.*, 9:56-60. Control circuitry 112 accessed and used this saved pointer metadata to detect the presence of additional pages of boot data:

[A]t step 517, the electronic device can refer to *a next page pointer, which was saved from a previous boot data page* (e.g., from a previous iteration of step 516). The electronic device can determine whether this saved next page pointer references a subsequent page (and not the current page). If the electronic device determines that the saved next page pointer references a different page, and therefore the current page should not contain boot data, process 500 can return to step 504 without maintaining information from the current non-data page.

Id., 9:65-10:7.

132. Post's disclosure of "saving" pointer metadata, and using this metadata in detecting additional boot pages to be loaded from NVM, indicated that the metadata was being stored in memory 114, which is where the other boot data was being stored. Post described embodiments where all data from the boot storage portion of NVM 120 was stored in memory 114, together:

[T]he electronic device can first ***load the contents of all of the pages in the boot data storage area*** (including the non-data pages) ***into memory 220***. The loaded content may then be post-processed to discard the unneeded information obtained from the non-data pages."

INTEL-1005, 5:53-57. At the very least, Post strongly suggests saving the pointer metadata in memory 114, to be consistent with the location of the other boot data. Control circuitry 112 was already reading from and writing to memory 114 to perform the functions of loading boot data from NVM 120 and then executing the boot data; it would have been efficient and straightforward to use this same memory for reading and writing pointer metadata. Indeed, storing the metadata in a memory

different from memory 114 would have added an additional step to the process, reducing efficiency and adding complexity to the process. Further, saving the metadata in RAM would have allowed control circuitry 112 to access it quickly during the boot loading process, which would have further improved boot speed. The asserted combinations thus disclosed that the POR sequencer was *configured to copy* the boot data and pointer metadata (“*descriptor*”) from NVM 120 to memory 114, and this would also have been obvious.

4. Claim 1, Limitation c

wherein the POR sequencer facilitates a boot process and verifies an integrity of loaded data and code in the RAM;

a. Wherein the POR sequencer facilitates a boot process.

133. As I demonstrated above in Section X.F.3, the asserted combinations disclosed a power-on reset (POR) sequencer. Post’s control circuitry 112, modified to include Wilson’s register file and circuitry to perform the register loading functionality of the RLA engine, was a “POR sequencer.” This POR sequencer *facilitate[d] a boot process*, as I demonstrate below.

134. Post taught that control circuitry 112 facilitated a boot process by executing processes for “initiating boot up of an electronic device,” shown in Post’s Figure 4, “loading boot data into memory,” shown in Figure 5, and “storing boot data in a reprogrammable non-volatile memory,” shown in Figure 6. (INTEL-1005, 8:15-16, 9:14-15, 10:60-62; *id.*, 8:7-12 (“Referring now to FIGS. 4-6, flowcharts of

illustrative processes are shown.... The steps of these illustrative processes may be performed by any suitable component or combination of components in an electronic device, such as by control circuitry 112 of FIG. 1.”.)

135. Control circuitry 112 initiated boot up of System on Chip 110 (“an electronic device”) by executing a process to locate the first page of boot data:

Turning first to FIG. 4, a flowchart of illustrative process 400 is shown for initiating bootup of an electronic device. Process 400 may be performed to identify which memory location of a non-volatile memory contains the first page of boot data (e.g., bootloader data). Using the steps of process 400, boot data does not need to be stored at the first memory location addressed by the mask ROM code, and can instead begin at a suitable memory location (e.g., more reliable location) for a particular non-volatile memory.

Process 400 may begin at step 402 in response to, for example, a startup or reboot command of the electronic device.

(INTEL-1005, 8:15-27). Figure 4 showed the steps of the process:

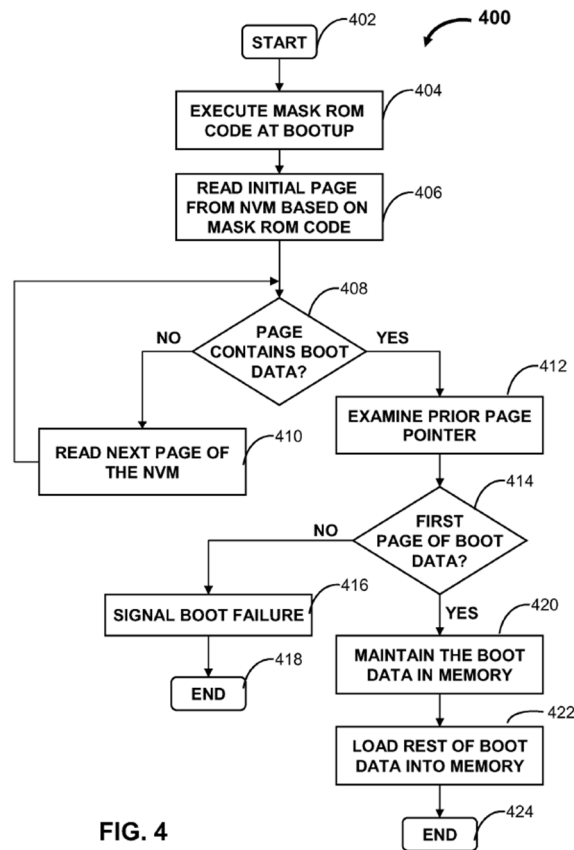
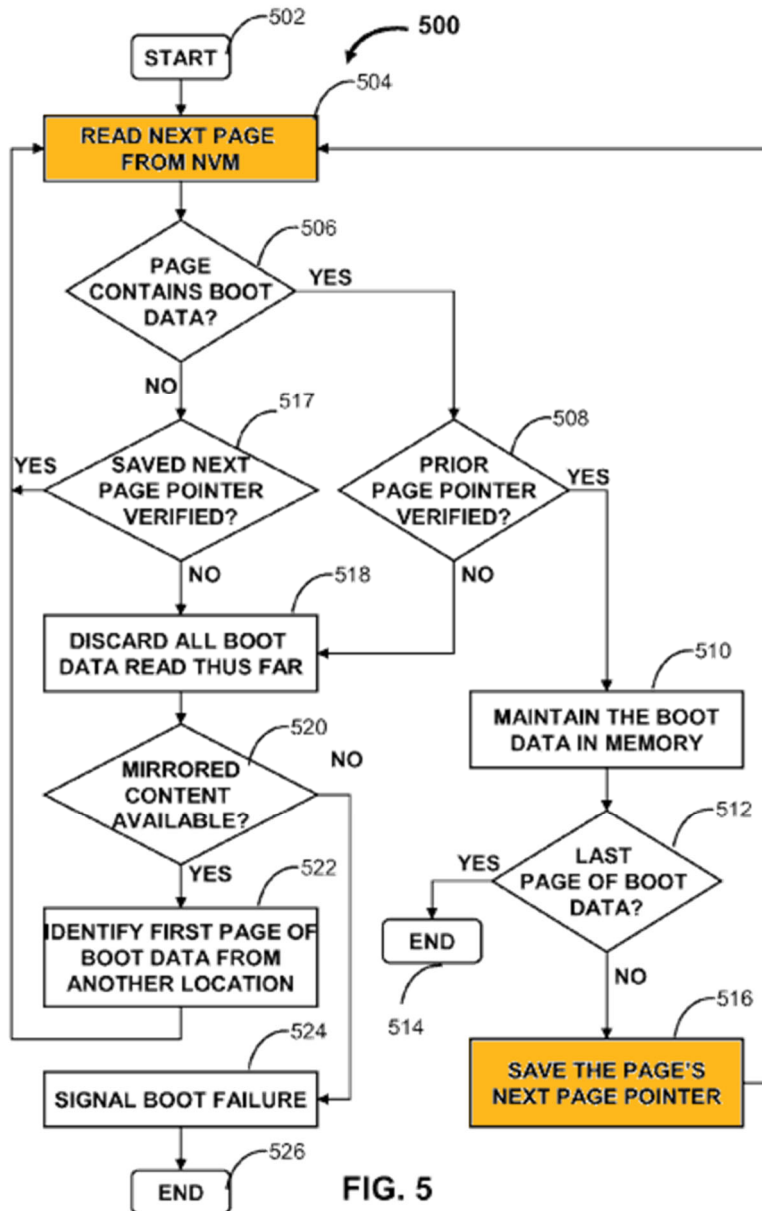


FIG. 4

(*Id.*, Fig. 4; 8:25-9:13.)

136. Post further taught that control circuitry 112 used pointers to fetch boot data and associated metadata stored in NVM 120 and copy it to system RAM (memory 114 of SoC 110, as shown in Post’s Fig. 1). The process “for loading boot data into memory” was shown in annotated Figure 5 below.



(*Id.*, 9:14-15.)

137. The process began with reading a page from the nonvolatile memory and determining whether the current page contained boot data:

Process 500 may begin at step 502. At step 504, the electronic device may read a page from the non-volatile

memory. For example, once the electronic device identifies and processes the first boot data page (i.e., using process 400 of FIG. 4), the electronic device can read from the page immediately following the first boot data page.

Then, at step 506, the electronic device may determine whether the current page contains boot data.

(*Id.*, 9:19-26; *id.*, Fig. 5 (step 504).) Post's device examined the next page pointer of the current boot data page to determine whether the pointer referenced a subsequent page of the NVM:

Following step 510, the electronic device can determine at step 512 whether the current page contains the last page of boot data. For example, the electronic device can examine the next page pointer to determine whether the pointer references a subsequent page of the non-volatile memory, or whether the pointer indicates that there are no more pages of boot data.

(*Id.*, 9:47-52.) If the pointer referenced another boot data page, the device saved the current page's next page pointer for use in processing later pages of boot data:

If the current page is not the last page, process 500 may instead continue to step 516, where the electronic device can save the current page's next page pointer for use in processing subsequent pages.

(*Id.*, 9:56-60.) The process then "return[ed] to step 504 [i.e., reading a page from the non-volatile memory] to obtain the rest of the boot data." (*Id.*, 9:60-62.) Post described repeating this process until there were no more pages of boot data, at which point all the boot data was stored in system RAM:

Following step 510, the electronic device can determine at step 512 whether the current page contains the last page of boot data. For example, the electronic device can examine the next page pointer to determine whether the pointer references a subsequent page of the non-volatile memory, or whether the pointer indicates that there are no more pages of boot data. If the current page is the last page, process 500 may end at step 514. At this point, all of the boot data may be stored in memory, so the electronic device may transfer control of the bootup process from the mask ROM code to the boot data.

(*Id.*, 9:49-55.)

b. And verifies an integrity of loaded data and code in the RAM.

138. As I demonstrate below, Post taught that its control circuitry 112 *verified an integrity* of the boot data that was loaded into memory 114 (“*loaded data and code in the RAM*”) using next page pointers 302, prior page pointers 304, and/or boot data page indicators 306. (INTEL-1005, 1:62-64 (“Using these pointers, the electronic device can load the boot data and verify that pages expected to have boot data actually contain boot data (and vice versa).”); 7:61-64 (“Boot data indicator 306, as well as the other boot data indicators, can take on any suitable value. These boot data indicators can provide another way in which the electronic device can verify that a page is a boot data page.”).)

139. Post taught that its next and prior page pointers and boot data indicator could be used to “verify the integrity” of boot data. (*Id.*, 6:54-55 (“In some embodiments, the electronic device can use the next page pointers to verify the

integrity of boot data storage.”); 7:21-23 (“As with the next page pointers, the prior page pointers may allow an electronic device to verify and account for all of the boot data stored in the NVM.”); 7:58-64 (“Boot data indicators 306 ... can provide another way in which the electronic device can verify that a page is a boot data page”).) Post’s page pointers provided a map of locations of boot data in the NVM allowed the electronic device to verify that all boot data was accounted for:

In some embodiments, the NVM can store a map indicating which memory locations are being used to store boot data. For example, using a linked-list format, each page storing boot data can have metadata that includes a “next page pointer” pointing to the next page that contains boot data. In some embodiments, each of these pages may include a “prior page pointer” pointing to a previous page that includes boot data. Using these pointers, the electronic device can load the boot data and verify that pages expected to have boot data actually contain boot data (and vice versa). If such verification fails, the electronic device can signal a boot failure unless a redundant copy of the boot data is stored elsewhere in the boot data storage area. Thus, storing a map of pages containing boot data can provide a way to verify that all of the boot data is accounted for and that the boot data can be properly loaded from the NVM.

(*Id.*, 1:55-2:3.) If boot data transferred from NVM to system RAM 114 could be verified, it would be retained in system RAM:

If the electronic device determines that the current page contains boot data, the electronic device can verify the prior page pointer of the current page data. That is, at step 508, the electronic device can confirm whether the prior

page pointer points back to the expected page (i.e., the most recently read and/or processed boot data page).

If, at step 508, the prior page pointer can be verified, the electronic device can maintain the boot data from the current page in memory. Thus, by performing steps 506 and 508 before deciding to maintain the boot data, the electronic device can perform several checks on the validity and accuracy of the boot data stored in the non-volatile memory. This may be beneficial, because accurate boot data may be critical to the operation of the electronic device.

(*Id.*, 9:33-46.)

140. Post taught an additional way—error codes used in conjunction with page pointers—in which control circuitry 112 could verify an integrity of the boot data that was loaded in memory 114:

In some embodiments, the electronic device can use the next page pointers to verify the integrity of boot data storage. For example, since next page pointer 302 points to page 2, page 2 is expected to contain boot data. The electronic device may confirm that page 2 provides boot data by, for instance, employing an error correcting code on the contents of page 2. If the error correction results in uncorrectable errors, the boot data for page 2 may be unobtainable, and the electronic device may not have all of the boot data necessary for device operation (unless another copy of the boot data is located elsewhere). In this case, the electronic device may signal a boot failure. Without next page pointer 302 (or other suitable metadata), it would be unclear whether the failed verification indicates that boot data has been lost (as would be the case with page 2) or whether the failed verification indicates that the page is actually a non-data page (as would be the case if verification were attempted on page 3). Thus, the linked-list format provided by the next page

indicators can ensure that all of the pages containing boot data are accounted for.

(*Id.*, 6:54-7:5.)

141. Post taught, and a POSITA would have understood, that Post's integrity verification functions were performed by control circuitry 112. (*Id.*, 8:7-12 ("Referring now to FIGS. 4-6, flowcharts of illustrative processes are shown.... The steps of these illustrative processes may be performed by ... control circuitry 112 of FIG. 1.").) Those processes included verifying the integrity of boot data that was loaded from nonvolatile memory using pointers and error correcting codes. A POSITA would also have found it obvious for Post's control circuitry 112 to perform Post's integrity verification functions because Post described control circuitry 112 as "control[ing] the general operations and functions of SoC 110 and electronic device 100 in general" and including "any suitable components, circuitry, or logic, such as one or more processors." (*Id.*, 8:33-38.)

5. Claim 1, Limitation d

wherein the descriptor comprises a POR sequencer descriptor, and wherein the POR sequencer uses the POR sequencer descriptor which is a preassembled descriptor that is stored in the nonvolatile memory,

142. The boot data of the asserted combinations (which included initial register values) and associated pointer metadata facilitated a boot process and thus *comprised a POR [Power-On Reset] sequencer descriptor*. As I discussed for claim

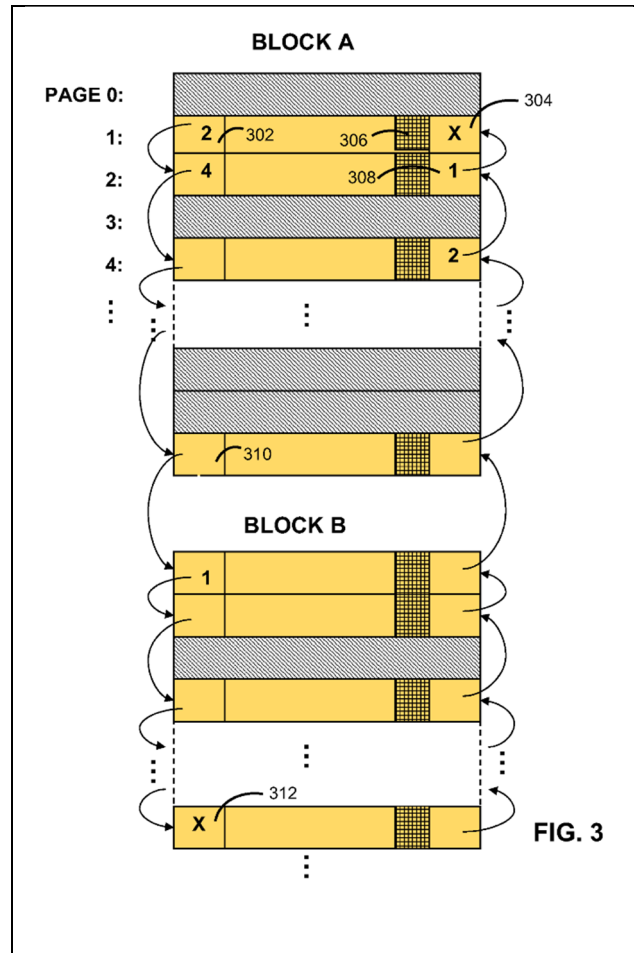
1[b] (“*configured to detect ... a descriptor used in a boot process*”), pointer metadata was used to load additional boot data; and the boot data was used to load initial values into registers and to load a second stage bootloader and device kernel and operating system to complete the boot process. *See* Sec. X.F.3.b. Control circuitry 112, which (as modified to include Wilson’s register file and circuitry to perform the register loading functionality of Wilson’s RLA engine) was a POR sequencer, was the component that loaded and executed the boot data and initialized register values. *See* Sec. X.F.3.b, X.F.4. Thus, the asserted combinations disclosed that *the POR sequencer use[d] the POR sequencer descriptor*. The ’694 patent similarly describes the POR sequencer descriptor as being used by the POR Sequencer to facilitate a boot process. INTEL-1001, 2:47-50 (“The POR Sequencer makes use of a POR Sequencer Descriptor which is a preassembled descriptor that is stored in a nonvolatile memory of the system.”), Abstract (“The POR sequencer controls the overall boot process of the embedded system. Said sequencer uses descriptors (POR Sequencer descriptors)[.]”).

143. The POR sequencer descriptor of the asserted combinations was stored as pages of boot data in NVM 120 (“*preassembled descriptor that is stored in the nonvolatile memory*”). Post’s NVM 120 stored boot data pages in a linked-list format, where each boot data page included both the boot data and next-page and prior-page pointer metadata:

FIG.3 illustrates ways in which a linked-list format may be used to store a map of which pages are currently used as boot data pages. FIG. 3 is a graphical view of two illustrative blocks, blocks A and B, of a non-volatile memory. The graphical view of blocks A and B may represent the state of two blocks in the boot data storage area of NVM 120 (FIG. 1). ... In blocks A and B, each of the boot data pages can store both data (e.g., boot data) and metadata. The metadata for page 1, for example, can include a next page pointer 302 and a prior page pointer 304. The boot data pages of blocks A and B may form a first linked list using the next page pointers, including next page pointer 302. The boot data pages can also form a second linked list using the prior page pointers, including prior page pointers 304 and 308.

INTEL-1005, 6:19-35. Figure 3 illustrated this linked-list configuration in NVM

112:



INTEL-1005, Fig. 3. The boot data of the asserted combinations included initial register values (see Sec. X.F.3.b). Wilson disclosed, consistently, that initial register values were stored in boot data in NVM.

144. POSITA would have understood the pages of boot data and pointer metadata, stored in NVM 120, to be a *preassembled descriptor that is stored in the nonvolatile memory*. The '694 patent described precisely this same type of linked-list storage of the POR sequencer descriptor. INTEL-1001, Fig. 5 & 2:59-63 ("The POR Sequencer Descriptor of the current invention is not limited to a single data

structure. A linked-list of data structures is presented”). Wilson similarly disclosed storing boot data, including register initial values, in nonvolatile memory. (INTEL-1013, 2:62-63 (“these initial [register] values can be a part of boot data that is loaded in the microcontroller during a boot-up procedure”), 3:29-31 (“An external nonvolatile memory, such as external ROM 111, can also be provided to store boot data.”)).

6. Claim 1, Limitation e

wherein the POR sequencer descriptor includes register information and DMA controller descriptors, and

a. wherein the POR sequencer descriptor includes register information

145. As I discussed for claims 1[b] and 1[d] (*see* Secs. X.F.3.a-b, X.F.5) a POSITA would have understood the boot data of the asserted combinations to include initial register values, as disclosed by Wilson, and it would also have been obvious. Briefly, Wilson disclosed including initial register values in the boot data for a microcontroller embedded system:

The microcontroller can require that certain specific predefined initial values are stored in one or more of the registers of the register file before the microcontroller assumes ordinary operation. Since the register file is usually composed of volatile memory, the initial values can be stored in the register file as part of an initialization or boot operation. Therefore, these initial values can be a part of boot data that is loaded in the microcontroller during a boot-up procedure.

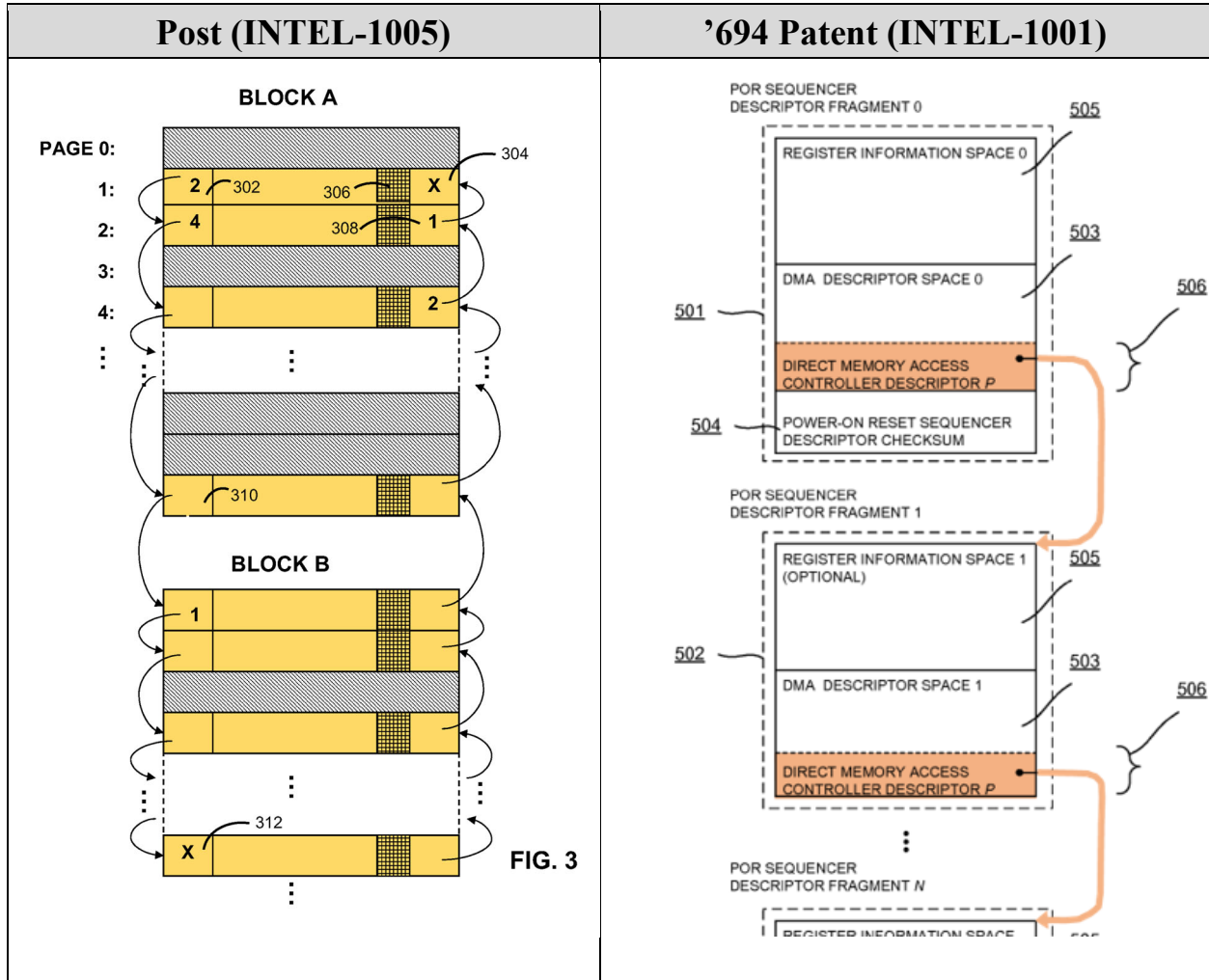
INTEL-1013, 2:56-63; *id.*, 6:50-54 (“Compressed boot data 500 is shown. ... The compressed data may include a plurality of initial register values intended for the register file.”). Given that the asserted combination included both Wilson’s register file and circuitry to perform the register load assist functionality of Wilson’s RLA engine, a POSITA would have been motivated to include initial register values in the boot data (as taught by Wilson) to facilitate register initialization necessary as a precursor to “ordinary operation.” I incorporate my other opinions as discussed above in Secs. X.F.3.a-b, X.F.5.

**b. wherein the POR sequencer descriptor includes ...
DMA controller descriptors**

146. A POSITA would have understood that in the asserted combinations, in which control circuitry 112 was modified to include a DMA controller that performed the boot data transfer functionality of Kao’s DMA engine (*see* Sec. X.F.2), the next-page pointer metadata included in the *POR sequencer descriptor* were *DMA controller descriptors*. Post’s pointer metadata identified the next page of boot data by providing the address of a specific page in NVM 120: “Looking first at the linked-list formed by the next page pointers, the next page pointers in blocks A and B can each reference the next boot page in the sequence. Next page pointer 302 of page 1, for example, can provide the address of page 2.” INTEL-1005, 6:38-42; *id.*, 6:51-53 (a “next page pointer can be an address referencing another page in

the same block”). Post described one embodiment in which “[boot data] pages may be selectively read such that some or all of the non-data pages are skipped over.” *Id.*, 10:48-59. A POSITA would have understood this to indicate another use of the boot page pointer metadata: control circuitry 112 would use each pointer to identify the address of the next boot data page, and “skip over” intervening non-boot data pages to retrieve only the page at the pointed-to address in NVM 120. Indeed, Post does not describe another way to perform this “selectively read” scenario, other than using the addresses of the pointer metadata.

147. The linked-list format of Post’s boot data, with next-page pointers pointing to the address of subsequent boot pages in NVM 120, is strikingly similar to the ’694 patent’s description of DMA controller descriptors that point additional locations in nonvolatile memory:



148. Control circuitry 112 of the asserted combinations would be modified to include a DMA controller that performed the functions of transferring pages of boot data from NVM 112 to memory 114, e.g., for pages subsequent to the first identified boot data page, as described in Figure 5 of Post. *See* Sec. X.F.2. Kao taught that the DMA controller loaded boot codes from nonvolatile memory automatically into volatile memory upon being provided with, among other information, the source address of the data to be transferred:

In one embodiment of the invention, to perform the initial transfer of the boot codes 132 from the NAND flash memory 130 to the boot code buffer 212, the first register 502 of the DMA engine 120 may be initialized with a source address SA and destination address DA and a size of the boot codes 132 being transferred. In some cases, the initialization information may be preset (e.g., stored in a ROM or hard-wired) in the DMA engine 120. Optionally, the initialization information may be stored in a predefined area of the NAND flash memory 130. The initialization information may also be received from any convenient external source. In some cases, a combination sources (predefined, ROM, NAND flash memory 130, and/or external sources) may be utilized to obtain the initialization information for the DMA engine 120.

INTEL-1004, 9:16-29. If the DMA controller knew where to retrieve the boot data to be transmitted—*i.e.*, an address in NVM—it performed the DMA operation automatically and without further involvement by the core processor. *Id.*, 8:61-65 (“when the embedded system is powered on and the DMA engine 120 is initialized, the DMA engine 120 may automatically transfer boot codes 132 and controller codes 134 from the NAND flash memory 130 to buffers 106, 116 in the DRAM device 110”), 9:10-13 (“when the PowerUp_Trigger signal is received, the DMA engine 120 may automatically load the boot and controller codes 132, 134”).

149. A POSITA would have understood that the DMA controller within control circuitry 112 (as modified in the asserted combinations) would have used Post’s pointer metadata as a *DMA controller descriptor* supplying the address of the boot data to be retrieved from NVM 120 and transferred to memory 114. This would

have been consistent with standard operation of the DMA controller as taught by Kao, which used the source address of the boot data to be transferred to perform a DMA transfer operation. Additionally, a POSITA would have found it obvious for the DMA controller of the asserted combinations to use Post's next-page pointer metadata as a "DMA controller descriptor." As I discussed for claim 1[a] (*see* Sec. X.B.2), a POSITA would have been motivated to incorporate a DMA controller that performed DMA transfers of boot data from NVM 120 to memory 114 into Post for a number of compelling reasons—not the least of which would be to offload work involved in performing boot data transfers and process 500 from the processor 112. Kao specifically taught that its DMA controller used the address of an item in NVM as a *descriptor* that enabled identification of the boot data to be transferred. This address data is exactly the information supplied by Post's next-page pointer metadata, and Post described using it for the same purpose, to enable a transfer from NVM to memory 114. A POSITA would have understood Kao as suggesting the use of Post's next-page pointer metadata to allow Kao's DMA controller to perform the transfer of boot data pages from NVM 120 to memory 114.

150. A POSITA would also have understood the DMA controller's use of next-page pointer metadata as a DMA controller descriptor to be the combination of known elements—specifically, the DMA controller described by Kao and Post's next-page pointer addresses—according to the known methods described by Kao, of

using a NVM source address to identify the data to be transferred. And this combination would have achieved predictable results: the DMA controller would have identified the pointed-to page of boot data and transferred that page of boot data to memory 114. Thus, the asserted combinations disclose and render obvious a POR sequencer descriptor that includes *DMA controller descriptors*.

7. Claim 1, Limitation f

wherein the POR sequencer reads and uses the register information to update configuration registers of the embedded system.

151. As I discussed above for claim 1[b] (*see* Sec. X.F.3), in the asserted combinations, wherein Post’s control circuitry 112 is modified to include Wilson’s register file and circuitry to perform the register loading functionality of the RLA engine (“*POR sequencer*”), a POSITA would have understood that the boot data would have included initial register values, as taught by Wilson. For the same reasons, a POSITA would have found it obvious that the POR sequencer of the asserted combinations “*read[] and use[d] the register information to update configuration registers of Post’s SoC 110 (“embedded system”)*” as part of the boot-up process.

152. Wilson taught that loading initialization values into a microprocessor’s registers was typically part of the boot up process. (INTEL-1013, 1:24-28) (“Many microcontrollers require a boot-up process when initially powered up. The boot-up

process allows a microcontroller to perform various actions to prepare the microcontroller for ordinary operation, such as load initialization values in its registers, load instructions in its instruction memory, etc.”). Wilson further taught loading initial register values into the registers of his System on Chip 110 as part of boot up:

Register file 102 can include various registers which are used by the processor 101 in the ordinary operation of the microcontroller. The microcontroller can require that certain specific predefined initial values are stored in one or more of the registers of the register file before the microcontroller assumes ordinary operation. Since the register file is usually composed of volatile memory, the initial values can be stored in the register file as part of an initialization or boot operation. Therefore, these initial values can be a part of boot data that is loaded in the microcontroller during a boot-up procedure.

(INTEL-1013, 2:54-63.)

153. Wilson taught that its RLA engine stored the portions of the boot data that included initial register values in their “final destinations,” meaning the registers in register file 102 corresponding to the initial values. (INTEL-1013, 2:1-3 (“The RLA engine can then decompress the boot data and store various portions of it in their final destinations (such as, for example, in respective registers).”); 6:27-29 (“At step 308, the RLA engine accesses the compressed boot data from the volatile memory, decompresses it and uses it to load the various registers in register file 102.”).)

154. Figure 3 illustrated the steps of a boot up process that included the steps of sending a set of compressed boot data to the microcontroller (step 300), storing the data in nonvolatile memory in the microprocessor (step 304), the RLA's decompressing the data and loading it into the microcontroller's registers (step 308), and activating the microcontroller's processor (step 310):

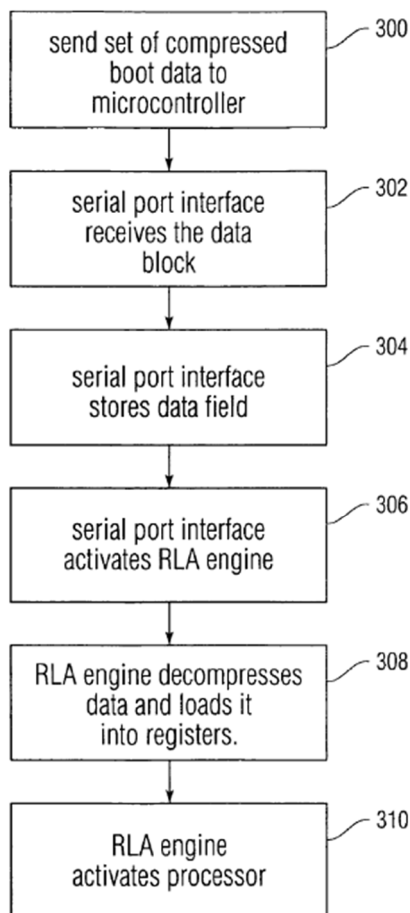


Fig. 3

(*Id.*, Fig. 3.) Additional information regarding the process of Figure 3 is provided in the specification. (*Id.*, 5:31-6:48.)

155. A POSITA would have understood that Wilson's retrieving the compressed initial register data from volatile memory 103 (where it had been stored by serial port interface 105), decompressing that data, and loading it into the microcontroller's registers constituted *read[ing] and us[ing] the register information to update* the registers of control circuitry 112 of the asserted combinations ("*the embedded system*"). A POSITA would have understood that these registers included *configuration registers* because Wilson taught that processors included registers that were "custom designed to perform specific predefined functions," such as configuring the touch screen display and pixel color values of a mobile phone:

In some embodiments the microcontroller is used to control a portable device, such as, for example a mobile phone featuring a multifunction touch screen. Since the microcontroller can be used to power a small device, it can be custom designed to perform specific predefined functions. For example, while most general purpose processors feature a register set, wherein each register is as long as the processor's usual word length (e.g., 32, or 64 bits), a custom designed microcontroller as featured in certain embodiments of the present invention can include a more unorthodox register file.

FIG. 2A shows an exemplary register file according to an embodiment of the present invention. The register file includes two register sets—register set 200 and register set 210. The register file shown in FIG. 2A may correspond, for example, to a microcontroller with a 16-bit processor, i.e., microcontrollers featuring a 16-bit word size. Nevertheless, register set 200 comprises 512 five-bit

registers, while register set 210 comprises 64 nine-bit registers. An embodiment of the present invention can feature a register file comprising of either of the two register sets, a combination of the two register sets, or an entirely different register set.

Thus, the size of the various registers can differ from the processor word size. This is the case because registers of unorthodox sizes may be chosen as best suited for certain specific predefined operations the microcontroller is to perform. For example, the microcontroller can be used to display images on a screen and each five-bit register of the register file 200 can represent a color value for a pixel on a screen.

(*Id.*, 3:35-62.) Wilson described in detail a specific mobile phone embodiment of his microcontroller including a host processor and a touch sensitive screen:

Computing system 700 can also include host processor 728 for receiving outputs from multi-touch panel processor 702 and performing actions based on the outputs that may include, but are not limited to, moving an object such as a cursor or pointer, scrolling or panning, adjusting control settings, opening a file or document, viewing a menu, making a selection, executing instructions, operating a peripheral device connected to the host device, answering a telephone call, placing a telephone call, terminating a telephone call, changing the volume or audio settings, storing information related to telephone communications such as addresses, frequently dialed numbers, received calls, missed calls, logging onto a computer or a computer network, permitting authorized individuals access to restricted areas of the computer or computer network, loading a user profile associated with a user's preferred arrangement of the computer desktop, permitting access to web content, launching a particular program, encrypting or decoding a message, and/or the like. Host processor 728 may also perform additional functions that may not be related to multi-touch panel

processing, and can be coupled to program storage 732 and display device 730 such as an LCD display for providing a user interface (UI) to a user of the device.

Multi touch panel processor 704 can be implemented as part of the microcontroller of the present invention. Random access memory 712 can be part of the same microcontroller as the multi-touch processor.

The above described system can be, for example, a mobile telephone with a touch sensitive screen.

(*Id.*, 11:11-39, Fig. 7.)

156. Wilson's process of *read[ing] and us[ing] the register information to update configuration registers of the embedded system,*" discussed above, is similar to the "register initialization phase" described in the '694 patent in which registers are loaded with initial values:

FIG. 7 illustrates the register initialization phase by the POR sequencer 207. Starting from block 701, the POR sequencer 207 reads the first register information in the register information space 401 of the POR sequencer descriptor 212 from the RAM 208. The POR sequencer 207 then processes the register information and then updates the corresponding configuration register based on the contents of the register information in block 702. Steps 701 and 702 are repeated for succeeding register information until the last register information of the current POR sequencer descriptor 212 is read and processed. After the last register information is read, the POR sequencer 207 writes the DMA controller descriptor 406 to the DMA controller's register space in block 704. The POR sequencer 207 then activates the DMA controller 203 in block 705. The DMA controller 203 then processes the DMA controller's descriptor 406 that it received and initiates transfer of data from the storage

device 209 to the RAM 208 via the storage device controller 204 in block 706. When the data transfer is finished, in block 707, the DMA controller 203 informs the POR sequencer 207 that the data transfer has completed. The POR sequencer 207 then reads the POR sequencer DMA descriptor's control information 404 in block 608. It then checks if the DMA controller descriptor 406 points to a new set of register information of another POR sequencer descriptor fragment 502 in block 709. If the test is positive, the register initialization phase is repeated starting with block 701. If the test is negative, then the register initialization phase has ended.

(INTEL-1001, 7:17-44, Fig. 7).

157. Thus, for the reasons discussed above, a POSITA would have found it obvious that the asserted combinations practice claim 1, limitation f.

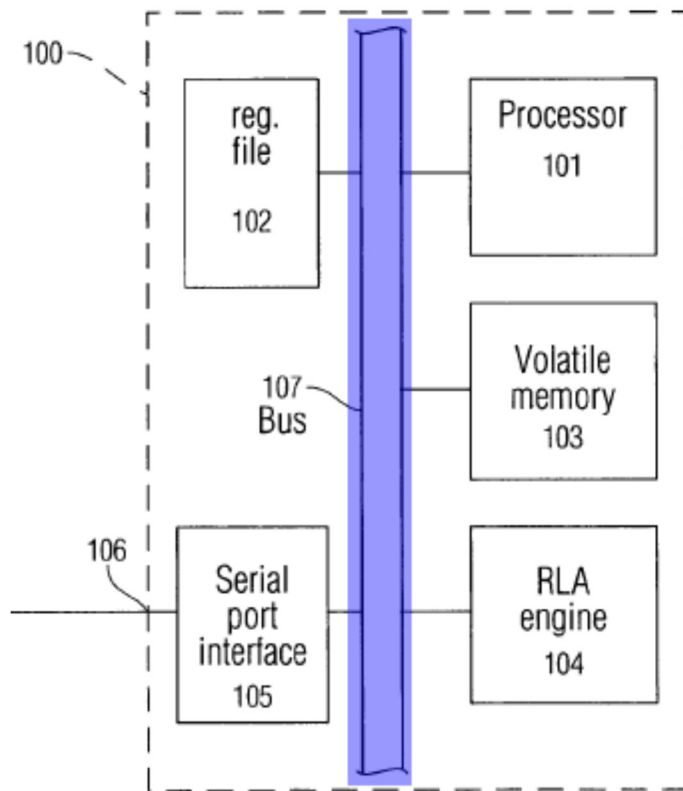
G. Dependent Claim 5

The apparatus of claim 1, further comprising: a CPU bus in the embedded system.

158. Claim 5 of the '694 Patent is obvious in view of Post/Wilson/Kao and Post/Wilson/Kao/So. A POSITA would have understood Post's SoC 110 to include a CPU bus, and at a minimum, this would have been obvious based on Post's disclosure that processor 112 communicated with the various other components of SoC 110. For example, processor 112 interfaced with NVM 118 to load boot data from NVM 112 and it stored the boot data in memory 114. INTEL-1004, 4:3-5 & 3:53-57. A POSITA would have expected SoC 110 to include a CPU bus that allowed the processor 112 to communicate with these other components. INTEL-

1016, 150 (“Buses are simply wires that interconnect the various other components within the CPU.”); INTEL-1008, 8 (“Buses allow for the transfer of electrical signals between different parts of the computer system and thereby transfer information from one device to another.”). At a minimum, a POSITA would have been motivated to include a CPU bus in SoC 110, because processor 112 required structure that enabled this inter-component communication and because a CPU bus was a conventional and nearly-ubiquitous means for achieving this interconnection in an embedded system.

159. Additionally, Wilson disclosed a CPU bus 107: “Bus 107 is used for communications between the various modules of the microcontroller. The bus may allow each module to communicate with each other module according to known protocols.” INTEL-1013, 3:11-14. As shown in Figure 2 of INTEL-1013, bus 107 interconnected processor 101 with the various other components within microcontroller 100:



Id., Fig. 2 (bus 107 highlighted in purple).

160. It would have been obvious to include Wilson’s teaching of using a bus such as bus 107 (a “CPU bus”) into Post’s SoC 110. Wilson specifically motivated this combination through its teaching that bus 107 allowed components within a microcontroller to communicate with each other. INTEL-1013, 3:11-14. Further, Post’s SoC 110 included modules that needed to communicate with and/or control each other: for example, control circuitry 112 read program code from boot ROM 116, interacted with NVM interface 118 to load boot data from NVM 120, and loaded boot data and other information into memory 114. INTEL-1005, Fig. 1, 3:33-65. A POSITA would have recognized the need (and the implicit disclosure) of

structure that enabled these communications and would have been familiar with a bus as a standard technique for transferring electrical signals within embedded systems:

A bus is a physical group of signal lines that have a related function. Buses allow for the transfer of electrical signals between different parts of the computer system and thereby transfer information from one device to another. For example, the data bus is the group of signal lines that carry data between the processor and the various subsystems that comprise the computer.

INTEL-1008, 8. Wilson's microcontroller 100 was similar to Post's SoC 110, including in the sense that both included a number of modules that communicated with each other, and a POSITA would have looked to Wilson's bus 107 as a technique that had worked in one system and would have benefited the other (Post's SoC) in a similar way. Further, because Post describes specific communications between SoC components, but does not describe the structure that facilitates these communications, a POSITA would have looked to Wilson and its bus 107 as supplying implementation details.

H. Independent Claim 6

161. Claim 6 of the '694 Patent is obvious over the combinations of Post/Wilson/Kao and Post/Wilson/Kao/So. Claim 6 recites limitations that are similar to those of claim 1, as noted below, and I incorporate and rely upon my earlier analysis for claims that recite similar subject matter.

1. Claim 6 Preamble

An apparatus, comprising:

162. The preamble of claim 6 is similar to the preamble of claim 1 (“*An apparatus, comprising*”). I incorporate my analysis provided for the preamble of claim 1; the preamble of claim 6 is obvious for the same reasons.

2. Claim 6, Limitation a

an embedded system comprising one or more processors, a reset controller, a storage device controller, one or more direct memory access (DMA) controllers, a random access memory (RAM) and a memory controller, a nonvolatile memory and a nonvolatile memory controller, a debug interface, and

163. Claim 6[a] is similar to claim 1[a] (“*an embedded system comprising one or more processors, a reset controller, a storage device controller, one or more direct memory access (DMA) controllers, a random access memory (RAM) and a memory controller, a nonvolatile memory and a nonvolatile memory controller, a debug interface, and*”). I incorporate my analysis provided for claim 1[a]; claim 6[a] is obvious for the same reasons.

3. Claim 6, Limitation b

a power-on reset (POR) sequencer, wherein the POR sequencer uses a POR sequencer descriptor which is a preassembled descriptor that is stored in the nonvolatile memory,

164. Claim 6[b] is similar to claims 1[b] (“*a power-on reset (POR) sequencer ...*”) and 1[d] (“*... wherein the POR sequencer uses the POR sequencer descriptor which is a preassembled descriptor that is stored in the nonvolatile*

memory”). I incorporate my analysis provided for claims 1[b] and 1[d]; claim 6[b] is obvious for the same reasons.

4. Claim 6, Limitation c

wherein the POR sequencer descriptor includes register information and DMA controller descriptors, and

165. Claim 6[c] is similar to claim 1[e] (“*wherein the POR sequencer descriptor includes register information and DMA controller descriptors, and*”). I incorporate my analysis provided for claim 1[e]; claim 6[c] is obvious for the same reasons.

5. Claim 6, Limitation d

wherein the POR sequencer reads and uses the register information to update configuration registers of the embedded system.

166. Claim 6[d] is similar to claim 1[f] (“*wherein the POR sequencer reads and uses the register information to update configuration registers of the embedded system*”). I incorporate my analysis provided for claim 1[f]; claim 6[d] is obvious for the same reasons.

I. Dependent Claim 9

The apparatus of claim 6, wherein the POR sequencer facilitates a boot process and verifies an integrity of loaded data and code in the RAM.

167. The limitation added by claim 9 is similar to claim 1[c] (“*wherein the POR sequencer facilitates a boot process and verifies an integrity of loaded data*

and code in the RAM”). I incorporate my analysis provided for claim 1[c]; claim 9 is obvious for the same reasons.

J. Independent Claim 12

168. Claim 12 of the '694 Patent is obvious over the combinations of Post/Wilson/Kao and Post/Wilson/Kao/So. Claim 12 recites many limitations that are similar to those of claim 1, as noted below, and I incorporate and rely upon my earlier analysis for claims that recite similar subject matter.

1. Claim 12 Preamble

An article of manufacture, comprising:

169. I do not believe that a person of skill would understand this preamble to be limiting. The first limitation (a) of the claim provides that the claim describes “computer-readable medium having stored thereon instructions” that implement a specified method. The suggestion in the preamble that what follows is “[a]n article of manufacture” does not provide antecedent basis and is not necessary to complete the description of what follows in the rest of the claim.

170. If, nevertheless, the preamble is regarded as a claim limitation, Post’s disclosure embodies this limitation. Post’s electronic device may be one of many manufactured consumer devices including “a portable media player (e.g., an iPod ...), a cellular telephone (e.g., an iPhone ...), a pocket-sized personal computer, a personal digital assistan[t] (“PDA”), a desktop computer, a laptop computer, and any

other suitable type of electronic device or system.” (INTEL-1005, 2:61-67.) Additionally, Post’s electronic device includes control circuitry 112, which “can include any suitable components, circuitry, or logic, such as one or more processors,” (*id.*, 3:34-38), memory 114, boot ROM 116, and non volatile memory 120. (*Id.*, 3:1-2, 3:33-34.) A person of skill would recognize that each of these are manufactured items.

2. Claim 12, Limitation a

a non-transitory computer-readable medium having stored thereon instructions operable to permit an apparatus to perform a method comprising:

171. At boot-up, the control circuitry in Post’s electronic device executes program code stored in the boot ROM to search for, identify and retrieve boot code from the non volatile memory. INTEL-1005, 1:37-43 (“A NVM provided in an electronic device may include a boot data storage area in which boot data is stored. ‘Boot data’ may refer to any information, such as a bootloader (e.g., a first stage bootloader), that is loaded from the NVM based on code stored in a boot ROM (e.g., a mask ROM).”), 3:53-57 (“Boot ROM 116 may be configured to store program code that is executed by control circuitry 112 on bootup of electronic device 100. The program code may be executed to begin loading additional information from NVM 120 into memory 114.”). The control circuitry includes one or more processors (*id.*, 3:36-38), and a boot ROM is a computer-readable medium, as it

provides program code to the control circuitry processors. *Id.*, 3:41-43 (“At bootup of electronic device 100, control circuitry 112 may operate under the control of program code stored in boot ROM 116.”).

172. The manner in which the program code stored in the boot ROM is used by Post’s control circuitry to perform the methods of the remainder of this claim is discussed in detail in each element below. Generally, the boot ROM implements the procedures for searching, identifying and retrieving boot data from the non volatile memory (NVM) in accordance with the procedures illustrated in Post’s Figures 4 and 5. *Id.*, 8:7-15 (“Referring now to FIGS. 4-6, flowcharts of illustrative processes are shown in accordance with various embodiments of the invention. The steps of these illustrative processes may be performed by any suitable component ... such as by control circuitry 112 of FIG. 1. ... [T]hese components may operate under the control of mask ROM code (e.g., from boot ROM 116) or from an operating system and NVM driver.”).

173. Combining Post with Kao, with the result that the DMA engine for copying boot data from an NVM disclosed in Kao is incorporated into Post’s electronic device, does not change the resulting system’s reliance on the program code stored in the boot ROM to implement the procedures recited in claims 19 and 20. Kao describes that the DMA engine operates in accordance with data transfer parameters provided to it. (INTEL-1004, 8:15-31, 9:16-33 (parameters for transfer

of data may come from a variety of sources).) Post's method requires specific pages of data to be retrieved from the NVM in accordance with the procedures illustrated in Post's Figures 4 and 5 or in accordance with a process by which pages are selectively read so that non-data pages are skipped over. (INTEL-1005, 10:48-49.) It would be obvious to a person of skill, therefore, that the DMA engine parameters for retrieving the necessary pages must be supplied by the control circuitry that manages the boot code search and retrieval process, as directed by the code contained in the boot ROM.

3. Claim 12, Limitation b

detecting in a nonvolatile memory in the apparatus, by a power-on reset (POR) sequencer, a descriptor used in a boot process and copying, by the POR sequencer, the descriptor from the nonvolatile memory to a random access memory (RAM), and

174. Claim 12[b] is similar to claim 1[b] (“a power-on reset (POR) sequencer configured to detect, in the nonvolatile memory, a descriptor used in a boot process and configured to copy the descriptor from the nonvolatile memory to the RAM”). I incorporate my analysis provided for claim 1[b]; claim 12[b] is obvious for the same reasons.

4. Claim 12, Limitation c

facilitating, by the POR sequencer, a boot process and verifying, by the POR sequencer, an integrity of loaded data and code in the RAM,

175. Claim 12[c] is similar to claim 1[c] (“*wherein the POR sequencer facilitates a boot process and verifies an integrity of loaded data and code in the RAM*”). I incorporate my analysis provided for claim 1[c]; claim 12[c] is obvious for the same reasons.

5. Claim 12, Limitation d

wherein the descriptor comprises a POR sequencer descriptor, and wherein the POR sequencer uses the POR sequencer descriptor which is a preassembled descriptor that is stored in the nonvolatile memory,

176. Claim 12[d] is similar to claim 1[d] (“*wherein the descriptor comprises a POR sequencer descriptor, and wherein the POR sequencer uses the POR sequencer descriptor which is a preassembled descriptor that is stored in the nonvolatile memory*”). I incorporate my analysis provided for claim 1[d]; claim 12[d] is obvious for the same reasons.

6. Claim 12, Limitation e

wherein the POR sequencer descriptor includes register information and DMA controller descriptors, and

177. Claim 12[e] is similar to claim 1[e] (“*wherein the POR sequencer descriptor includes register information and DMA controller descriptors, and*”). I

incorporate my analysis provided for claim 1[e]; claim 12[e] is obvious for the same reasons.

7. Claim 12, Limitation f

wherein the POR sequencer reads and uses the register information to update configuration registers of the embedded system.

178. Claim 12[f] is similar to claim 1[f] (“*wherein the POR sequencer reads and uses the register information to update configuration registers of the embedded system*”). I incorporate my analysis provided for claim 1[f]; claim 12[f] is obvious for the same reasons.

XI. CONCLUSION

179. In signing this declaration, I recognize that the declaration will be filed as evidence in a contested case before the Patent Trial and Appeal Board of the United States Patent and Trademark Office. I also recognize that I may be subject to cross-examination in the case and that cross-examination will take place within the United States. If cross-examination is required of me, I will appear for cross-examination within the United States during the time allotted for cross-examination.

180. I hereby declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful

false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

Executed this 31st day of March 2023 in Los Altos, California.

Respectfully submitted,

A handwritten signature in black ink, appearing to read "Donald Alpert", with a long horizontal flourish extending to the right.

Donald Alpert, Ph.D.

APPENDIX A



Donald Alpert, Ph.D.
Camelback Computer Architecture, LLC
2310 Homestead Road
Suite C1 #411
Los Altos CA 94024
(650) 386-5301

Donald Alpert, Ph.D., is an independent consultant, specializing in microprocessors and computer systems. For more than a decade in his role as president of Camelback Computer Architecture LLC, Dr. Alpert has provided clients with expert services for patent litigation, technology assessment, and product development. He has over 40 years of wide-ranging academic and industry experience that enable him to provide expert consulting services in the following areas:

- Computers, Computer Systems
- Computer architecture
- Microprocessors
- Semiconductors
- PC Systems
- Embedded Systems
- Design Validation and Debug
- Performance Evaluation
- Cache Memory
- Instruction Sets
- Media Processing
- Power Management

Patent Analysis: Dr. Alpert has served as an expert on dozens of patent litigation cases, handling both infringement and invalidity issues. Dr. Alpert has worked as both a consulting expert and testifying expert assisting attorneys representing major high-tech corporations. He has substantial experience testifying in depositions and at trials before U.S. Federal Courts and the International Trade Commission. Dr. Alpert also consults to assess patents for acquisition and for pre-litigation review. See [here](#) for a summary of testifying experience.

Product Development: Dr. Alpert is a widely recognized innovator with over 30 patents. Dr. Alpert has worked at some of the most ground-breaking technology firms, including Zilog, National Semiconductor, and Intel, where he was the principal architect of the Pentium® Processor. See [here](#) for more information and example projects.

Employment Experience

Camelback Computer Architecture, LLC, Sacramento, CA

Consultant

2/02-Present

Computer architecture consulting services to assess technology, develop products, and protect IP.

Intel Corporation, Chandler, Arizona

Principal Engineer

12/99-1/02

Led architecture team of 10 responsible for definition and performance evaluation of Itanium™ Processor Family microprocessors.

Principal Engineer

5/98-12/99

Led architecture team of 4 responsible for technical strategy and product definition of PC graphics processors and chipsets.

Stanford University, Stanford, California

Consulting Assistant Professor

9/97-4/98

Responsible for teaching undergraduate and graduate classes in computer architecture and participating in processor-design research projects.

Intel Corporation, Santa Clara, California

Engineering Manager

10/93-6/97

Responsible for microarchitecture definition and architecture validation of the Itanium™ microprocessor, which was the first implementation of Intel's 64-bit extension architecture. Led a team of 60 engineers.

Architecture Manager

1/89-10/93

Responsible for the Pentium® Processor technical definition. Worked with customers and Intel marketing to identify product requirements and with the design engineering team to evaluate design options. Led a team of seven engineers responsible for creating technical specifications and for modeling and simulating performance using traces from DOS, Windows, and UNIX execution. After completing the Pentium Processor, led an interdisciplinary team of 30 engineers to define a new 64-bit instruction set architecture.

National Semiconductor Corporation, Herzlia, Israel

Architecture Manager

5/85-1/89

Responsible for defining new microprocessors in the Series 32000 family, including the NS32532 and "Swordfish" high-performance microprocessors. Led a team of five engineers working in the following areas: modeling and simulating high-performance microprocessors, evaluating novel computer architectures, and writing technical specifications for new products.

Zilog, Incorporated, Campbell, California

Component Architect

6/80-4/85

Worked on the Z80,000 CPU, a 32-bit microprocessor that integrated an instruction pipeline, cache, and memory management unit. Responsible for specifying the non-privileged instruction set, evaluating performance, and writing the reference manual for the microprocessor.

Burroughs Corporation, Pasadena, California

Associate Engineer

7/76-7/77

Designed computer peripheral interface controllers using MSI-TTL and 8080 microprocessors.

Alpx Computer Corporation, Danbury, Connecticut

8/75-6/76

Consulting engineer working on design of video games.

Action, Washington, DC

6/73-6/75

Peace Corps volunteer teaching mathematics and physics in a secondary school in Ghana, West Africa.

Donald Alpert Camelback Computer Architecture, LLC (650) 386-5301

Education

Stanford University 9/77-6/84
Graduate study in computer design. National Science Foundation Graduate Fellowship.
Awarded MSEE in 1978 and PhDEE in 1984.

M.I.T. 9/69-6/73
Member of Tau Beta Pi and Eta Kappa Nu. Awarded BSEE in 1973.

Professional Activities

Member of Microprocessor Report editorial board (2002-present)

Anthem College Commencement Address (18-Dec-2009)

Panel session about history of microprocessor development at National Semiconductor Corporation, Computer History Museum (2008).

[\[http://archive.computerhistory.org/resources/access/text/Oral_History/102658246.05.01.acc.pdf\]](http://archive.computerhistory.org/resources/access/text/Oral_History/102658246.05.01.acc.pdf)

IEEE Technical Committee on Microprocessors and Microcomputers, Chair (1999-2000) and Vice-Chair (2001-present).

Lectures about the Itanium™ Processor Family at U.C. Irvine (1999, 2000).

Cool Chips I, Keynote speaker (Tokyo, 1998)

IEEE Hot Chips Symposium Program Co-chair (1994), member of program committee (1993, 1995).

Lectures about the Pentium® Processor at Stanford University, University of Illinois, and Cornell (1993-1994).

Appeared in Stanford UVC Educational Video production about the Pentium Processor (1993).

IEEE International Symposium on Computer Architecture member of program committee (1996) and Session Chair for Cache Design (1989).

Teaching Experience

Faculty Associate at Arizona State University teaching undergraduate class on assembly language programming (Spring/2001) and computer architecture (Spring/2002).

Consulting Associate Professor at Stanford University teaching undergraduate and graduate classes in computer architecture (Fall/1997, Winter/1998).

Consulting Professor at Tel Aviv University teaching graduate class in computer architecture (Spring/1987)

Peace Corps volunteer in a secondary school in Ghana, West Africa teaching mathematics and physics (1973-1975).

Don Alpert, "How Microprocessors Upset the Computer Industry," Microprocessor Report, December 11, 2006.

Don Alpert, "Evolving PC Chip-Set Topologies," Microprocessor Report, April 17, 2006.

Don Alpert, "Will Microprocessors Become Simpler?" Microprocessor Report, November 17, 2003.

Don Alpert, "Itanium Processor Status Report," Microprocessor Report, July 28, 2003.

Don Alpert, "Scalable MicroSupercomputers," Microprocessor Report, March 17, 2003.

Don Alpert, "ColdFire Goes Fully Superscalar," Microprocessor Report, October 28, 2002.

Donald Alpert and Alan Jay Smith, "Guest Editor Introduction," IEEE Micro special issue on Hot Chips VI, April 1995.

Donald Alpert and Dror Avnon, "Architecture of the Pentium Processor," IEEE Micro, June 1993, pp. 11-21. Selected best paper in IEEE Micro for 1993.

D. Alpert, A. Averbuch, and O. Danielli, "Performance Comparison of Load/Store and Symmetric Instruction Set Architectures," Proceedings, 18th International Symposium on Computer Architecture, June 1990, pp. 172-181.

Benjamin Maytal, Sorin Iacobovici, Donald Alpert, et al, "Design Considerations for a General-Purpose Microprocessor," Computer, January 1989, pp. 66-76.

Donald Alpert and Michael Flynn, "Performance Trade-offs for Microprocessor Cache Memories," IEEE Micro, August 1988, pp. 44-54.

D. Alpert, J. Levy, and B. Maytal, "Architecture of the NS32532 Microprocessor," Proceedings, 1987 IEEE International Conference on Computer Design, October 1987, pp. 168-172. Awarded best presenter at Design and Test sessions.

D. Alpert, D. Biran, L. Epstein, et al, "Trends in VLSI Microprocessor Design," Proceedings, First Annual Conference on Computer Technology, Systems and Applications (CompEuro '87), May 1987, pp. 564-567.

Y Sidi, D. Alpert, D. Biran, et al, "Design Considerations of an Advanced 32-Bit Microprocessor," Proceedings, 15th Conference of Electrical and Electronic Engineers in Israel, April 1987.

D. Alpert, "Trends in VLSI Microprocessor Design," Proceedings, 15th Conference of Electrical and Electronic Engineers in Israel, April 1987.

Donald Alpert, Michael J. Flynn, and Scott Wakefield, "Directly Executed Languages for VLSI Processor Design," Proceedings, 1983 IEEE International Conference on Computer Design, October 1983, pp. 609-612.

Donald Alpert, "Powerful 32-Bit Micro Includes Memory Management," Computer Design, October 1983, pp. 213-220.

Don Alpert, Dean Carberry, Mike Yamamura, et al, "32-Bit Processor Chip Integrates Major System Functions," Electronics, July 14, 1983, pp. 113-119.

Patents

4,802,085, Apparatus and method for detecting and handling memory-mapped I/O by a pipelined microprocessor

5,249,286, Selectively locking memory locations within a microprocessor's on-chip cache

5,263,153, Monitoring control flow in a microprocessor

5,416,913, Method and apparatus for dependency checking in a multi-pipelined microprocessor

5,442,756, Branch prediction and resolution apparatus for a superscalar computer processor

5,475,824, Microprocessor with apparatus for parallel execution of instructions

5,479,652, Microprocessor with an external command mode for diagnosis and debugging

5,481,751, Apparatus and method for storing partially-decoded instructions in the instruction cache of a CPU having multiple execution units

5,559,986, Interleaved cache for multiple accesses per clock cycle in a microprocessor

5,606,676, Branch prediction and resolution apparatus for a superscalar computer processor

5,617,554, Physical address size selection and page size selection in an address translator

5,621,886, Method and apparatus for providing efficient software debugging

5,638,525, Processor capable of executing programs that contain RISC and CISC instructions

5,657,253, Apparatus for monitoring the performance of a microprocessor

5,659,679, Method and apparatus for providing breakpoints on taken jumps and for providing software profiling in a computer system

5,669,011, Partially decoded instruction cache

5,671,435, Technique for software to identify features implemented in a processor

5,675,825, Apparatus and method for identifying a computer microprocessor

5,692,167, Method for verifying the correct processing of pipelined instructions including branch instructions and self-modifying code in a microprocessor

5,729,724, Adaptive 128-bit floating point load and store operations for quadruple precision compatibility

5,740,413, Method and apparatus for providing address breakpoints, branch breakpoints, and single stepping

5,764,959, Adaptive 128-bit floating point load and store instructions for quad-precision compatibility

5,774,686, Method and apparatus for providing two system architectures in a processor

5,790,834, Apparatus and method using an ID instruction to identify a computer microprocessor

5,802,605, Physical address selection and page size selection in an address translator

5,958,037, Apparatus and method for identifying features and the origin of a computer microprocessor

5,991,874, Conditional move using a compare instruction generating a condition field

6,052,801, Method and apparatus for providing breakpoints on a selectable address range

6,219,774, Address translation with/bypassing intermediate segmentation translation to accommodate two different instruction set architecture

6,385,718, Computer system and method for executing interrupt instructions in operating modes

6,408,386, Method and apparatus for providing event handling functionality in a computer system

6,584,558, Article for providing event handling functionality in a processor supporting different instruction sets

7,010,671, Computer system and method for executing interrupt instructions in two operating modes

7,389,403, Adaptive computing ensemble microprocessor architecture

7,802,073, Virtual core management

PARTIES	DISTRICT	DIVISION	CASE NUMBER	TESTIMONY
Intergraph Hardware Technologies Company vs. Hewlett-Packard Company	Eastern District, TX	Marshall	2-02CV-312-TJW	Deposition 4/2004
MediaTek Inc. vs. Sunext Technology Co., Ltd.	International Trade Commission		Investigation No. 337-TA-523	Deposition 4/2005 Hearing 7/2005
Asyst Technologies, Inc. vs. Jenoptik AG	Northern District, CA	San Jose	98-20451-JF	Deposition 8/2006 Trial 1/2007
Cornell University vs. Hewlett-Packard Company	Northern District, NY	Syracuse	5:01-CV-1974	Deposition 9/2006 Trial 5/2008 Trial 7/2008
LG Electronics, Inc. vs. Quanta Computer Inc.,	Northern District, CA	Oakland	C 01-0287 CW	Deposition 7/2007
Network Appliance, Inc. vs. Sun Microsystems Inc.	Northern District, CA	San Francisco	C-07-06053-EDL	Markman 8/2008
HTC Corp. vs. Apple Inc.	International Trade Commission		Investigation No. 337-TA-721	Markman 10/2010 Deposition 3/2011 Hearing 5/2011
HTC Corp. vs. Apple Inc.	International Trade Commission		Investigation No. 337-TA-808	Markman 4/2012 Deposition 7/2012
Applied Micro Circuits Corp. vs. Sandforce, Inc.	Superior Court of California	Santa Clara County	Case No. 110CV190188	Deposition 3/2013
Intellectual Ventures vs. Motorola Mobility	District of Delaware		Case No. 1:11-cv-00908-SLR	Deposition 6/2013 Trial 1/2014 Retrial 3/2015
Maxim Integrated Products vs. Starbucks, et al.	Western District of Pennsylvania	Pittsburgh	Multi-District Litigation No. 2354	Deposition 7/2013 Deposition 12/2014
NXP vs. RIM	Middle District Of Florida	Orlando	Case No. 6:12-cv-00498-ACC-TBS	Deposition 9/2013 Trial 3-4/2014
Sandisk vs. Netlist	Patent Trial and Appeal Board		IPR2014-00970 IPR2014-00971	Deposition 2/2015 Deposition 6/2015
PLL Technologies vs. Xilinx	Patent Trial and Appeal Board		IPR2015-00148	Deposition 8/2015 Deposition 12/2015

Donald Alpert - Expert Witness Testimony

PARTIES	DISTRICT	DIVISION	CASE NUMBER	TESTIMONY
PUMA vs. Apple	Eastern District of Texas	Marshall	Case No.2:15-cv-00621	Deposition 8/2016
Intellectual Ventures vs. EMC	Patent Trial and Appeal Board		IPR2016-01106	Deposition 5/2017
SK Hynix vs. Netlist	Patent Trial and Appeal Board		IPR2017-00548	Deposition 8/2017
Intellisoft vs. Acer	Superior Court of California	Santa Clara County	Case No. 1:14-CV-272381	Deposition 10/2017
Visual Effect Innovations vs. NVIDIA	Northern District, CA	San Francisco	Case No. 3:17-cv-03187-VC	Deposition 12/2017
Qualcomm vs. Apple	International Trade Commission		Investigation No. 337-TA-1093	Deposition 7/2018 Hearing 9/2018
SK Hynix vs. Netlist	Patent Trial and Appeal Board		IPR2018-00303	Deposition 7/2018
TecSec vs. Cisco	Eastern District, VA	Alexandria	Case No. 1:10-cv-00115-LO-TCB	Deposition 9/2020
SK Hynix vs. Netlist	Patent Trial and Appeal Board		IPR2020-01044 IPR2020-01042	Deposition 3/2021
Microchip vs. HD Silicon Solutions	Patent Trial and Appeal Board		IPR2021-01567	Deposition 6/2022
Acer vs. ACQIS	Eastern District of Texas	Marshall	Case No. 2:21-cv-00275-JRG-RSP	Deposition 7/2022
Samsung vs. Netlist	Patent Trial and Appeal Board		IPR2022-00062 IPR2022-00064	Deposition 7/2022
Micron vs. Netlist	Patent Trial and Appeal Board		IPR2022-00263	Deposition 9/2022
Intel vs. BiTMICRO	Western District of Texas	Waco	Case No. 6:22-cv-00335	Deposition 11/2022

APPENDIX B

APPENDIX B – U.S. PATENT NO. 10,120,694 CHALLENGED CLAIM LIMITATIONS

Claim Limitation	'694 Patent Claim Language
1[pre]	An apparatus, comprising:
1[a]	an embedded system comprising one or more processors, a reset controller, a storage device controller, one or more direct memory access (DMA) controllers, a random access memory (RAM) and a memory controller, a nonvolatile memory and a nonvolatile memory controller, a debug interface, and
1[b]	a power-on reset (POR) sequencer configured to detect, in the nonvolatile memory, a descriptor used in a boot process and configured to copy the descriptor from the nonvolatile memory to the RAM,
1[c]	wherein the POR sequencer facilitates a boot process and verifies an integrity of loaded data and code in the RAM,
1[d]	wherein the descriptor comprises a POR sequencer descriptor, and wherein the POR sequencer uses the POR sequencer descriptor which is a preassembled descriptor that is stored in the nonvolatile memory,
1[e]	wherein the POR sequencer descriptor includes register information and DMA controller descriptors, and
1[f]	wherein the POR sequencer reads and uses the register information to update configuration registers of the embedded system.
5	The apparatus of claim 1, further comprising: a CPU bus in the embedded system.
6[pre]	An apparatus, comprising:
6[a]	an embedded system comprising one or more processors, a reset controller, a storage device controller, one or more direct memory access (DMA) controllers, a random access memory (RAM) and a memory controller, a nonvolatile memory and a nonvolatile memory controller, a debug interface, and
6[b]	a power-on reset (POR) sequencer, wherein the POR sequencer uses a POR sequencer descriptor which is a preassembled descriptor that is stored in the nonvolatile memory,
6[c]	wherein the POR sequencer descriptor includes register information and DMA controller descriptors, and

Claim Limitation	'694 Patent Claim Language
6[d]	wherein the POR sequencer reads and uses the register information to update configuration registers of the embedded system.
9	The apparatus of claim 6, wherein the POR sequencer facilitates a boot process and verifies an integrity of loaded data and code in the RAM.
12[pre]	An article of manufacture, comprising:
12[a]	a non-transitory computer-readable medium having stored thereon instructions to permit an apparatus to perform a method comprising:
12[b]	detecting in a nonvolatile memory in the apparatus, by a power-on reset (POR) sequencer, a descriptor used in a boot process and copying, by the POR sequencer, the descriptor from the nonvolatile memory to a random access memory (RAM), and
12[c]	facilitating, by the POR sequencer, a boot process and verifying, by the POR sequencer, an integrity of loaded data and code in the RAM,
12[d]	wherein the descriptor comprises a POR sequencer descriptor, and wherein the POR sequencer uses the POR sequencer descriptor which is a preassembled descriptor that is stored in the nonvolatile memory,
12[e]	wherein the POR sequencer descriptor includes register information and DMA controller descriptors, and
12[f]	wherein the POR sequencer reads and uses the register information to update configuration registers of the embedded system.