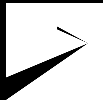
# Adventures in Reverse Engineering Broadcom NIC Firmware

## Unlocking servers with 100% open source firmware

Hugo Landau

2023-12-27 37C3



# Project Ortega: Motivations

# How to become an activist at age 10

- When I was 10 years old, I heard the soon-to-be-released Windows XP would have a “feature” called “product activation”
- I had never heard of FSF, open source, etc. or even used the internet, but I intuitively knew this was wrong and boycotted Windows XP
- Why? Because I intuitively knew that your computer should be on your side — not anyone else’s

# Your computer should be on your side

. . . but is it?

- No
- Product activation
- DRM
- Locked bootloaders
- Tivoisation
- Your smartphone is controlled by Apple/Google/$VENDOR whether or not you want it
- **Antifeatures:** Designed to benefit the manufacturer's interests over yours

# Beyond just computers

- Now anything with a chip might be used to try and control you
- Companies have figured out they can use software to control how a product is used after they sell it
  - Technological “workaround” to the first sale principle
- Printers, CNC machines, cars, tractors, you name it... even trains
  - e.g. preventing repair or use of third party components
  - Real examples exist for all of these, and countless others
  - 37C3: Breaking “DRM” in Polish trains (today, 11pm)
- References:
  - “The Coming War on General-Purpose Computation” (Cory Doctorow, 28C3)

# The principle of owner control

A simple principle must be defended:

- All hardware and software must be designed to put the interests of its owner first, over any vendor, any third party, any government
- Owner control: the diametrical opposite of DRM

# Making owner control a reality

- All of the software and firmware on your machine should be yours to audit, inspect, and change to your needs
- Ergo: All of the firmware on your machine must be open source
- Antifeatures are largely impossible under these conditions
  - If you don't like a feature, you can just remove it
  - Ability to impose antifeatures is a big motivation to keep firmware proprietary
  - "Open source DRM" is an inherently nonsensical concept
- If a vendor doesn't want to open something because then the user could remove feature X from it, that feature is an antifeature by definition

# Open source firmware and security

- Open source firmware isn't just about owner control
- Need to be able to audit firmware to trust it
  - Potential for backdoors is immense
  - Or just zero-days in shoddy vendor code
- Increasing concern about supply-chain attacks
  - *"This firmware is signed by $VENDOR, so it's safe"* is not a good security model
  - Reproducible builds help mitigate backdoors inserted into compiled code
    - "Trusting trust" attacks
    - Open source is a requirement for this

# x86 prevents owner control

- Both Intel and AMD have signed firmware blobs which can't be replaced
  - Used for DRM and other functions
  - Fully open source firmware for x86 is now impossible
- Many SBCs have open firmware, but they're not fast
- Where can we get a desktop or server with open firmware?
  - IBM makes fast server CPUs (POWER9)
  - Amazingly, they agreed to open source all the firmware
- Talos II: EATX motherboard, POWER9 CPU
  - 99% open firmware
  - But the Broadcom Ethernet controller has a firmware blob
  - Let's reverse the firmware and get rid of it!

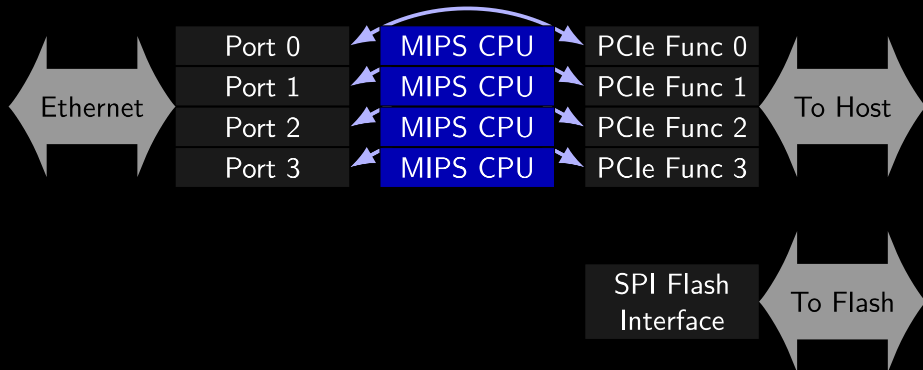
# Device overview

# Broadcom BCM5719 Ethernet Controller

- Quad-port PCIe Gigabit Ethernet Controller
  - 13th Generation chip descending from the Tigon line of NICs released by Alteon Networks in 1997
  - These chips have a **long** history — and the IP somehow ended up with Broadcom
- Has special features oriented to server applications
- The BCM5719 supports NC-SI, a standard allowing a server BMC to share the host's network connection, piggybacking on the host's Ethernet ports
  - Network Controller — Sideband Interface
  - If you've used a server where you can access the host and the BMC over the same port, this is how it works
- Modern servers have a Baseboard Management Controller (BMC), a SoC implementing remote management features (IPMI, Serial over LAN, etc.)
  - Yes, it usually runs Linux
- It needs a network connection
  - Using a separate port can be wasteful
  - NC-SI allows it to share the host's ports

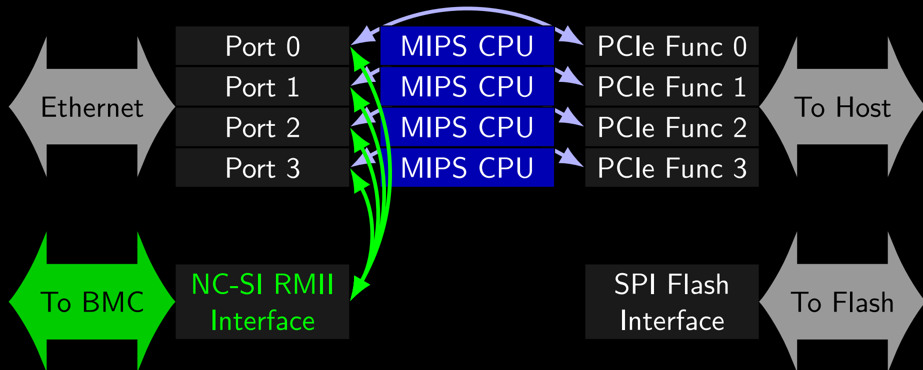
# Broadcom BCM5719 Ethernet Controller Block Diagram

- 4-port PCIe Gigabit Ethernet controller
  - PCIe on one side, Gigabit Ethernet on the other



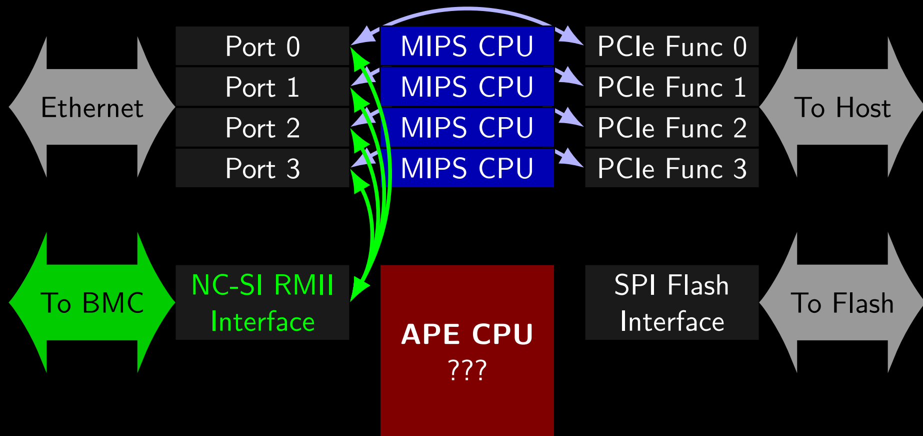
# Broadcom BCM5719 Ethernet Controller Block Diagram

- NC-SI lets server BMC (IPMI) piggyback on host network connection
  - Oriented towards server use



# Broadcom BCM5719 Ethernet Controller Block Diagram

- NC-SI lets server BMC (IPMI) piggyback on host network connection
  - Oriented towards server use



# Recon: Examining the flash layout

| |
|---|
| Header |
| Image Pointers |
| Configuration, MAC Addrs |
| Stage 1 MIPS FW |
| Stage 2 MIPS FW |
| PXE Option ROM (x86 legacy BIOS) |
| PXE Option ROM (x86-64 UEFI BIOS) |
| Extended Image Pointers |

| |
|---|
| Config Management Tool |
| APE Code (NC-SI) |
| iSCSI Boot Code |
| iSCSI config for each port |
| Extended VPD |
| (free space) |

We don't care about... most of this, actually!

# Recon: Examining the flash layout

| Flash layout |
| --- |
| Header |
| Image Pointers |
| Configuration, MAC Addrs |
| Stage 1 MIPS FW |
| Stage 2 MIPS FW |
| PXE Option ROM (x86 legacy BIOS) |
| PXE Option ROM (x86-64 UEFI BIOS) |
| Extended Image Pointers |
| Config Management Tool |
| APE Code (NC-SI) |
| iSCSI Boot Code |
| iSCSI config for each port |
| Extended VPD |
| (free space) |

We don't care about... most of this, actually!

# MIPS

- Each port has an ancient MIPS core — roughly MIPS III, no hardware MUL/DIV
- The firmware for these was fully reversed, but these cores turn out to be **almost vestigial**
  - Originally the MIPS cores handled dataflow, two cores per port (one for RX, one for TX)
  - Then dataflow was moved into hardware, and only one core was kept
  - Now these cores are left with almost nothing to do
- What do these cores still do?
  - Device Init: Loading MAC addresses from flash, setting device and Ethernet PHY registers
  - Even a lot of this init code turns out not to be enabled in practice and is vestigial
- After initialization, the MIPS core for each port enters an infinite loop checking if housekeeping tasks need to be performed

# MIPS: What does it actually do?

What housekeeping tasks? Well. . .

```
for (;;) {
  S2MainLoop_Init1(&init);  // Doesn't actually do anything
  S2ConfigureAPE();         // Almost never does anything

  if (GetReg(REG_STATUS) & REG_STATUS__VMAIN_POWER_STATUS) {
    S2VPDAttentionCheck();  // Is the host asking for my serial number? No?
    continue;               // Time to check again!
  }
  ...
```

After initialization, if the host is on, each port has an entire core which spends its **entire life** looping checking if the host is trying to request its serial number!

- These cores implement random dregs of functionality nobody has yet bothered to move elsewhere

# Your princess is in another castle

- The MIPS cores do some device initialisation, but are otherwise largely unused
- There is a mysterious "APE" block on the device
  - Is it a core? Is it something else? Broadcom doesn't say
  - There is an "APE Code" image on the flash
  - There are references to "NCSI" in APE-related code
- Presumably, the APE is some core which implements NC-SI
- We need to reverse engineer the APE code image and figure out how it works if we want to have working NC-SI
- However, the APE image is compressed with some unknown algorithm
- APE memory space isn't accessible from PCIe or MIPS cores either
- How do we get in to this thing?

# How I ended up reverse engineering x86 real mode decompression code

# Two problems

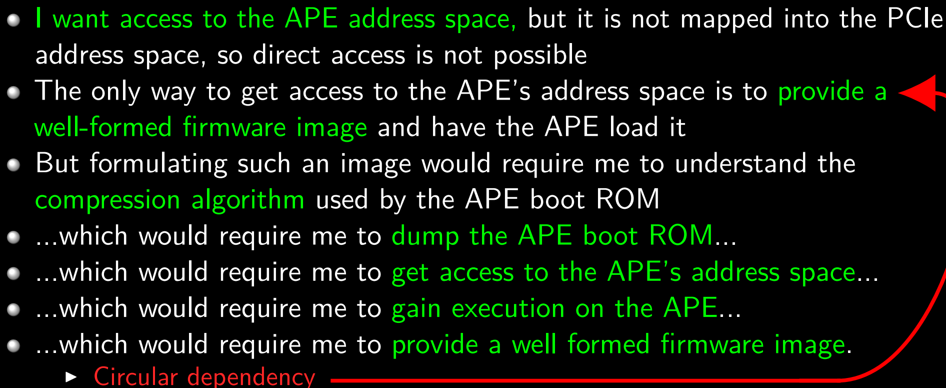
**Problem 1**

- I want to disassemble the APE code
- But the APE code is compressed
- It doesn't appear to be any common or recognisable compression algorithm
  - ...although the number of partial text strings in the compressed image suggests it's quite a crappy algorithm

**Problem 2**

- I want to gain access to the APE memory space
- But it's not mapped into PCIe or accessible indirectly
- It can only be accessed by the APE core
- Only way to gain execution on the APE core is by providing a valid firmware image

# Two problems: a conundrum



- I want access to the APE address space, but it is not mapped into the PCIe address space, so direct access is not possible
- The only way to get access to the APE's address space is to provide a well-formed firmware image and have the APE load it
- But formulating such an image would require me to understand the compression algorithm used by the APE boot ROM
- ...which would require me to dump the APE boot ROM...
- ...which would require me to get access to the APE's address space...
- ...which would require me to gain execution on the APE...
- ...which would require me to provide a well formed firmware image.
    - Circular dependency

# Problem: Decompression

- We need to decompress the APE image in order to study it
- We don't have the decompression code
  - The boot ROM for the APE must have it
  - But we can't seem to find any way into the APE's memory space
  - Circular dependency
- Let's take a break and look somewhere completely different
- The SPI flash also contains things like PCI Option ROM images
  - PXE Boot support, config menu support, etc.
  - These images look like they are compressed with a suspiciously similar algorithm
  - Since they are Option ROMs they must be self-decompressing
  - They must contain the decompression code!

# PCI Option ROMs

- There are two PXE images:
  - PC BIOS (x86 real mode code, oh my. . . )
  - UEFI BIOS (nice, modern, x64 protected mode code)
- Obviously I'd rather reverse the latter

**However,**

- The UEFI standard specifies its own standard UEFI compression algorithm
- The UEFI PXE option ROM is compressed with this, not the APE compression algorithm
- Meaning, the only specimen for the decompression code I'm looking for is in the x86 real mode PC BIOS image
  - . . . oh no

# Reversing x86 real mode code

- Putting it shortly: not fun
- Decompilers? Forget it
- Segment registers are constantly being changed and make everything confusing to follow

```
mov  al, es:[bx]
```

- Decompression code is always hairy as it is, but trying to “eyeball” this code proved unpleasant and impossible

# Reversing x86 real mode code

- Segment registers are constantly being changed and make everything confusing to follow
- Consider a simple load:

```
mov  al, es:[bx]
```

- Memory address is determined by segment register es **and** general register bx
- bx is 16 bits so it can't address enough memory
- ...so the compiler Broadcom uses to generate code which constantly changes the value of es as needed to address different regions of memory
- This makes following the code very confusing
- Imagine debugging protected mode code if the memory mappings kept changing

# Reversing x86 real mode code

- Ad-hoc reversing of the decompression code was horrible, but nonetheless attempted
- The code seemed to work at first to decompress the APE firmware for the first few instructions, but proved to have bugs which corrupted code subtly — $!~“@!
  - Explained a lot of baffling disassembler output
- Needed a way of reversing this code which **can't introduce bugs**
  - Otherwise, corruption which isn't immediately obvious may frustrate reversing and cause wild goose chases down the line
  - Reversing decompression algorithms is annoying enough without it being real mode

# A methodical approach

- Idea: Construct an "x86 real mode emulator" in C
  - "Emulator" is too big a word
- Every line of x86 real mode assembly was turned into a C comment

```
//                  mov     es, [bp+var_4]
//                  mov     bx, si
//                  inc     si
//                  mov     al, es:[bx]
```

# The x86 real mode "emulator"

- An "emulation" environment for real mode code was then built in C
  - . . . I say "emulation"

```
static uint16_t _segES; // just a global variable!
reg_t eax, ebx, ecx, edx, esi, edi; // x86 registers are globals!
static inline void SetES(uint16_t seg) {
  _segES = seg;
}
static inline uint8_t Load8ES(uint16_t off) { // ES-relative load
  return FarptrDeref8(FarptrFromParts(_segES, off));
}
```

  - Each comment assembly line then had equivalent C for this "emulator" placed under it

# The x86 real mode “emulator”

```
//                  mov     es, [bp+var_4]
SetES(var4);
//                  mov     bx, si
ebx.x = esi.x;
//                  inc     si
++esi.x;
//                  mov     al, es:[bx]
eax.l = Load8ES(ebx.x);
```

- Every single line of assembly is a C comment with a *trivial and obviously correct* translation into “x86-C” below it
- Can be eyeballed several times to check the correspondence is correct
- No structured control flow — gotos only
- Successfully compiles as a Linux binary, executes, and **fully decompresses the APE firmware!**

# "Raising" to real C

- The output of the "x86-C" decompression program was recorded
- The program was then modified to "raise" the "x86-C" code to more readable C
  - goto $\rightarrow$ if/for/while, etc.
- After each extremely small change, the decompressor was re-executed to confirm the output had not changed
  - Mistakes in "raising" were immediately caught
- Eventually via this methodical approach, all use of "goto" or the x86 "emulator" was dropped
- Result: bug-free description of the decompression algorithm in clean, readable C

# The Decompression Algorithm

- Already have the APE firmware decompressed, but at this point I was able to determine the nature of the decompression algorithm
- Extremely simple dictionary compression
- Compression stream comprises two types of symbol: literal bytes and dictionary references
  - A simple 2048 byte dictionary buffer
  - All output data (literal or referenced) is added to the dictionary buffer
  - References output some subset of the dictionary buffer

# The Decompression Algorithm

- I was also able to determine the identity of the compression algorithm
  - LZSS (Lempel-Ziv-Storer-Szymanski)
  - It has a Wikipedia article

Lempel–Ziv–Storer–Szymanski

From Wikipedia, the free encyclopedia

Most implementations stem from a public domain 1989 code by Haruhiko Okumura.[3][4]

3. ^ Simtel.net mirror. Haruhiko Okumura implementation of 1989. Archived on February 3, 1999.
4. ^ Haruhiko Okumura. History of Data Compression in Japan. Archived on January 10, 2016.

- The decompression code was linked from Wikipedia the entire time
  - Posted to simtel BBS in 1989: `msdos/arcutils/lz_comp2.zip`

# The Decompression Algorithm

- The decompression code was linked from Wikipedia the entire time
  - Posted to simtel BBS in 1989: `msdos/arcutils/lz_comp2.zip`
- Code is public domain
- Decompression code fits on one screen
- Broadcom version changes some constants slightly, but same algorithm

```c
void Decode(void) /* Just the reverse of Encode(). */
{
	int  i, j, k, r, c;
	unsigned int  flags;

	for (i = 0; i < N - F; i++) text_buf[i] = ' ';
	r = N - F;  flags = 0;
	for ( ; ; ) {
		if (((flags >>= 1) & 256) == 0) {
			if ((c = getc(infile)) == EOF) break;
			flags = c | 0xff00;		/* uses higher byte cleverly */
		}							/* to count eight */
		if (flags & 1) {
			if ((c = getc(infile)) == EOF) break;
			putc(c, outfile);  text_buf[r++] = c;  r &= (N - 1);
		} else {
			if ((i = getc(infile)) == EOF) break;
			if ((j = getc(infile)) == EOF) break;
			i |= ((j & 0xf0) << 4);  j = (j & 0x0f) + THRESHOLD;
			for (k = 0; k <= j; k++) {
				c = text_buf[(i + k) & (N - 1)];
				putc(c, outfile);  text_buf[r++] = c;  r &= (N - 1);
			}
		}
	}
}
```

# The Decompression Algorithm

- Your Broadcom Option ROM is compressed using some DOS compression code someone posted to a BBS in 1989
- If I had known the code was lying on an internet archive of 1989 BBS postings (of all places) the entire time, I could have saved a lot of pain
  - The only thing better than reverse engineering x86 real mode code is finding out you didn't have to
  - *but* I couldn't have figured out which algorithm it was without going through that process (especially with the tweaked constants)
  - Circular dependencies again

# Penetrating the APE

# The APE

- Disassembly began in earnest to study the now decompressed APE code
- APE turns out to be... an ARM Cortex-M3
  - Common little-endian 32-bit microcontroller core, like you'd get on an STM32
  - The APE has I/O peripherals and registers not mapped into PCIe and thus not available to the host
  - I want access
- With knowledge of the decompression algorithm, I could write a tool to build and compress a new APE image and flash it
- But image headers were still a bit mysterious
- And I wanted a quick way in that didn't require me to flash an image to the SPI flash permanently

# Disaster

- I suddenly discover that the APE firmware image appears to have an RSA signature at the end
- Oh no. Oh no, no, no
  - I suddenly get very depressed
  - I only continue after encouragement from others
    - “Maybe it isn’t checked”... I am skeptical, but continue
  - Have I mentioned RE is an emotional rollercoaster?

# APE: Looking for ways in

- Looked for ways to get shellcode execution on the APE using the existing code
- Lots of shared memory for communication with host & MIPS CPUs, so seemed plausible
- APE code implements a simple mailbox-style IPC mechanism using some SRAM accessible to the host, to allow host to send commands
- One of these commands allowed "scratchpad read"/"scratchpad write"
- Allows read/write to the APE's private SRAM
  - But it is bounds checked
  - But the bounds check **overlaps with the area the APE code is loaded into!?**

## Shellcode execution

- Send scratchpad write commands to upload shellcode to a certain code region
- Send a command which causes APE to jump into that code region
- **Success!**

# APE: Looking for ways in

- Not really a vulnerability
  - Can only be “exploited” from the host
  - Host is trusted and can reflash the entire firmware on the SPI flash anyway
  - It just saves writing a new image to flash, good for debug
- For the curious: I didn’t find any remotely exploitable vulnerabilities in any code

# APE: Shellcode (ape_shell.c)

- I wrote shellcode for the APE
- Shellcode implements a shared memory mailbox IPC mechanism with the host very similar to that used by APE firmware
- Shellcode allows arbitrary memory load/store or jump
- Shellcode immediately modifies entries in the ARM core's interrupt table to catch hardfaults
  - If a hardfault occurs during a memory access, it is trapped by our custom handler and we return an error

- We can now access the APE's private memory space!
- The APE boot ROM was successfully dumped for the first time
  - Got a much nicer ARM copy of the decompression code. If only I could have found a way to get to this copy first! Circular dependencies again...
  - Allowed image header format to be determined
  - Boot ROM turns out to have a way to boot an APE image from SRAM, instead of flash, which also proved useful

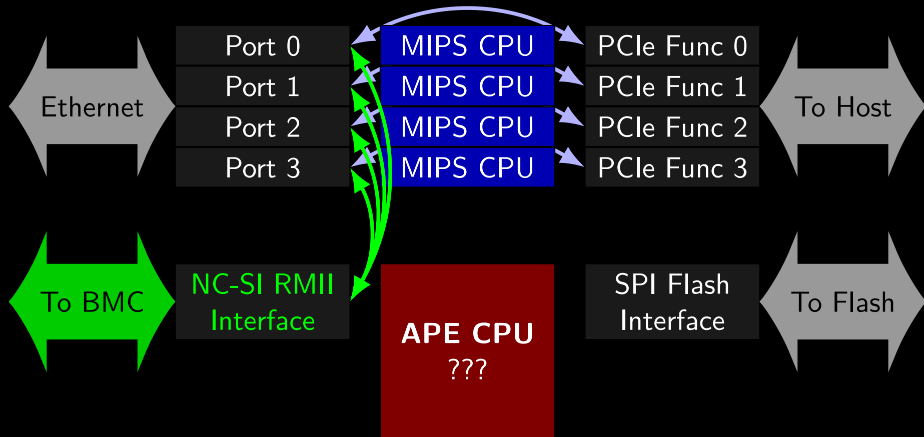
# Disaster averted

- The APE boot ROM **does not check** the RSA signature on the APE code image!
  - Why is this signature here?
- At this point we know how to get execution on the APE and form valid firmware images
- Build tooling was developed to allow building custom compressed APE images in C
  - Needed a compression function as well as the decompression function, so used the original BBS code for it
- Debug tooling was developed for probing APE registers, loading images from memory, etc.

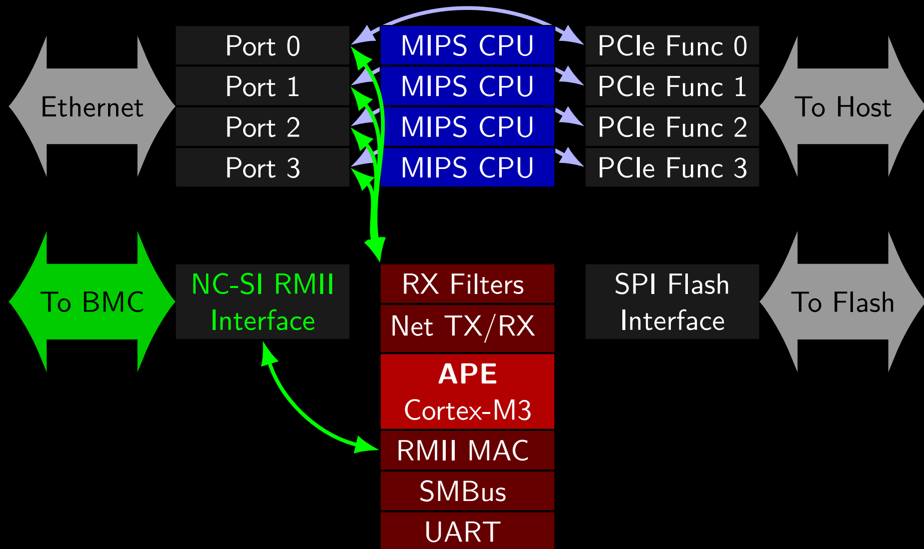
# APE: Reversing in earnest

- The APE firmware was comprehensively reverse engineered and an understanding of the APE's peripherals was obtained
  - Registers were completely unknown, no documentation
  - Broadcom diagnostic tool provided information on some registers
  - Others were simply guessed
- The APE has the following **special** peripherals:
  - Peripheral to talk to the BMC via the RMII interface
  - Peripheral to transmit to each network port
  - Peripheral to receive from each network port
  - Management filters to configure what APE receives from network
  - SMBus (not investigated)
  - UART (not investigated)
- All PCIe-mapped device registers are also accessible

# APE: Block Diagram



# APE: Block Diagram



# APE: What it basically does

- This is what the APE essentially does:
  - It reads Ethernet frames from the RMU peripheral (NC-SI RMII interface to the BMC)
  - Simple UART-style FIFO interface — no DMA, all memcpy
    - ★ Performance is not great (you won't get gigabit)
  - It writes the Ethernet frame to special port-specific SRAM and sets some registers to get it sent out
  - And on and on... (and likewise in the opposite direction)

# APE: What could it do?

If the APE were malicious (e.g. firmware compromise), what could it do?

- The APE can set management filters to do matches on packet headers to determine if they get forwarded to the APE
- This is usually used for broadcasts, DHCP, ARP, etc.
- APE can choose whether host also gets a copy, or whether the selected traffic only goes to the APE

**Ergo:**

- APE can eavesdrop on traffic from the network
- APE can prevent the host from seeing traffic from the network
- APE can MitM traffic between the network and the BMC
  - Better hope the BMC doesn't come with a default password
  - Even if it does, APE can MitM SSH and own the box (unless you check the SSH host key, if the vendor even provides it)

# Skeletons: The Great Broadcom BitBang

# The Great Broadcom BitBang

A problem

- We are able to build our own replacement firmware for the APE and get NC-SI working
- But it only works after the machine has been turned on once, not before
- Hmm. . .

# Found in the reversed APE firmware. . .

What is this???

```
if (!GetDevReg(0, REG_CHIP_ID)) {
  SetAPEReg(REG_APE__GPIO,      PIN0_OUT                 |       PIN2_OUT
                      | PIN0_MODE_OUT | PIN1_MODE_OUT);
  SetAPEReg(REG_APE__GPIO,      PIN0_OUT                 |       PIN2_OUT
                      | PIN0_MODE_OUT | PIN1_MODE_OUT | PIN2_MODE_OUT);

  for (int i=39; i; -i)
    SetAPEReg(REG_APE__GPIO,
              GetAPEReg(REG_APE__GPIO) ^ (PIN0_OUT | PIN1_OUT));

  MaskAPEReg(REG_APE__GPIO, PIN2_OUT);

  while (!GetDevReg(0, REG_CHIP_ID));
}
```

# Found in the reversed APE firmware...

Simplified in psuedocode:

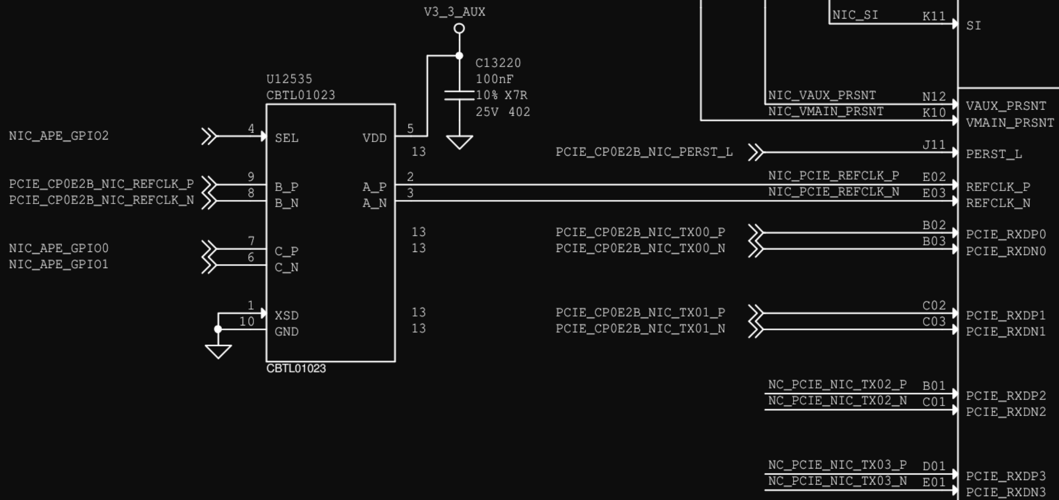
```
if (!Port0.REG_CHIP_ID) {      // If the silicon version register returns 0
                               //    (shouldn't be possible - it's a constant)
  set APE GPIO 0 as output (on)
  set APE GPIO 1 as output (off)
  set APE GPIO 2 as output (on)

  for (repeat 38 times)
    flip APE GPIOs 0 and 1;      // ?????

  set APE GPIO 2 as output (off)
  // Wait for the register to read nonzero
  while (!Port0.REG_CHIP_ID) { /* spin */ }
}
```

# What is this for???

- What do these “APE GPIOs” do?
  - Let’s take a look at the Talos II schematics



U9401A
BCM5719
R9405
1K
R9406
1K
NIC_SI
K11
SI
SPI
V3_3_AUX
C13220
100nF
10% X7R
25V 402
U12535
CBTL01023
NIC_VAUX_PRSNT
N12
VAUX_PRSNT
NIC_VMAIN_PRSNT
K10
VMAIN_PRSNT
NIC_APE_GPIO2
4
SEL
VDD
5
13
PCIE_CP0E2B_NIC_PERST_L
J11
PERST_L
PCIE_CP0E2B_NIC_REFCLK_P
PCIE_CP0E2B_NIC_REFCLK_N
9
8
B_P
B_N
A_P
A_N
2
3
NIC_PCIE_REFCLK_P
NIC_PCIE_REFCLK_N
E02
E03
REFCLK_P
REFCLK_N
13
13
PCIE_CP0E2B_NIC_TX00_P
PCIE_CP0E2B_NIC_TX00_N
B02
B03
PCIE_RXDP0
PCIE_RXDN0
NIC_APE_GPIO0
NIC_APE_GPIO1
7
6
C_P
C_N
1
10
XSD
GND
13
13
PCIE_CP0E2B_NIC_TX01_P
PCIE_CP0E2B_NIC_TX01_N
C02
C03
PCIE_RXDP1
PCIE_RXDN1
CBTL01023
NC_PCIE_NIC_TX02_P
NC_PCIE_NIC_TX02_N
B01
C01
PCIE_RXDP2
PCIE_RXDN2
NC_PCIE_NIC_TX03_P
NC_PCIE_NIC_TX03_N
D01
E01
PCIE_RXDP3
PCIE_RXDN3

U9401A
BCM5719

R9405
1K
R9406
1K

V3_3_AUX
C13220
100nF
10% X7R
25V 402

U12535
CBTL01023

NIC_SI K11 SI
SPI

NIC_VAUX_PRSNT N12 VAUX_PRSNT
NIC_VMAIN_PRSNT K10 VMAIN_PRSNT

NIC_APE_GPIO2 4 SEL VDD 5
13 PCIE_CP0E2B_NIC_PERST_L J11 PERST_L

PCIE_CP0E2B_NIC_REFCLK_P 9 B_P A_P 2 NIC_PCIE_REFCLK_P E02 REFCLK_P
PCIE_CP0E2B_NIC_REFCLK_N 8 B_N A_N 3 NIC_PCIE_REFCLK_N E03 REFCLK_N

13 PCIE_CP0E2B_NIC_TX00_P B02 PCIE_RXDP0
13 PCIE_CP0E2B_NIC_TX00_N B03 PCIE_RXDN0

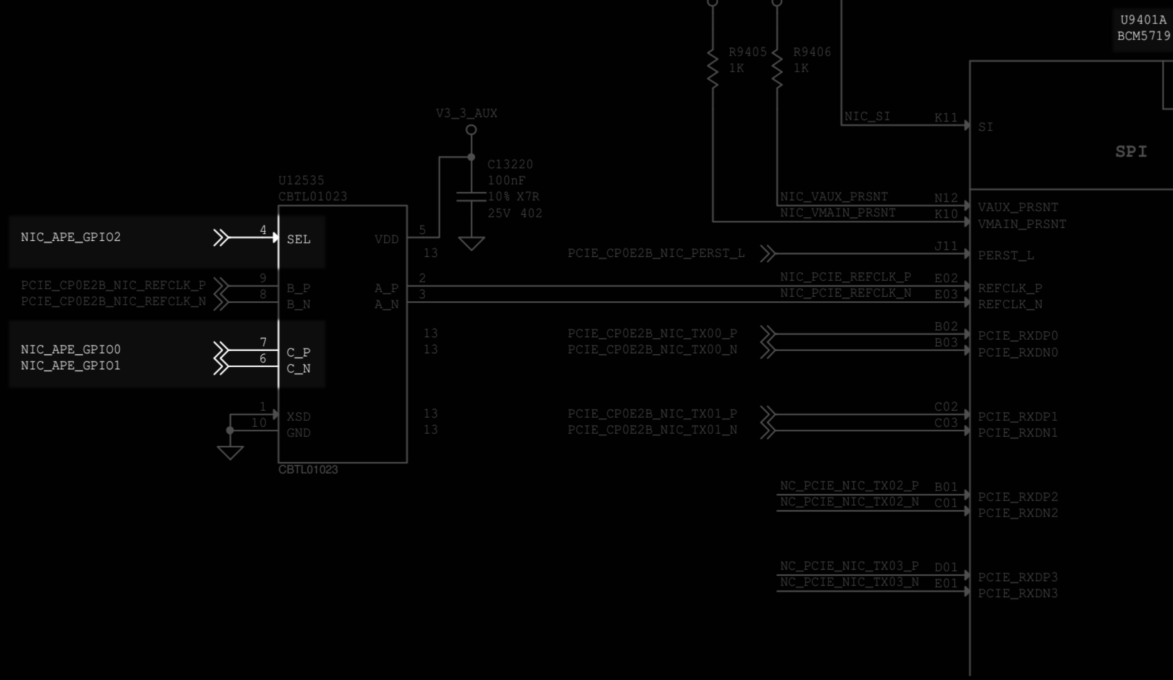
NIC_APE_GPIO0 7 C_P
NIC_APE_GPIO1 6 C_N

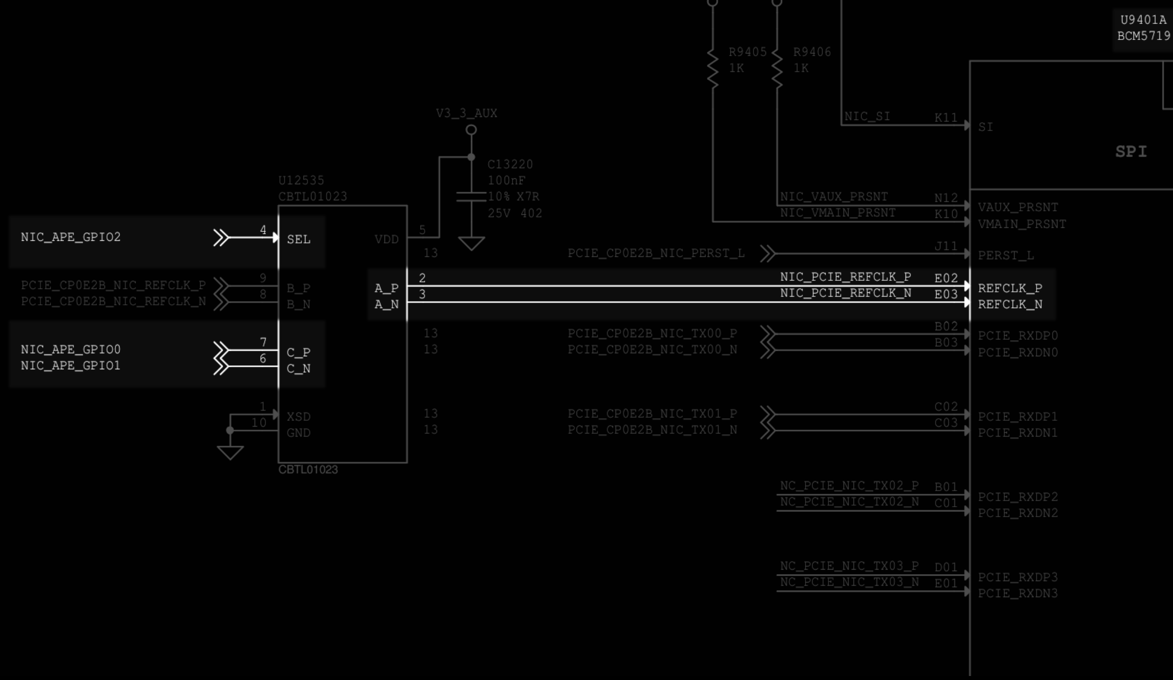
1 XSD 13 PCIE_CP0E2B_NIC_TX01_P C02 PCIE_RXDP1
10 GND 13 PCIE_CP0E2B_NIC_TX01_N C03 PCIE_RXDN1

CBTL01023

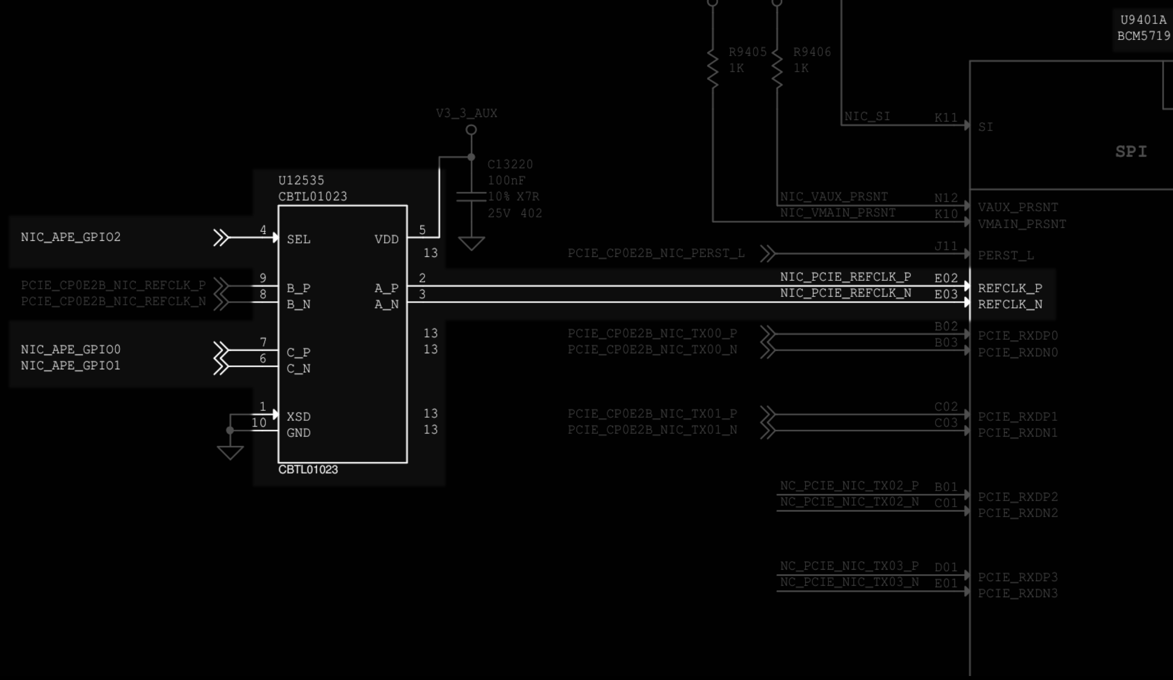
NC_PCIE_NIC_TX02_P B01 PCIE_RXDP2
NC_PCIE_NIC_TX02_N C01 PCIE_RXDN2

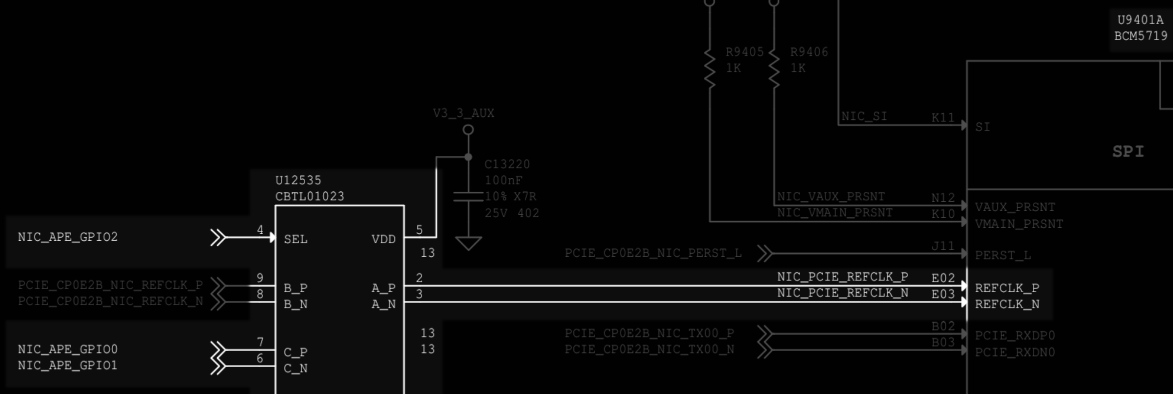
NC_PCIE_NIC_TX03_P D01 PCIE_RXDP3
NC_PCIE_NIC_TX03_N E01 PCIE_RXDN3


U9401A
BCM5719
R9405
1K
R9406
1K
NIC_SI
K11
SI
SPI
V3_3_AUX
C13220
100nF
10% X7R
25V 402
U12535
CBTL01023
NIC_VAUX_PRSNT
N12
VAUX_PRSNT
NIC_VMAIN_PRSNT
K10
VMAIN_PRSNT
NIC_APE_GPIO2
4
SEL
VDD
5
13
PCIE_CP0E2B_NIC_PERST_L
J11
PERST_L
PCIE_CP0E2B_NIC_REFCLK_P
PCIE_CP0E2B_NIC_REFCLK_N
9
8
B_P
B_N
A_P
A_N
2
3
NIC_PCIE_REFCLK_P
NIC_PCIE_REFCLK_N
E02
E03
REFCLK_P
REFCLK_N
PCIE_CP0E2B_NIC_TX00_P
PCIE_CP0E2B_NIC_TX00_N
B02
B03
PCIE_RXDP0
PCIE_RXDN0
NIC_APE_GPIO0
NIC_APE_GPIO1
7
6
C_P
C_N
1
10
XSD
GND
PCIE_CP0E2B_NIC_TX01_P
PCIE_CP0E2B_NIC_TX01_N
C02
C03
PCIE_RXDP1
PCIE_RXDN1
CBTL01023
NC_PCIE_NIC_TX02_P
NC_PCIE_NIC_TX02_N
B01
C01
PCIE_RXDP2
PCIE_RXDN2
NC_PCIE_NIC_TX03_P
NC_PCIE_NIC_TX03_N
D01
E01
PCIE_RXDP3
PCIE_RXDN3



U9401A
BCM5719
R9405
1K
R9406
1K
NIC_SI
K11
SI
SPI
V3_3_AUX
C13220
100nF
10% X7R
25V 402
U12535
CBTL01023
NIC_VAUX_PRSNT
N12
VAUX_PRSNT
NIC_VMAIN_PRSNT
K10
VMAIN_PRSNT
NIC_APE_GPIO2
4
SEL
VDD
5
13
PCIE_CP0E2B_NIC_PERST_L
J11
PERST_L
PCIE_CP0E2B_NIC_REFCLK_P
PCIE_CP0E2B_NIC_REFCLK_N
9
8
B_P
B_N
A_P
A_N
2
3
NIC_PCIE_REFCLK_P
NIC_PCIE_REFCLK_N
E02
E03
REFCLK_P
REFCLK_N
13
13
PCIE_CP0E2B_NIC_TX00_P
PCIE_CP0E2B_NIC_TX00_N
B02
B03
PCIE_RXDP0
PCIE_RXDN0
NIC_APE_GPIO0
NIC_APE_GPIO1
7
6
C_P
C_N
1
10
XSD
GND
13
13
PCIE_CP0E2B_NIC_TX01_P
PCIE_CP0E2B_NIC_TX01_N
C02
C03
PCIE_RXDP1
PCIE_RXDN1
CBTL01023
NC_PCIE_NIC_TX02_P
NC_PCIE_NIC_TX02_N
B01
C01
PCIE_RXDP2
PCIE_RXDN2
NC_PCIE_NIC_TX03_P
NC_PCIE_NIC_TX03_N
D01
E01
PCIE_RXDP3
PCIE_RXDN3


U9401A
BCM5719
R9405
1K
R9406
1K
NIC_SI
K11
SI
SPI
V3_3_AUX
C13220
100nF
10% X7R
25V 402
U12535
CBTL01023
NIC_VAUX_PRSNT
N12
VAUX_PRSNT
NIC_VMAIN_PRSNT
K10
VMAIN_PRSNT
NIC_APE_GPIO2
SEL
VDD
PCIE_CP0E2B_NIC_PERST_L
J11
PERST_L
PCIE_CP0E2B_NIC_REFCLK_P
PCIE_CP0E2B_NIC_REFCLK_N
B_P
B_N
A_P
A_N
NIC_PCIE_REFCLK_P
E02
NIC_PCIE_REFCLK_N
E03
REFCLK_P
REFCLK_N
PCIE_CP0E2B_NIC_TX00_P
PCIE_CP0E2B_NIC_TX00_N
B02
B03
PCIE_RXDP0
PCIE_RXDN0
NIC_APE_GPIO0
NIC_APE_GPIO1
C_P
C_N
XSD
GND
PCIE_CP0E2B_NIC_TX01_P
PCIE_CP0E2B_NIC_TX01_N
C02
C03
PCIE_RXDP1
PCIE_RXDN1
CBTL01023
NC_PCIE_NIC_TX02_P
NC_PCIE_NIC_TX02_N
B01
C01
PCIE_RXDP2
PCIE_RXDN2
NC_PCIE_NIC_TX03_P
NC_PCIE_NIC_TX03_N
D01
E01
PCIE_RXDP3
PCIE_RXDN3

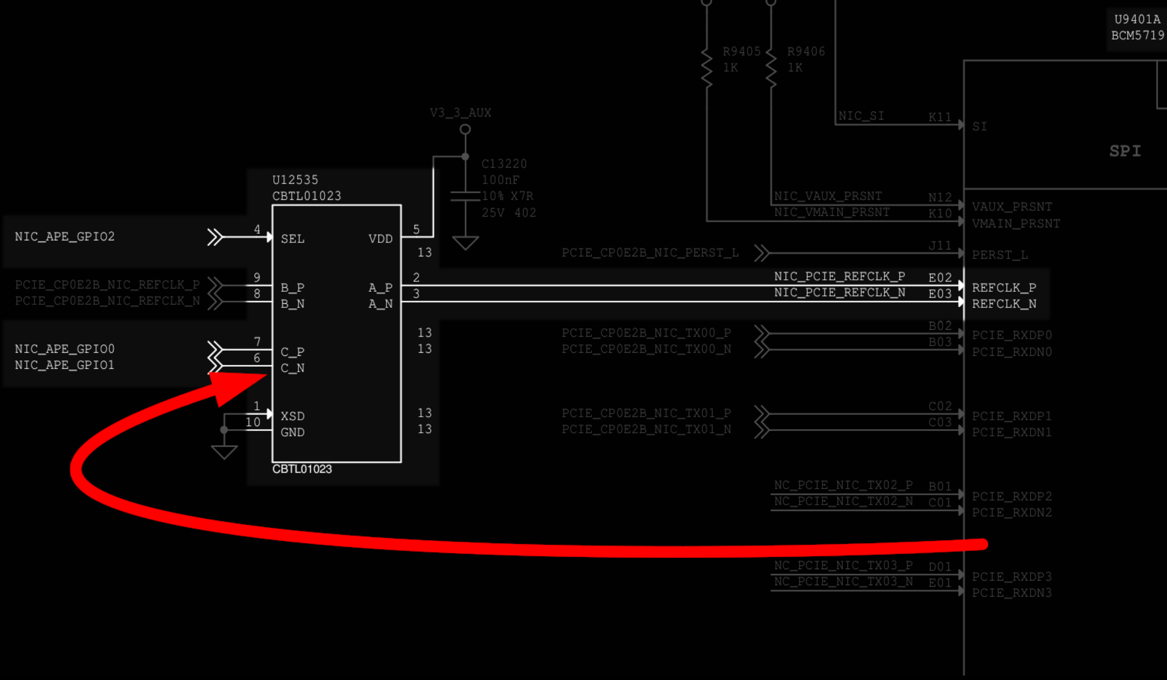


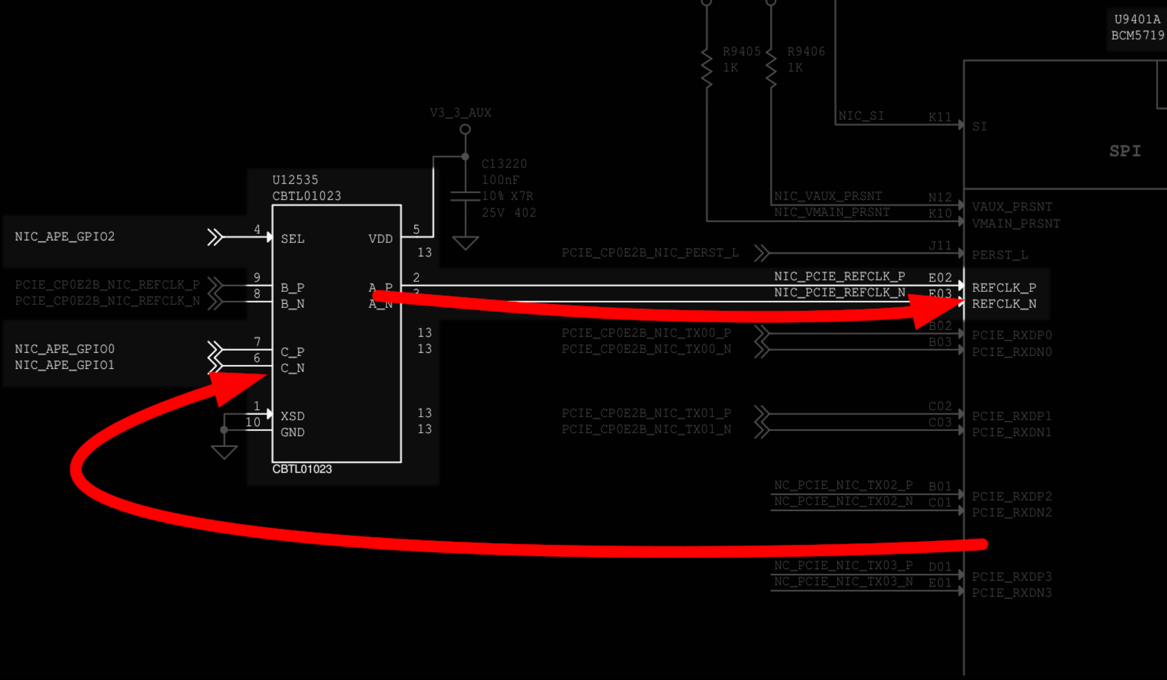

# CBTL01023

## 3.3 V, one differential channel, 2 : 1 multiplexer/demultiplexer switch for PCI Express Gen3

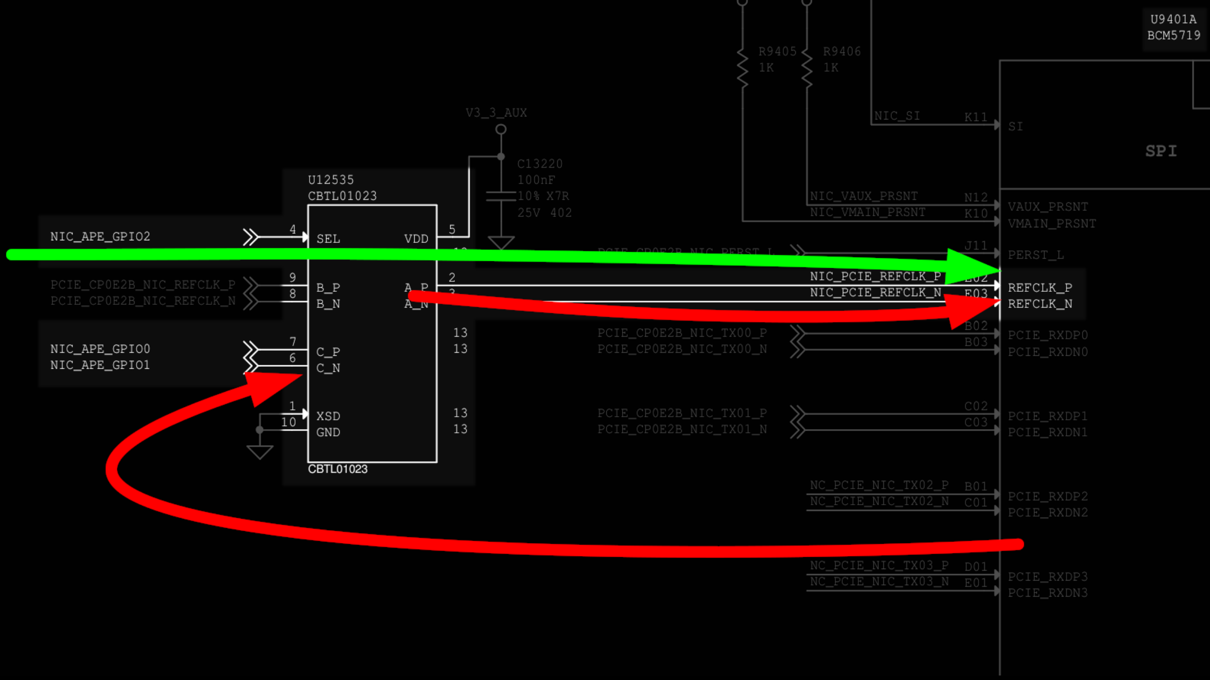
Rev. 1 — 24 October 2011

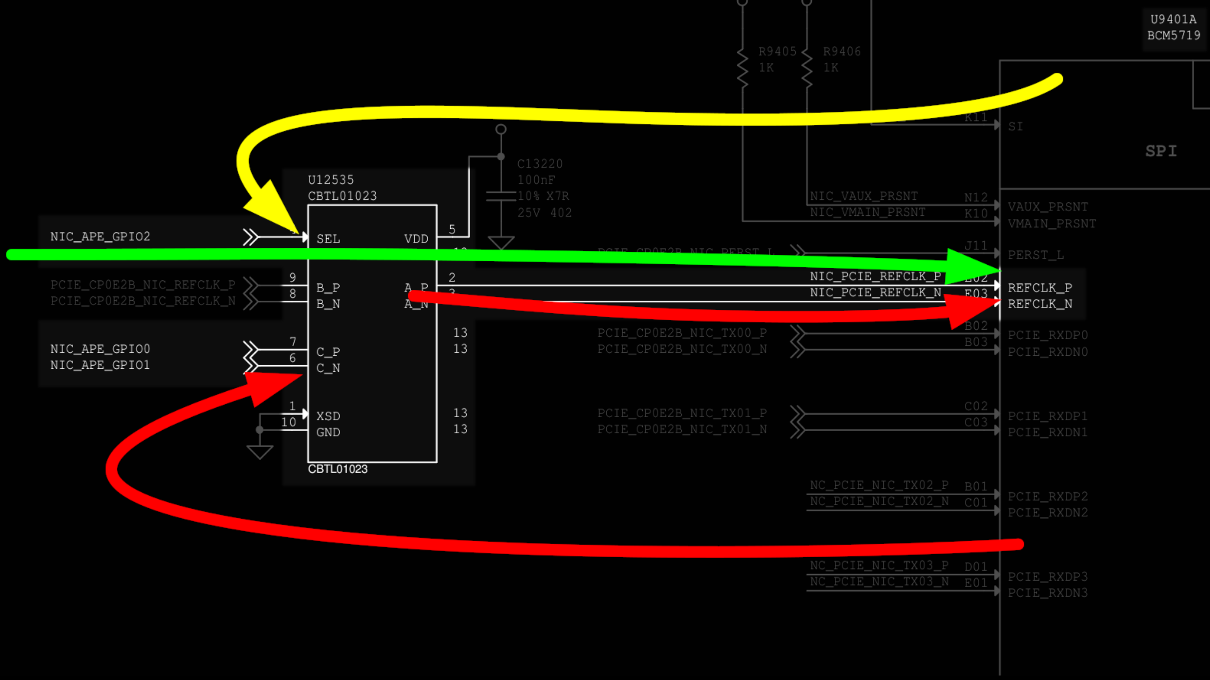
Product data sheet


U9401A
BCM5719
R9405
1K
R9406
1K
V3_3_AUX
C13220
100nF
10% X7R
25V 402
U12535
CBTL01023
NIC_APE_GPIO2
SEL
VDD
NIC_SI
K11
SI
SPI
NIC_VAUX_PRSNT
N12
VAUX_PRSNT
NIC_VMAIN_PRSNT
K10
VMAIN_PRSNT
PCIE_CP0E2B_NIC_PERST_L
J11
PERST_L
PCIE_CP0E2B_NIC_REFCLK_P
PCIE_CP0E2B_NIC_REFCLK_N
B_P
B_N
A_P
A_N
NIC_PCIE_REFCLK_P
E02
REFCLK_P
NIC_PCIE_REFCLK_N
E03
REFCLK_N
PCIE_CP0E2B_NIC_TX00_P
B02
PCIE_RXDP0
PCIE_CP0E2B_NIC_TX00_N
B03
PCIE_RXDN0
NIC_APE_GPIO0
NIC_APE_GPIO1
C_P
C_N
XSD
GND
PCIE_CP0E2B_NIC_TX01_P
C02
PCIE_RXDP1
PCIE_CP0E2B_NIC_TX01_N
C03
PCIE_RXDN1
CBTL01023
NC_PCIE_NIC_TX02_P
B01
PCIE_RXDP2
NC_PCIE_NIC_TX02_N
C01
PCIE_RXDN2
NC_PCIE_NIC_TX03_P
D01
PCIE_RXDP3
NC_PCIE_NIC_TX03_N
E01
PCIE_RXDN3



U9401A
BCM5719
R9405
1K
R9406
1K
V3_3_AUX
NIC_SI
K11
SI
SPI
C13220
100nF
10% X7R
25V 402
U12535
CBTL01023
NIC_VAUX_PRSNT
N12
VAUX_PRSNT
NIC_VMAIN_PRSNT
K10
VMAIN_PRSNT
NIC_APE_GPIO2
4
SEL
VDD
5
13
PCIE_CP0E2B_NIC_PERST_L
J11
PERST_L
PCIE_CP0E2B_NIC_REFCLK_P
PCIE_CP0E2B_NIC_REFCLK_N
9
8
B_P
B_N
A_P
A_N
2
3
NIC_PCIE_REFCLK_P
E02
NIC_PCIE_REFCLK_N
E03
REFCLK_P
REFCLK_N
13
13
PCIE_CP0E2B_NIC_TX00_P
PCIE_CP0E2B_NIC_TX00_N
B02
B03
PCIE_RXDP0
PCIE_RXDN0
NIC_APE_GPIO0
NIC_APE_GPIO1
7
6
C_P
C_N
1
10
XSD
GND
13
13
PCIE_CP0E2B_NIC_TX01_P
PCIE_CP0E2B_NIC_TX01_N
C02
C03
PCIE_RXDP1
PCIE_RXDN1
CBTL01023
NC_PCIE_NIC_TX02_P
B01
NC_PCIE_NIC_TX02_N
C01
PCIE_RXDP2
PCIE_RXDN2
NC_PCIE_NIC_TX03_P
D01
NC_PCIE_NIC_TX03_N
E01
PCIE_RXDP3
PCIE_RXDN3


U9401A
BCM5719
R9405
1K
R9406
1K
NIC_SI
K11
SI
SPI
V3_3_AUX
C13220
100nF
10% X7R
25V 402
U12535
CBTL01023
NIC_VAUX_PRSNT
N12
VAUX_PRSNT
NIC_VMAIN_PRSNT
K10
VMAIN_PRSNT
NIC_APE_GPIO2
4
SEL
VDD
5
J11
PERST_L
PCIE_CP0E2B_NIC_REFCLK_P
PCIE_CP0E2B_NIC_REFCLK_N
9
8
B_P
B_N
A_P
A_N
2
NIC_PCIE_REFCLK_P
NIC_PCIE_REFCLK_N
E03
REFCLK_P
REFCLK_N
13
PCIE_CP0E2B_NIC_TX00_P
PCIE_CP0E2B_NIC_TX00_N
B02
B03
PCIE_RXDP0
PCIE_RXDN0
NIC_APE_GPIO0
NIC_APE_GPIO1
7
6
C_P
C_N
1
10
XSD
GND
PCIE_CP0E2B_NIC_TX01_P
PCIE_CP0E2B_NIC_TX01_N
C02
C03
PCIE_RXDP1
PCIE_RXDN1
CBTL01023
NC_PCIE_NIC_TX02_P
NC_PCIE_NIC_TX02_N
B01
C01
PCIE_RXDP2
PCIE_RXDN2
NC_PCIE_NIC_TX03_P
NC_PCIE_NIC_TX03_N
D01
E01
PCIE_RXDP3
PCIE_RXDN3



U9401A
BCM5719
R9405
1K
R9406
1K
K11
SI
SPI
C13220
100nF
10% X7R
25V 402
U12535
CBTL01023
NIC_VAUX_PRSNT
NIC_VMAIN_PRSNT
N12
K10
VAUX_PRSNT
VMAIN_PRSNT
NIC_APE_GPIO2
SEL
VDD
5
J11
PERST_L
PCIE_CP0E2B_NIC_REFCLK_P
PCIE_CP0E2B_NIC_REFCLK_N
9
8
B_P
B_N
A_P
A_N
2
3
NIC_PCIE_REFCLK_P
NIC_PCIE_REFCLK_N
E03
REFCLK_P
REFCLK_N
13
13
PCIE_CP0E2B_NIC_TX00_P
PCIE_CP0E2B_NIC_TX00_N
B02
B03
PCIE_RXDP0
PCIE_RXDN0
NIC_APE_GPIO0
NIC_APE_GPIO1
7
6
C_P
C_N
1
10
XSD
GND
13
13
PCIE_CP0E2B_NIC_TX01_P
PCIE_CP0E2B_NIC_TX01_N
C02
C03
PCIE_RXDP1
PCIE_RXDN1
CBTL01023
NC_PCIE_NIC_TX02_P
NC_PCIE_NIC_TX02_N
B01
C01
PCIE_RXDP2
PCIE_RXDN2
NC_PCIE_NIC_TX03_P
NC_PCIE_NIC_TX03_N
D01
E01
PCIE_RXDP3
PCIE_RXDN3

# You have got to be kidding

```
if (!GetDevReg(0, REG_CHIP_ID)) {
  ...GPIO weirdness...
  while (!GetDevReg(0, REG_CHIP_ID));
}
```

- REG_CHIP_ID is a silicon version register ("BCM5719 Rev A0")
  - It should **never** be zero, it's a hardcoded constant
  - If it's reading zero, this means half the chip isn't "up" yet
  - This APE code is trying to fix this and get the rest of the chip to come up

# The Great Broadcom BitBang

- Turns out: Broadcom hardcoded the chip so the non-APE parts only come online after several dozen PCIe REFCLK cycles
  - After all, the host PCIe will always be there right?
  - . . . Unless the host is off
- Rather than actually fix this problem they **mandated every customer put an ANALOGUE PCIe MUX CHIP on their board so the APE firmware can MANUALLY BITBANG CLOCK CYCLES on the PCIe REFCLK input of the SAME CHIP to get the rest of it to come up**
- This is not a joke
- If you have a server with a BCM5719 and NC-SI, your server vendor has been forced to add an extra PCIe mux chip to the BOM because Broadcom ***decided to ship this as a design!***

# Project Ortega: Aftermath

# Project Ortega: Outputs

- Since all of this is based on reversing proprietary code, I was “tainted” by the process
  - There was a desire to produce replacement open source firmware with a clean licence
  - All of the gathered knowledge was used to produce documentation on how to write replacement firmware (cleanroom reverse engineering)
- Evan Lojewski (meklort) wrote replacement firmware in C++ using this information
  - This firmware now ships on all newly ordered Talos II and Blackbird POWER9 systems, making these systems 100% open source firmware — **mission accomplished!**
  - It is also distributed by LVFS (fwupd)
  - You can also install it on a BCM5719 PCIe card for your PC
  - Please give a **big thanks to meklort** for making open source firmware for the BCM5719 a reality

# RE is an emotional rollercoaster

- Sometimes, you think "I'm never going to figure this out"
  - A million registers with unknown names and unknown values
  - An enormity of unknown code
- Then you figure something out, and it lets you figure other things out, and discoveries snowball: the avalanche effect
- Then the avalanche ends and you're stuck again
- Constant oscillation between exhilaration and a sense of impossibility
- Successful RE requires managing these emotions and persevering
- Weirdly like doing a 100-dimensional crossword
  - With a million rows and a million columns

# Looking back

- I did not consider myself good at RE before starting this project
- I just decided to take a quick look at the firmware image to see how inscrutable it was, and was surprised by what I found... then I got sucked in
  - Never could have imagined getting to this point
  - I have a lot of curiosity — a powerful motivator
  - What motivates others to do RE?

# Join the community

- Come talk on IRC: #talos-workstation (Liberachat)
- Use the open source firmware: https://github.com/meklort/bcm5719-fw
- Read the Project Ortega documentation: https://github.com/hlandau/ortega
- Ask me questions about any of this: https://www.devever.net/~hl/contact
- I am always available to answer questions on BCM5719


## Acknowledgements

- Thanks to IBM for open sourcing the POWER9 firmware
- Thanks to Timothy Pearson at Raptor Engineering for creating the Talos II and providing hardware and resources in support of this effort
- Thanks to Evan Lojewski for creating open source replacement firmware
- Thanks to the Talos community for the support and encouragement without which this project could never have been possible