Apollolake implementation

Andrey Petrov
Software Engineer
Intel

Contents

- The Apollolake SoC
- New boot flow
- How coreboot fits in
  - New bootblock
  - New romstage
  - New memory mapping
  - New postcar stage
- FSP 2.0 driver
- How to build coreboot for Apollolake SoC
- Questions?

Apollolake SoC

The Apollolake SoC

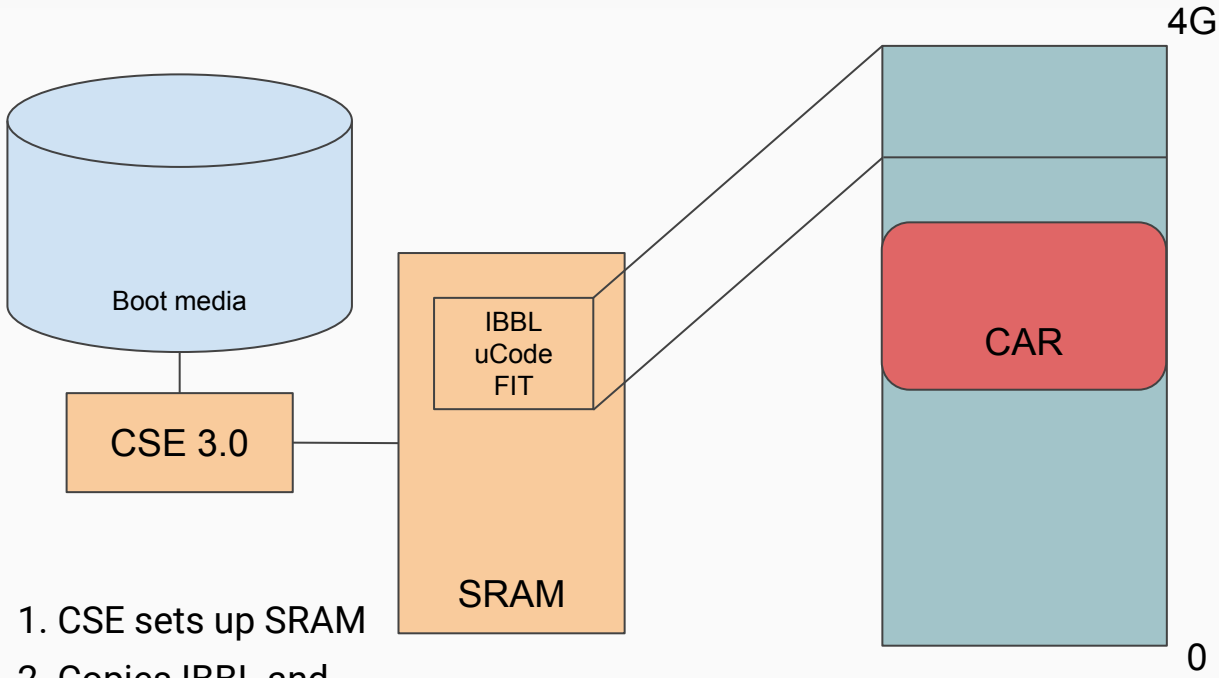
- New Atom SoC, 14nm, successor of Braswell
- Can boot firmware from new media (eMMC, UFS, USB, etc)
- 1 MiB L2 cache per core and 24KiB L1 cache
- Comes with on-die SRAM (384 KiB)
  - But it is read-only for the CPU
  - Gets torn early in the boot flow
- No traditional IO-based UART
  - dw8250 chip is used
  - Only MMIO can be used
- Old PIT timer is gone

New boot flow

New boot flow (1/2) Key boot components

1. PMC
    a. 32b ARC controller, 64kB local instruction memory, 16kB local data memory, local ROM
    b. First microcontroller active after reset
    c. Always On (during runtime)
2. CSE: Converged Security Engine
    a. 32b uiA controller, 384kB local SRAM, 32k combined cache, local ROM
    b. Responsible for retrieving and validating all firmware
    c. 4 different CSE firmware loads
        - ROM: responsible for retrieving and validating CSE bring up FW from NVM
        - RBE: ROM boot extensions
        - ukernel/Bring up: responsible for initializing all other pieces of the system and retrieving and validating all other pre-OS FW
        - Runtime: ukernel+run time applications
          Can share SRAM with host CPU, control IMR (isolated memory ranges), etc
3. Host CPUs
4. Storage controllers

New boot flow (2/2)



1. CSE sets up SRAM
2. Copies IBBL and assets into shared SRAM
3. Maps portion of SRAM into CPU address space
4. CPU is out reset. uCode is loaded, and IBBL is executed from reset vector
5. IBBL sets up CAR, relocates to CAR, and initiates firmware transfer into SRAM
6. CSE copies MRC into SRAM and IBBL copies it into CAR
7. MRC is run in CAR
8. Once memory is trained, CSE copies rest of firmware into DRAM
9. Rest as usual

How does coreboot fit in?

Quick summary

- Bootblock acts as IBBL
- We do not relocate bootblock to CAR, we just load romstage into it
- We do not use CSE to transfer from boot media
  - Memory mapped SPI is used instead
- romstage is run in CAR
- FSP-M (MemoryInit) runs in CAR
- FSP-S (SiliconInit) runs in ramstage
- Note we do not rely on memory mapping and each stage/component is “loaded”, so other media driver can be added

New x86 bootblock

Old bootblock

- Monolithic flow
- Mix of assembly code and C code compiled with romcc
- Assumes fixed memory mapping
- Limited functionality: parse cbfs, jump

New bootblock

- Bootblock sets up cache-as-ram by itself
- Provides “C” environment early on
- Can link with any other code
- Linked as all other stages
- Shares CAR memory layout with other CAR stages
    - Pre-mem variables (e.g timestamps) can be preserved

How coreboot fits in - bootblock

BOOTBLOCK (executed in SRAM)

1.1. Entry point is in common code bootblock assembly `src/arch/x86/bootblock_crt0.S`

- Protected mode switch, GDT, etc are provided by common x86 code

1.2. Jump into SoC-provided `bootblock_pre_c_entry`, implemented in `src/soc/intel/apollolake/cache_as_ram.S`

- sets up cache-as-ram and early MTRRs
- Non-evict mode (NEM) is activated
- Stack pointer is set to CAR region
- Jumps into `src/soc/intel/apollolake/bootblock.c bootblock_c_entry()`

1.3. Sets up essentials then jumps into `main()` in `src/lib/bootblock.c`

1.4. Library bootblock main runs callbacks:

- `console_init()`
- `bootblock_soc_early_init()`
- `bootblock_mainboard_early_init()` in order to disable watchdog, set up BARs, configure GPIOs, etc

1.5. Finally `run_romstage()`

1.6. `run_romstage()` loads and starts romstage

New romstage

Unlike other x86 SoCs, runs in CAR

- Entry point is **`car_stage_entry()`**
  - “carstage” infrastructure is shared for “car” stages like verstage and romstage
  - FSP driver loads FSP-M blob in CAR
  - `fsp_memory_init()`
  - Once memory is trained cache needs to be enabled
- So, memory is trained, how do you tear down CAR?
  - We want to enable L2 cache as early as possible
  - .. but you can’t tear down car while running in it
  - ramstage can be uncompressed and loaded into uncached memory
  - .. but it will be slow
- Load new stage **postcar** that is dedicated to tear down

New postcar stage

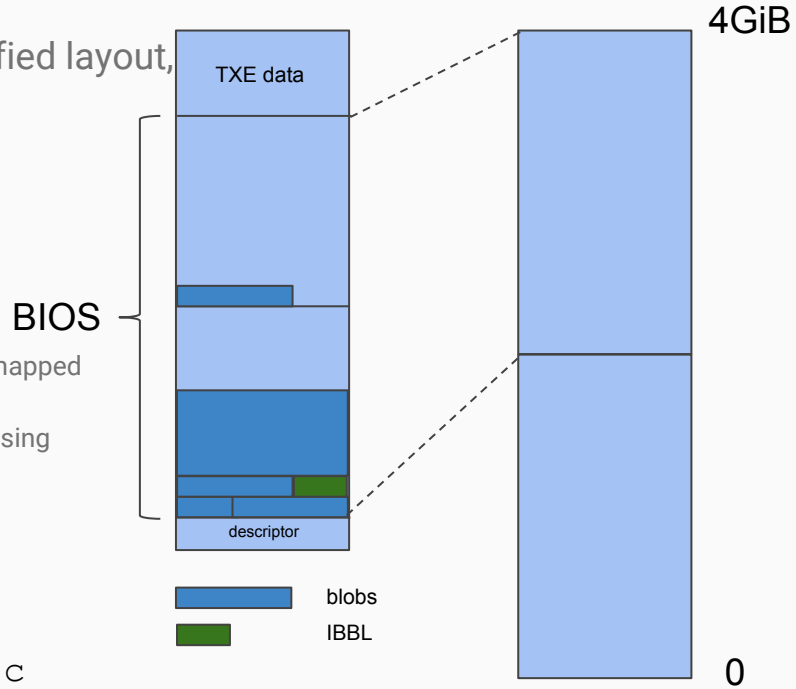
- Small (~20kb) new generic stage
- Runs in DRAM (cached or uncached)
- SoC provides a callback to flush caches and tear down CAR
- Sets up MTRRs to ensure ramstage is cached
- Loads and runs ramstage

CQOS instead of NEM

- Traditionally CAR is implemented by setting MTRRs and turning on No Evict Mode (NEM)
  - In NEM mode, data written stay in cache and is not evicted
  - However, rest of cache can not be used because all cache fills are blocked
  - In NEM, L1 cache is still usable
- How L2 can be still used for storing data/code and cache?
- With CQOS
    - Originally meant to be used by OS to prevent low-priority tasks cause cache evictions
    - Cache ways can be locked to prevent fills
    - Rest of cache behaves normally
    - This way we can use CAR and take advantage of L2 cache
    - Currently WIP, patches pending upload

New memory mapping

- Since eMMC and SPI needed to have unified layout, boot media layout has been changed
- Intel firmware descriptor:
  - ME region is not used
  - BIOS and ME are **fused** together
  - TXE gets its own region for data storage
- Memory mapping
  - Unlike previous SoC, not entire flash is memory mapped
  - Only BIOS region is mapped under 4gb
  - Size of “BIOS” region in descriptor affects addressing



`soc/intel/apollolake/mmap_boot.c`
`mainboard/xxx/Kconfig, IFD_BIOS_START, IFD_BIOS_END`

FSP 2.0 driver

FSP 2.0 driver

FSP 2.0

- Since different components are ran in different non-contiguous addresses, monolithic blob does not work
- 3 blobs instead of one
  - FSP-T, to set up CAR, not used
  - **FSP-M**, MemoryInit
  - **FSP-S**, SiliconInit
- Blobs can be run XIP or from cache-as-ram
- Driver provides coreboot<->FSP interface
- Driver offers callbacks for SoC to implement: before memory init, before silicon init

How to build Apollolake

How can you build Apollolake coreboot

1. Make sure your board is hard strapped for “SPI Boot Source”
2. make nconfig, select mainboard, ex google/reef
3. add this to your .config:

   ```
   CONFIG_NEED_IFWI=y
   CONFIG_IFWI_FMAP_NAME="IFWI"
   CONFIG_IFWI_FILE_NAME="3rdparty/blobs/mainboard/google/reef/fitimage.bin"
   ```
4. Prepare fitimage.bin
   - Make sure you disable bootguard by editing the XMLs or fit.exe GUI.
   - Use fit.exe to generate "fitimage.bin".
   - This is a fully flashable image will act as a template to generate coreboot.rom
5. `dd` first 4k of fitimage.bin to produce descriptor.bin
6. Run `ifdtool -d fitimage.bin`to see the size of BIOS region
7. Update IFD_BIOS_START and IFD_BIOS_END to match ifdtool output
8. Drop fitimage.bin and descriptor.bin `3rdparty/blobs/mainboard/google/reef/fitimage.bin`
9. make
10. flash build/coreboot.rom

Questions?