Zynq Boot and Configuration Procedures

Accelerating Your Success™

Why Would This Presentation Matter to You?

Why Would This Presentation Matter to You?

- **If you are designing a Zynq®-7000 All Programmable SoC embedded processing system and need any of the following**
  - Fast boot and configuration time
  - Cost effective boot and configuration solution
  - Secure boot and configuration
  - Boot recovery (golden image/fallback)

Why Would This Presentation Matter to You?

- **If you are designing a Zynq®-7000 All Programmable SoC embedded processing system and need any of the following**
  - Fast boot and configuration time
  - Cost effective boot and configuration solution
  - Secure boot and configuration
  - Boot recovery (golden image/fallback)

- **Then you need to know about the Zynq-7000 All Programmable SoC available boot and configuration options**

Objective

- **Become familiar with the Xilinx® Zynq-7000 All Programmable SoC boot and configuration procedures**

- **Know how to choose the best boot and configuration method that meets your application needs**

Agenda

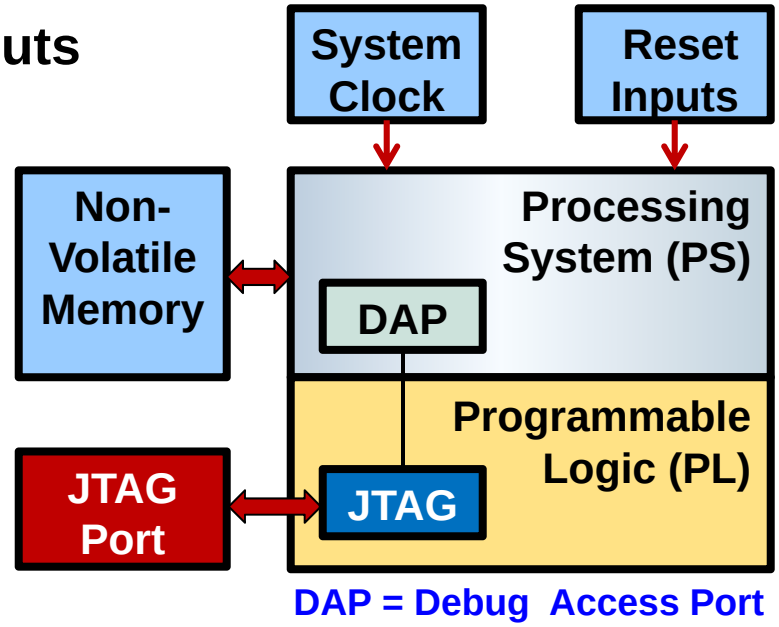
- Introduction to Zynq Boot and Configuration Process
- Non-Secure Boot and Configuration
- Secure Boot and Configuration
- Multi-Boot
- Boot and Configuration Devices
- Next Steps

Introduction to Zynq Boot and Configuration Process

Zynq High-Level Boot and Configuration Overview

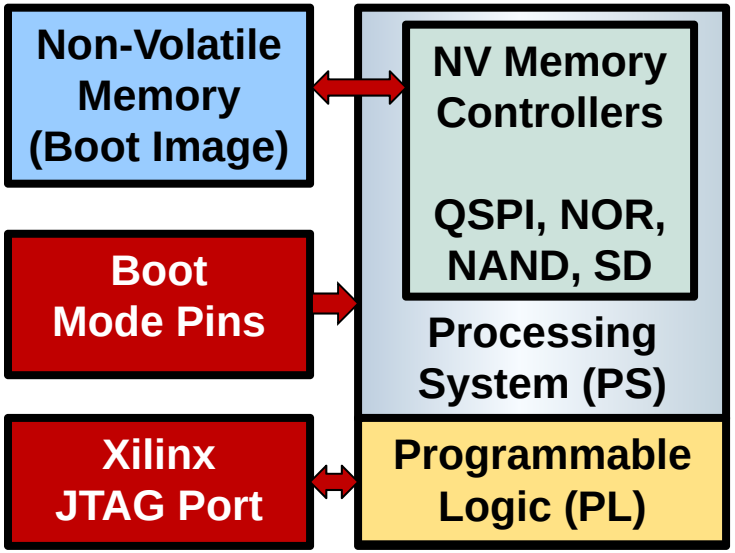
- **Zynq Processing System (PS) boots from external non-volatile memory just like an ASSP**
  - PS configures the Programmable Logic (PL)
  - User can also boot the PS and configure the PL over the JTAG port

- **External reset and system clock inputs are required to boot the PS**
  - *Power-On Reset (POR)*, asserted minimum of 100us after power good
  - *System Clock (PS_CLK)*, 30 – 60 MHz (typically 33.33 MHz)
  - *System Reset (SRST)* can be asserted after power-on to reset the processor (minimum of 3 *PS_CLK* clocks)



Zynq Boot and Configuration Options

- **Zynq supports the following boot and configuration modes**
  - *Secure boot* - boot image is encrypted
  - *Non-secure boot* – boot image is unencrypted
- **Secure and non-secure boot modes support four *Master Boot* methods where Zynq boots and configures itself from one of the following boot devices based on the *Boot Mode Pins***
  - QSPI Flash
  - NOR Flash
  - NAND Flash
  - SD Card
- **Non-secure boot mode also supports one *Slave Boot* method**
  - Used for debug and development
  - User boots the PS and configures the PL over the JTAG port



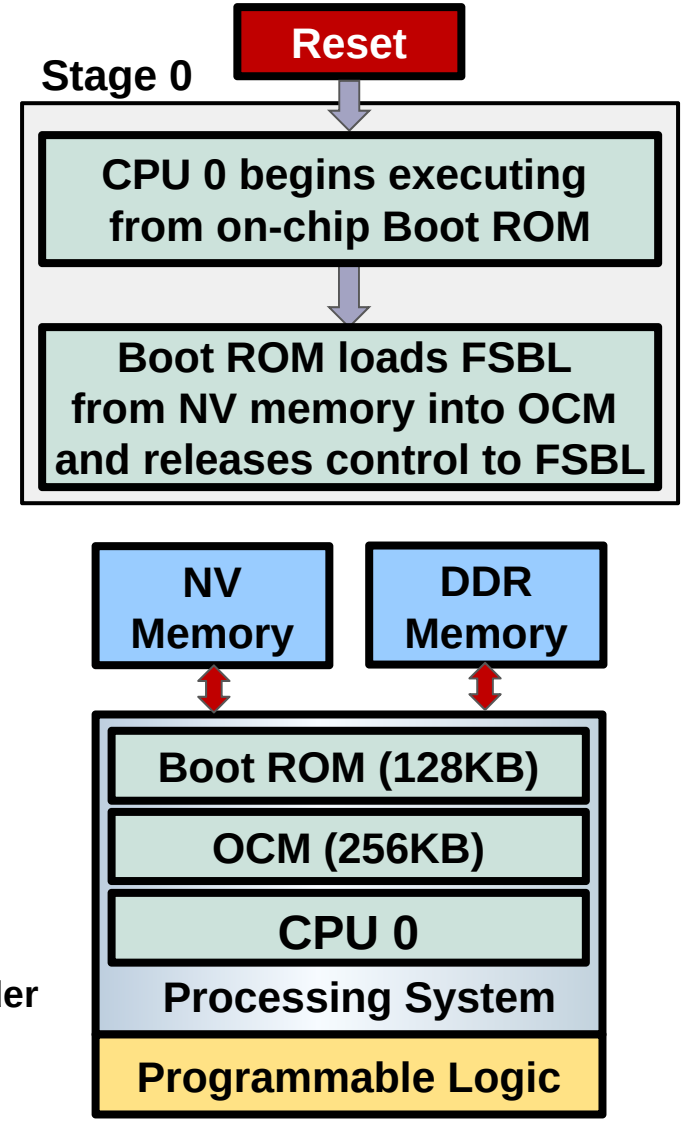
Boot Mode Pins

- **Boot Mode Pins are sampled on Power-On Reset (POR) and stored in the PS *BOOT_MODE* register**
  - *BOOT_MODE* register values are used to select the boot device

Boot Mode Pins/ Boot Device	MIO[5]	MIO[4]	MIO[3]	MIO[2]
Cascaded JTAG	0	0	0	0
Independent JTAG	0	0	0	1
NOR Flash	0	0	1	NA
NAND Flash	0	1	0	
QSPI Flash	1	0	0	
SD Card	1	1	0	

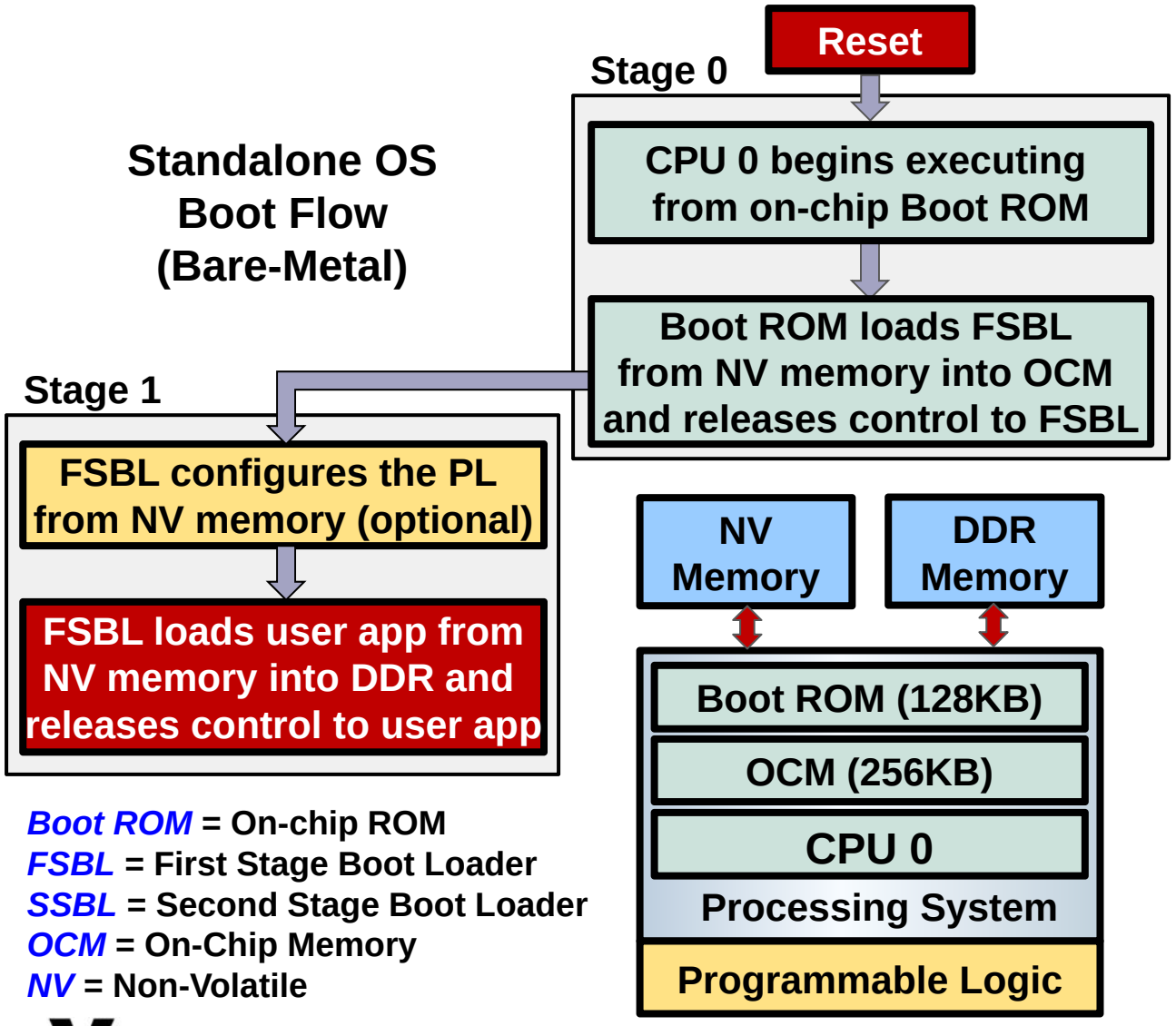
  - *Cascaded JTAG* – Xilinx tools are used to configure the PL and boot the PS
  - *Independent JTAG* – Xilinx tools are used to configure the PL while third party tools are used to boot the PS

Typical Zynq Boot and Configuration Flow



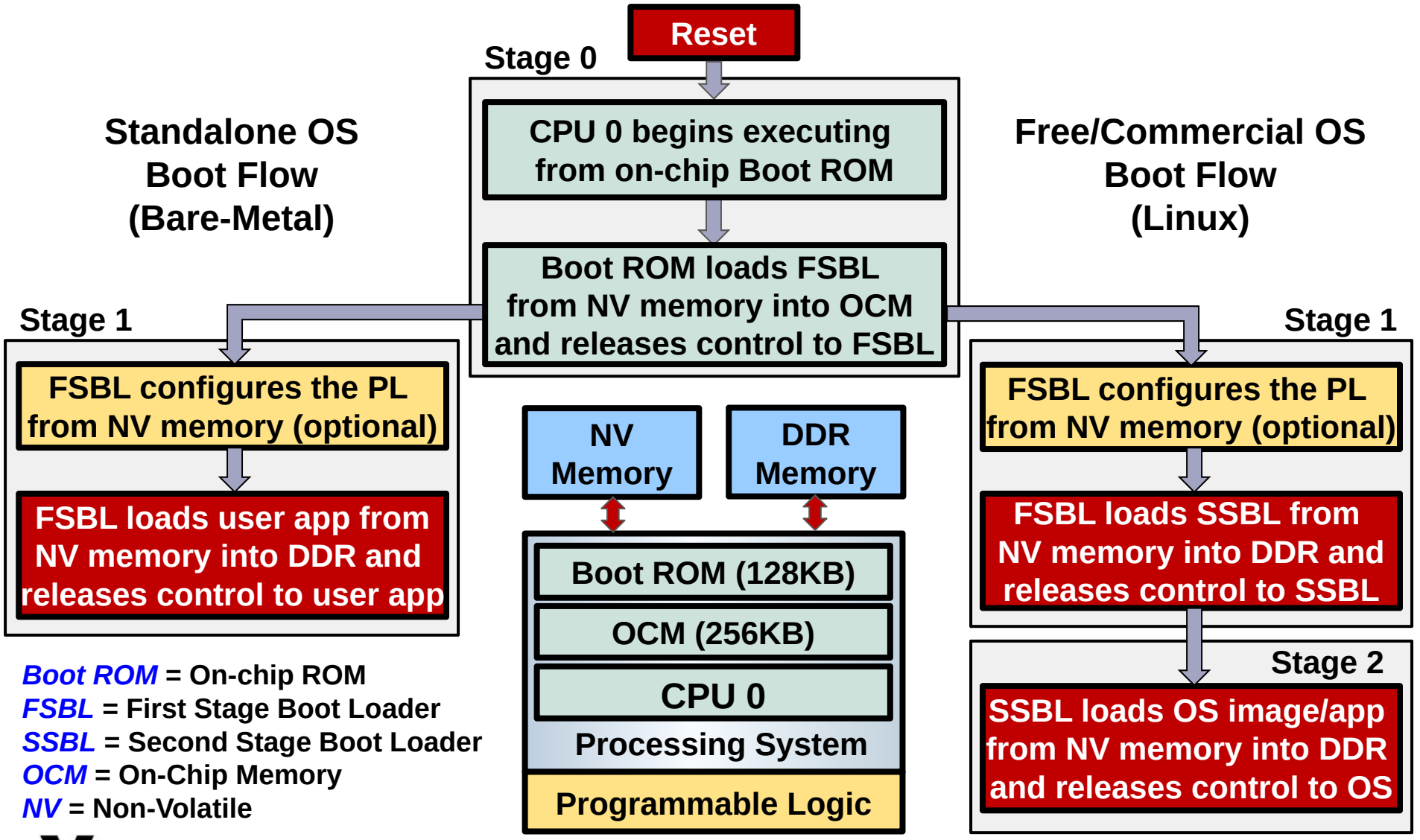
Boot ROM = On-chip ROM
FSBL = First Stage Boot Loader
SSBL = Second Stage Boot Loader
OCM = On-Chip Memory
NV = Non-Volatile

Typical Zynq Boot and Configuration Flow



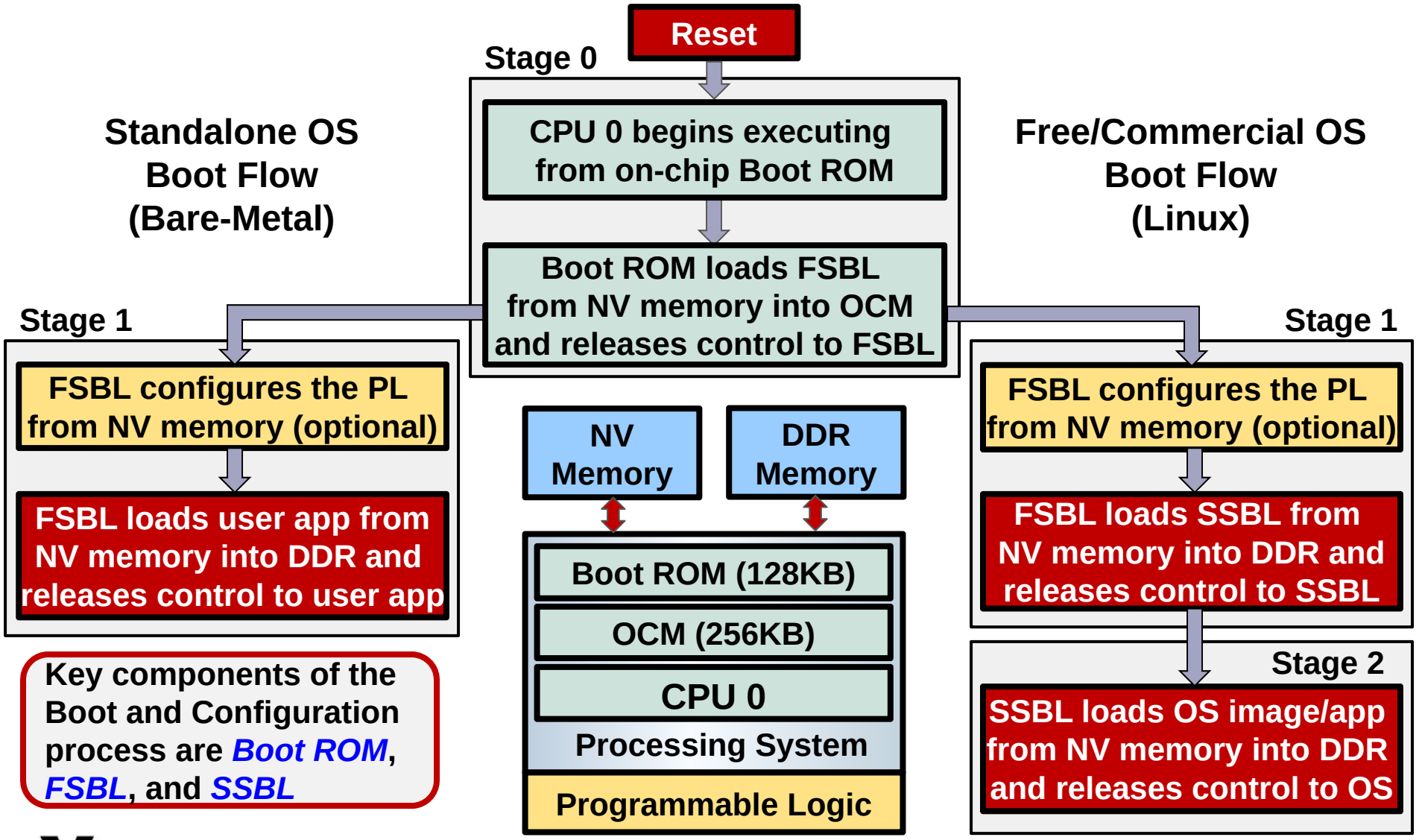
Boot ROM = On-chip ROM
FSBL = First Stage Boot Loader
SSBL = Second Stage Boot Loader
OCM = On-Chip Memory
NV = Non-Volatile

Typical Zynq Boot and Configuration Flow



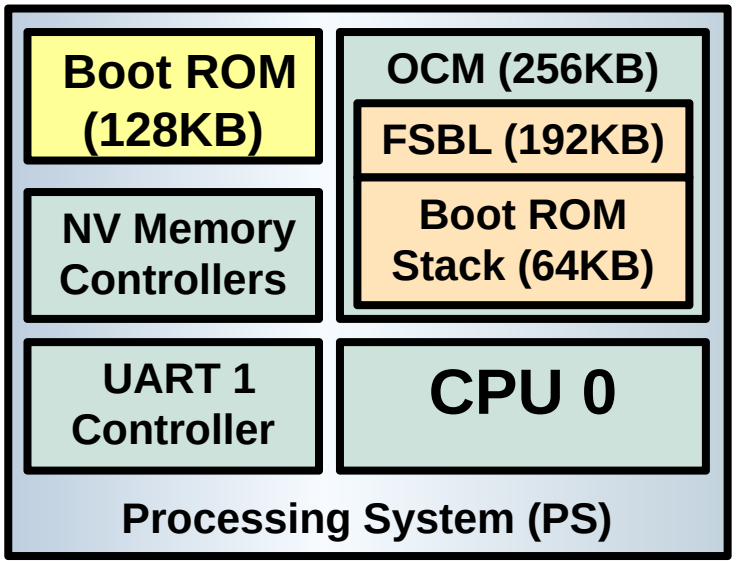
Boot ROM = On-chip ROM
FSBL = First Stage Boot Loader
SSBL = Second Stage Boot Loader
OCM = On-Chip Memory
NV = Non-Volatile

Typical Zynq Boot and Configuration Flow



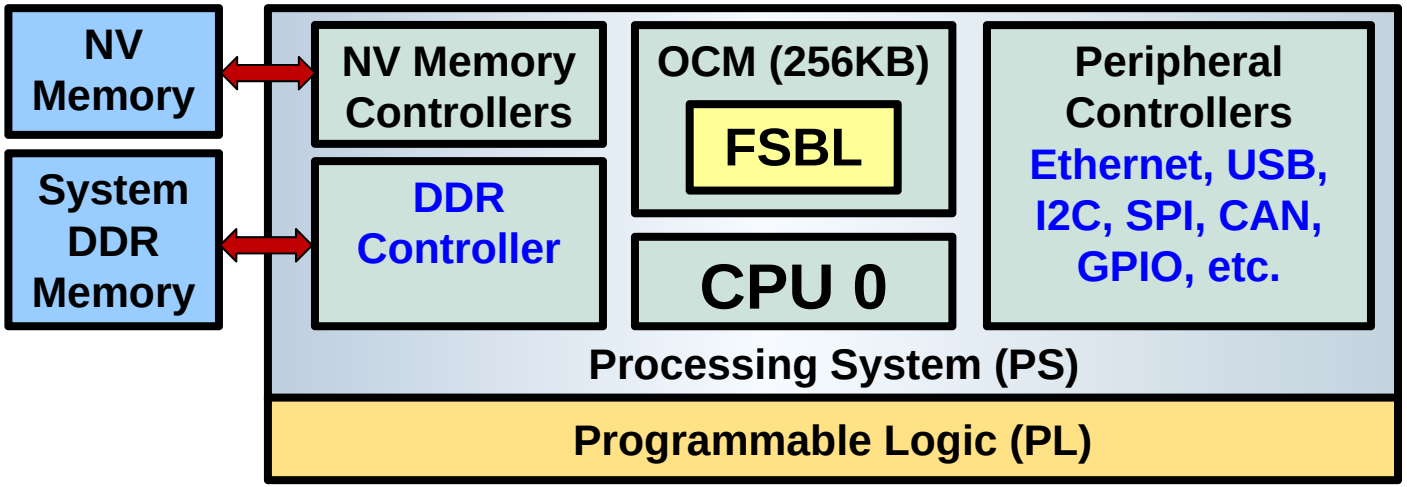
Zynq Boot ROM

- **Zynq PS includes a factory-programmed 128KB Boot ROM. On reset, Boot ROM performs several functions**
  - Initializes one of the NV memory controllers based on the *Boot Mode Pins*
    - *SD Card Boot* – SD 0 controller on MIO[40:45] pins
    - *QSPI Flash Boot* – QSPI 0 controller on MIO[1:6] pins
    - *NOR Flash Boot* – NOR controller on MIO[0:39] pins
    - *NAND Flash Boot* – NAND controller on MIO[0:14, 16:23] pins
  - Initializes UART1 on MIO[48:49] pins
  - Maps lower 192KB of OCM to 0x00 (FSBL code space) and upper 64KB to 0xFFFF_0000 (Boot ROM stack)
  - Loads the FSBL code from NV memory into the OCM and releases control to FSBL (max FSBL image size is 192KB)



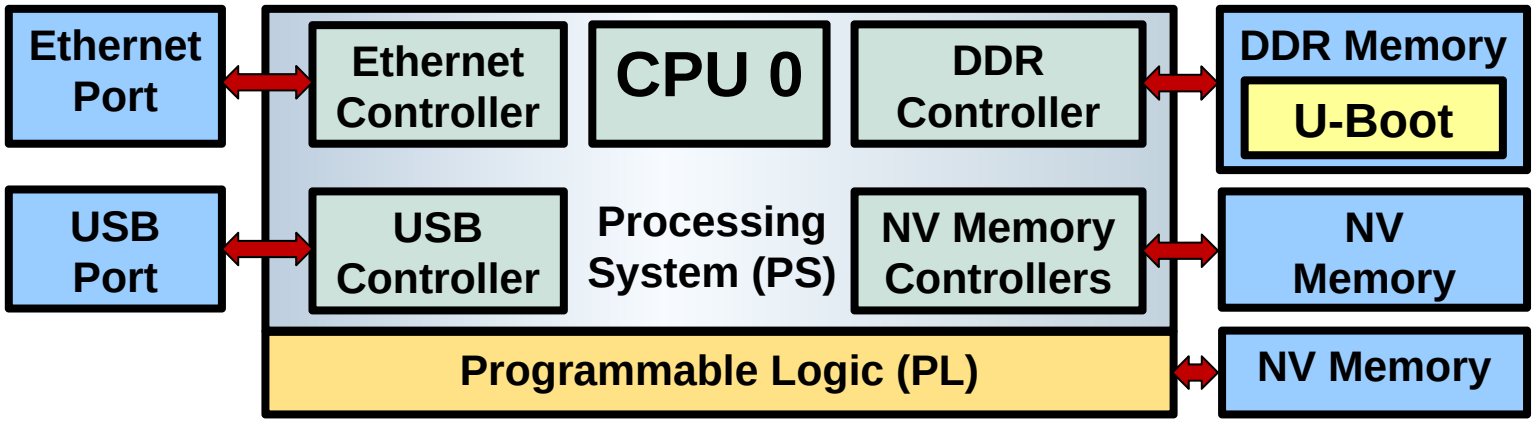
First Stage Boot Loader (FSBL)

- **FSBL is firmware source code provide by Xilinx and can be modified by users to perform additional tasks**
  - FSBL initializes PS peripherals/memory controllers and clocking blocks not initialized by the Boot ROM (Ethernet, USB, DDR, PLLs, etc.)
  - Maps the DDR to the 0x0010_0000 – 0x3FFF_FFFF address space
  - Loads the application code or SSBL from NV memory into the DDR
  - FSBL can optionally configure the PL or load the OS image/application



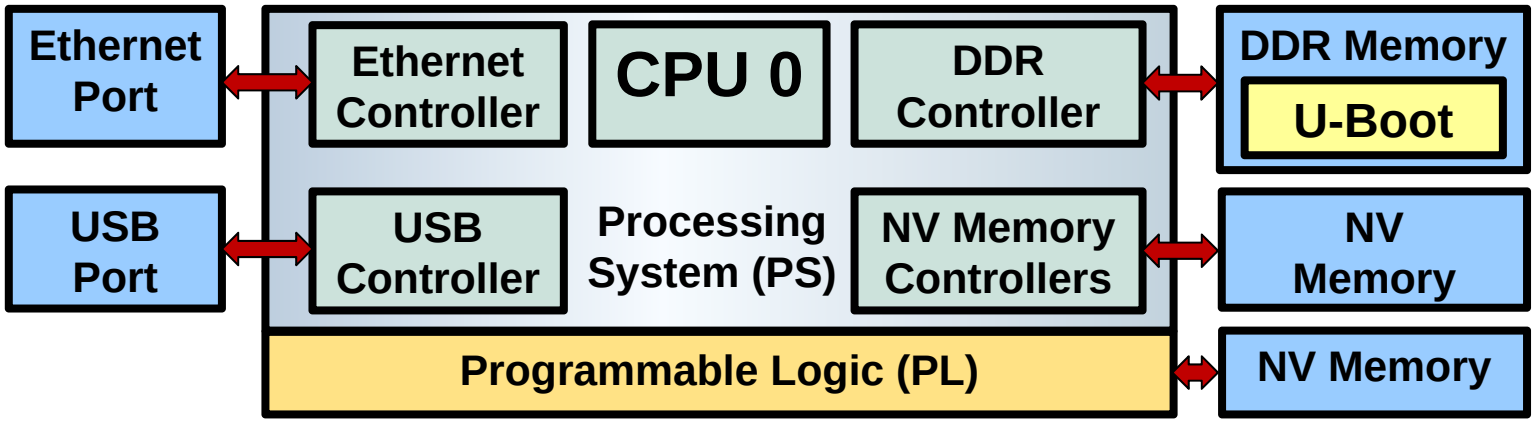
Second Stage Boot Loader (SSBL)

- **SSBL is responsible for loading the OS image and application into the system DDR memory**
  - Open source Universal Boot Loader (U-Boot) is an example of SSBL used for loading Linux OS image/application into the system memory
  - U-Boot can load the OS image/application from NV memory (connected to PS or PL), Ethernet, or USB port
  - Optionally, U-Boot can configure the Programmable Logic



Second Stage Boot Loader (SSBL)

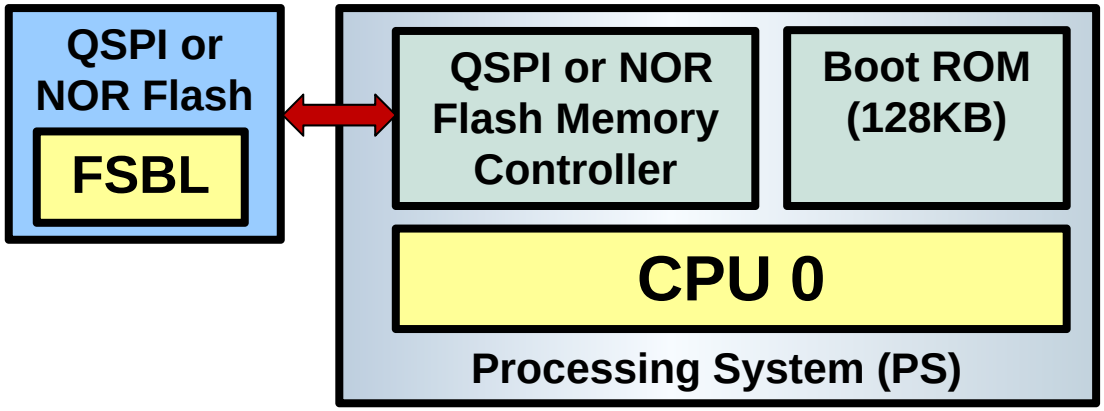
- **SSBL is responsible for loading the OS image and application into the system DDR memory**
  - Open source Universal Boot Loader (U-Boot) is an example of SSBL used for loading Linux OS image/application into the system memory
  - U-Boot can load the OS image/application from NV memory (connected to PS or PL), Ethernet, or USB port
  - Optionally, U-Boot can configure the Programmable Logic



Xilinx U-Boot Source - https://github.com/Xilinx/u-boot-xlnx
Instructions to Build U-Boot - www.wiki.xilinx.com/Build+U-Boot

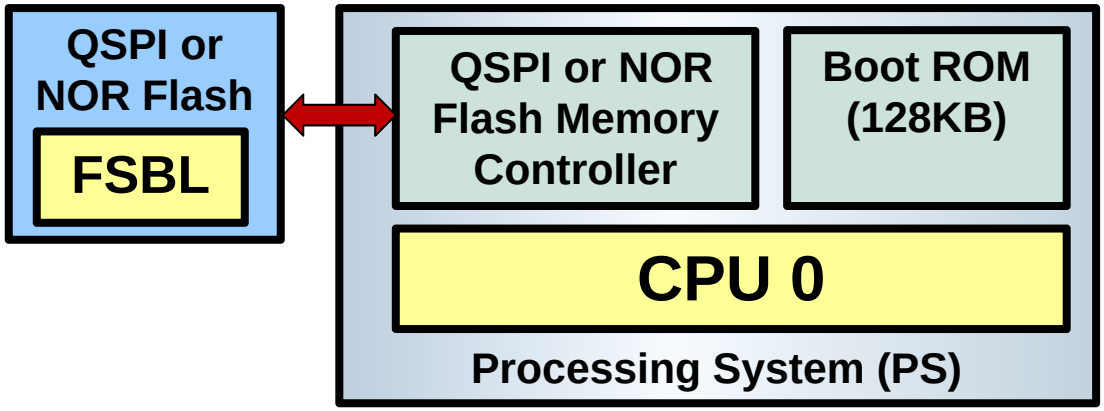
Execute-in-Place (XIP)

- **Typically, Boot ROM loads the FSBL from NV memory into the OCM and releases control to FSBL**
  - If the XIP feature is enabled in the boot image header, FSBL is executed directly from QSPI or NOR Flash in non-secure boot mode
    - Eliminates the need for Boot ROM to load the FSBL into the OCM
    - The FSBL maximum image size requirement of 192KB is removed
    - XIP feature is not supported for SD card or NAND Flash boot modes



Execute-in-Place (XIP)

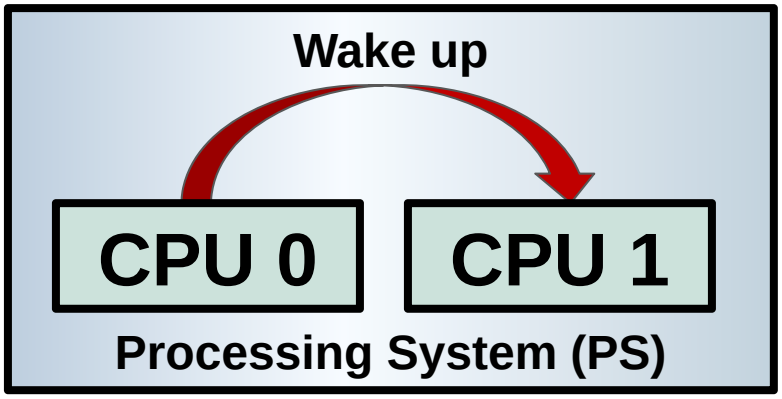
- **Typically, Boot ROM loads the FSBL from NV memory into the OCM and releases control to FSBL**
  - If the XIP feature is enabled in the boot image header, FSBL is executed directly from QSPI or NOR Flash in non-secure boot mode
    - Eliminates the need for Boot ROM to load the FSBL into the OCM
    - The FSBL maximum image size requirement of 192KB is removed
    - XIP feature is not supported for SD card or NAND Flash boot modes



A complete reference design using the XIP feature can be found at
http://www.wiki.xilinx.com/Zynq-7000+AP+SoC+DDRLess+System+Tech+Tip

Waking up Zynq CPU 1

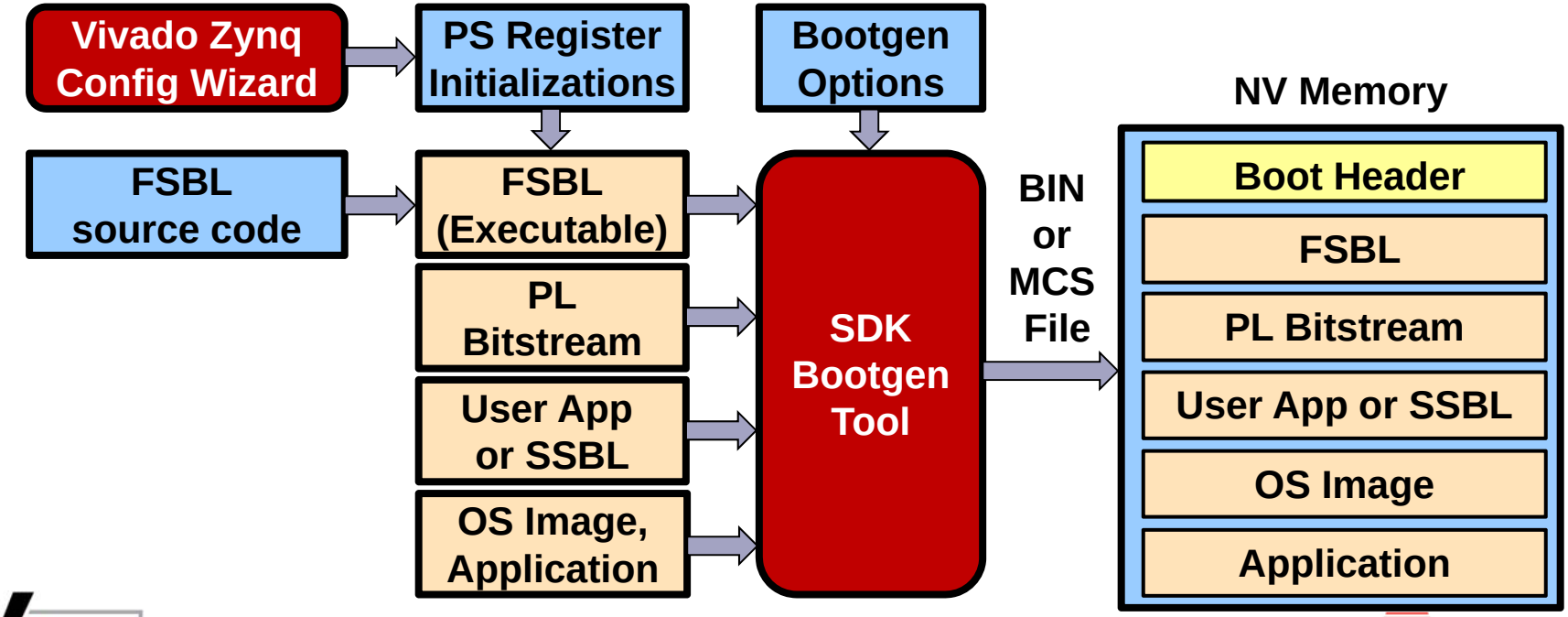
- **After reset, CPU 1 is in the idle state waiting for a wake up signal**
  - Boot ROM places CPU 1 in a low power *Wait For Event (WFE)* state
  - CPU 0 writes the CPU 1 starting instruction to the memory location 0xFFFFFFF0 and executes the *Send Event (SEV)* instruction
  - When CPU 1 receives the *CPU 0 SEV instruction*, it immediately reads the instruction at address 0xFFFFFFF0 and executes it
    - Typically, this is a jump instruction to the CPU 1 application



Non-Secure Boot and Configuration

Master Non-Secure Boot Image Generation Flow

- **Xilinx SDK Bootgen tool is used to generate the boot image stored in the non-volatile memory**
  - Bootgen creates a *Boot Header* describing the FSBL partition
    - All other boot partitions are described in *Partition Headers*
  - SD card boot image must be called *Boot.BIN*
  - The *MCS* file type is typically used when booting from a Flash device
  - Xilinx *SDK* or *U-Boot* is used to program the boot Flash device



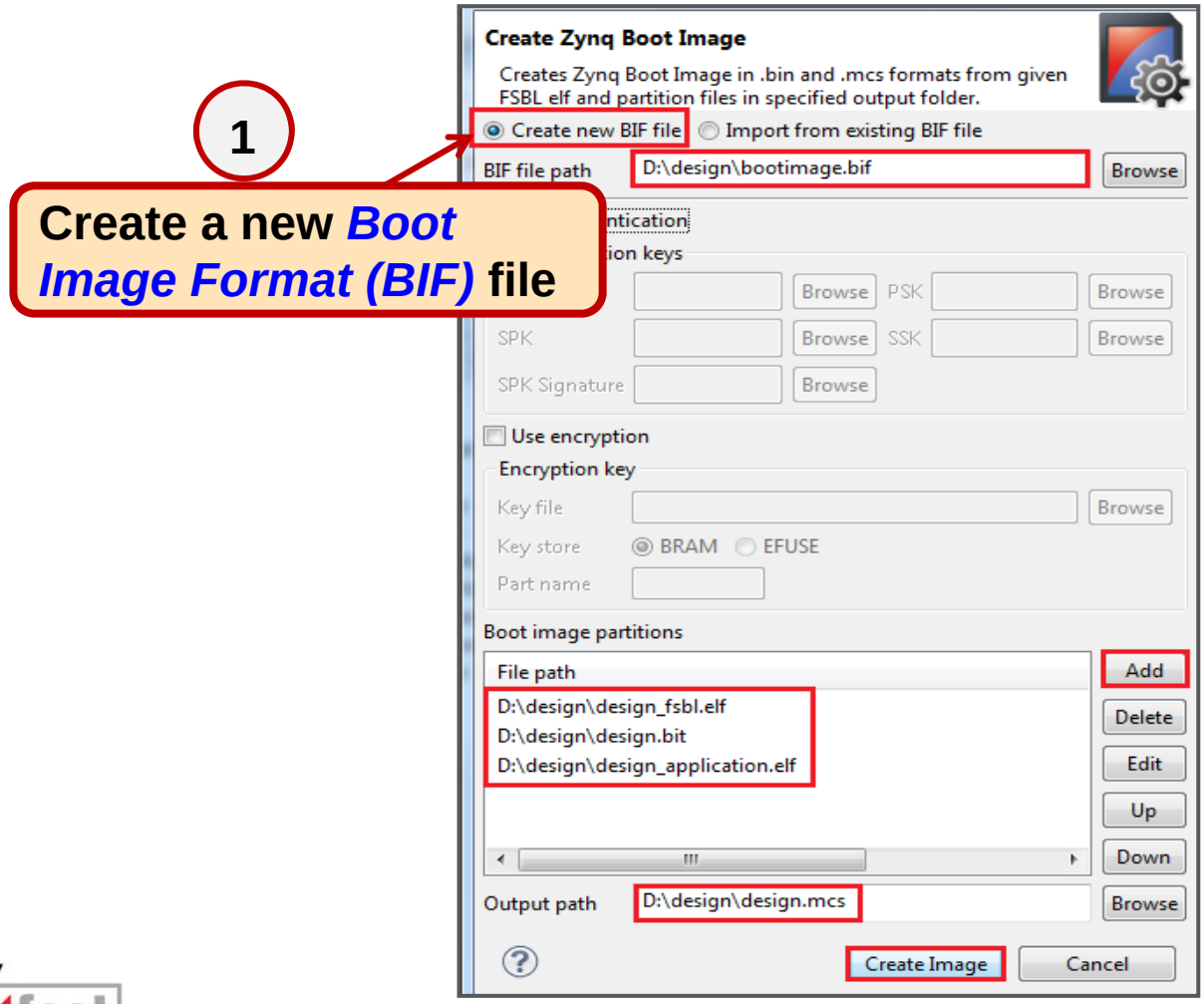
Boot Header

- **Boot Header is required for all master boot methods**
  - It occupies the first 2240 bytes (0x000 – 0x8BF) of the boot image
  - A subset of the *Boot Header* fields are shown below

Fields	Header Offset	Contents
Width Detection	0x020	0xAA995566 = Single QSPI 0xACCA50AF = Dual QSPI
Image Identification	0x024	0x584C4E58 ('XLNX') indicates a valid Boot Header
Encryption Status	0x028	0xA5C3C5A3 = eFUSE 0x3A5C3C5A = BBRAM 0x000000000 = Not encrypted
Length of Image	0x034	FSBL image size to be copied. 0x0 = XIP feature is enabled
Register Initialization	0x0A0 – 0x89C	These 2048 bytes can be used to initialize up to 256 PS control registers prior to the FSBL load

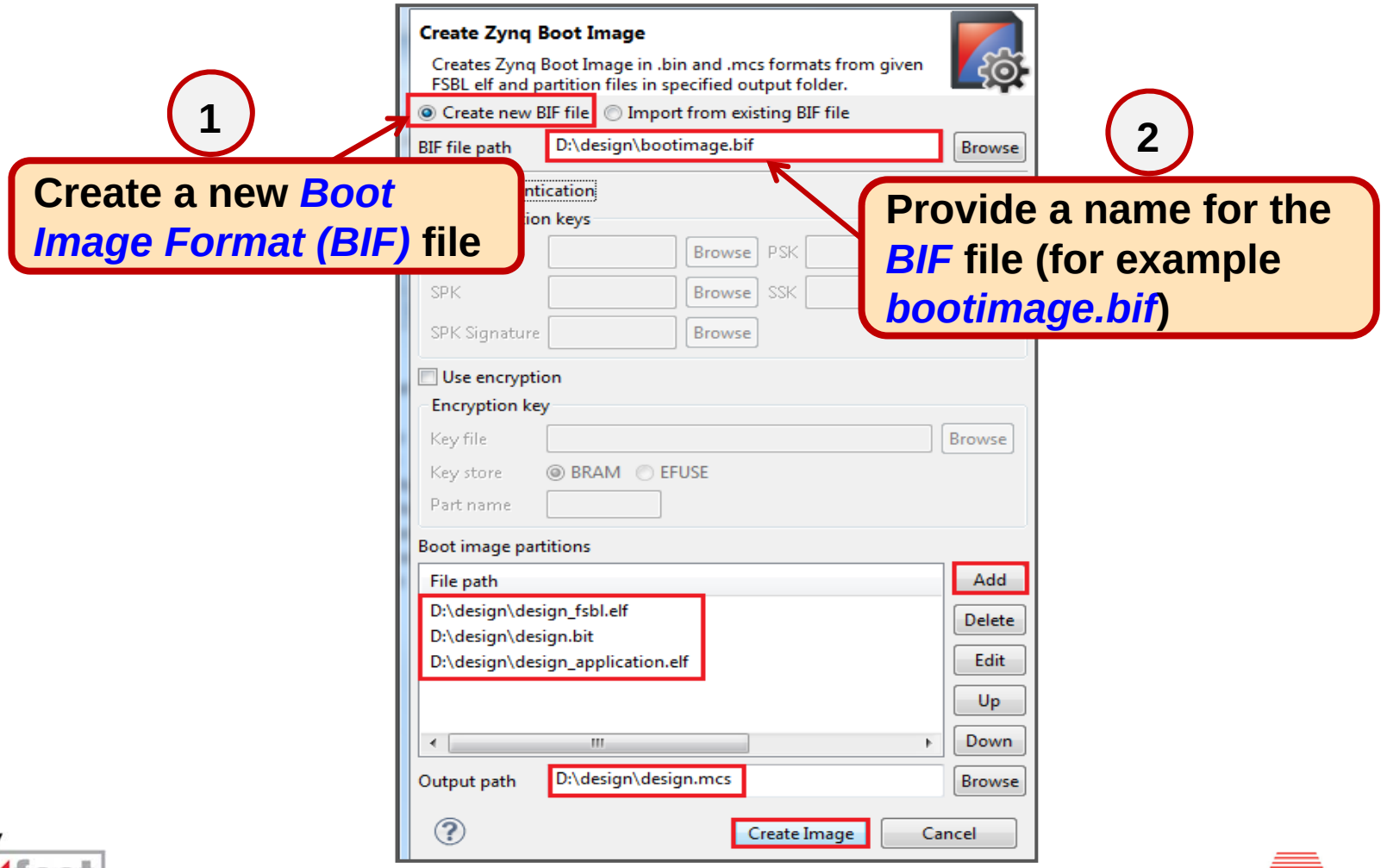
Non-Secure Boot Image Generation (Standalone OS)

- Invoke the ***Bootgen*** tool via ***Xilinx Tools > Create Zynq Boot Image*** from the SDK GUI



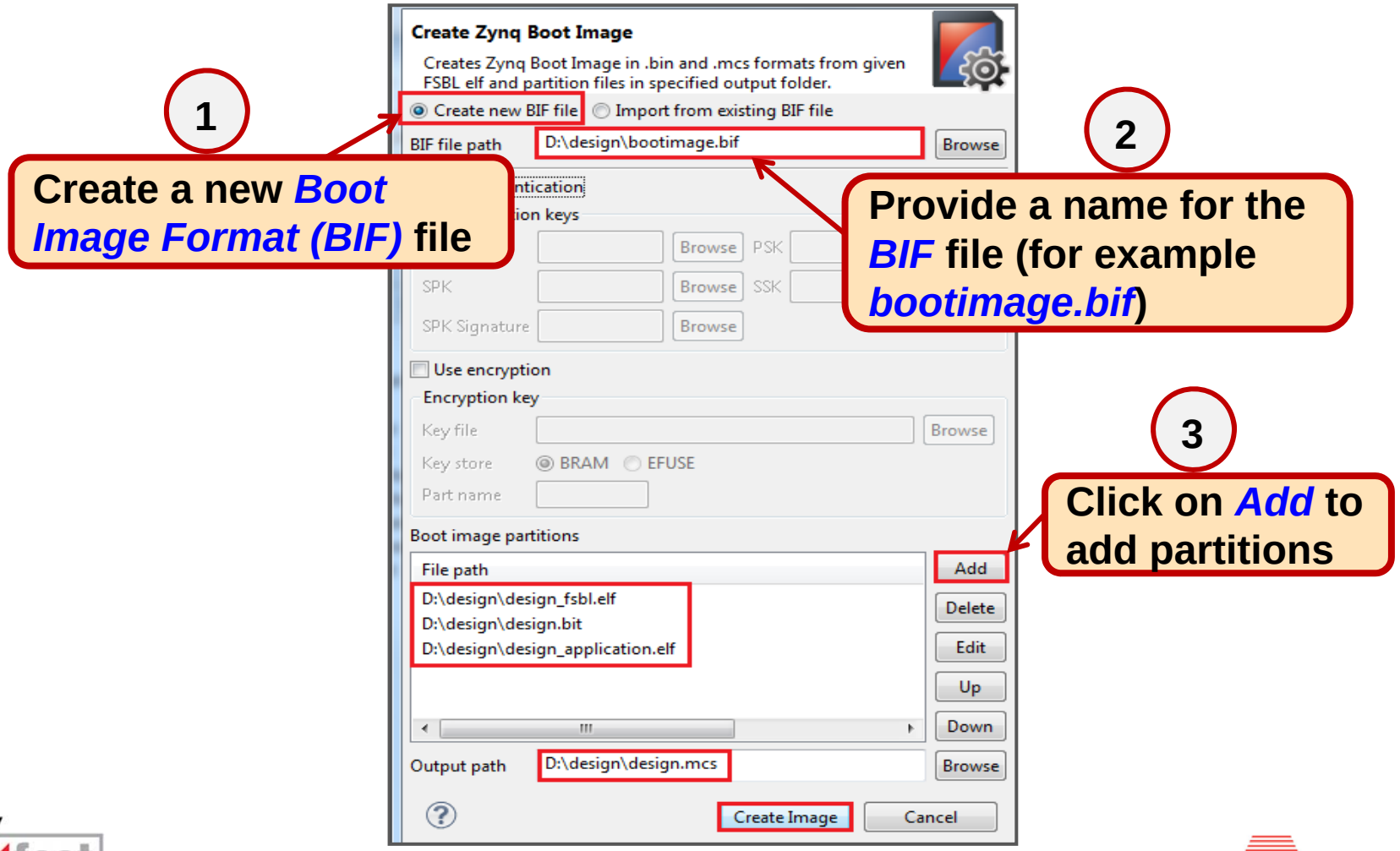
Non-Secure Boot Image Generation (Standalone OS)

- Invoke the ***Bootgen*** tool via ***Xilinx Tools > Create Zynq Boot Image*** from the SDK GUI



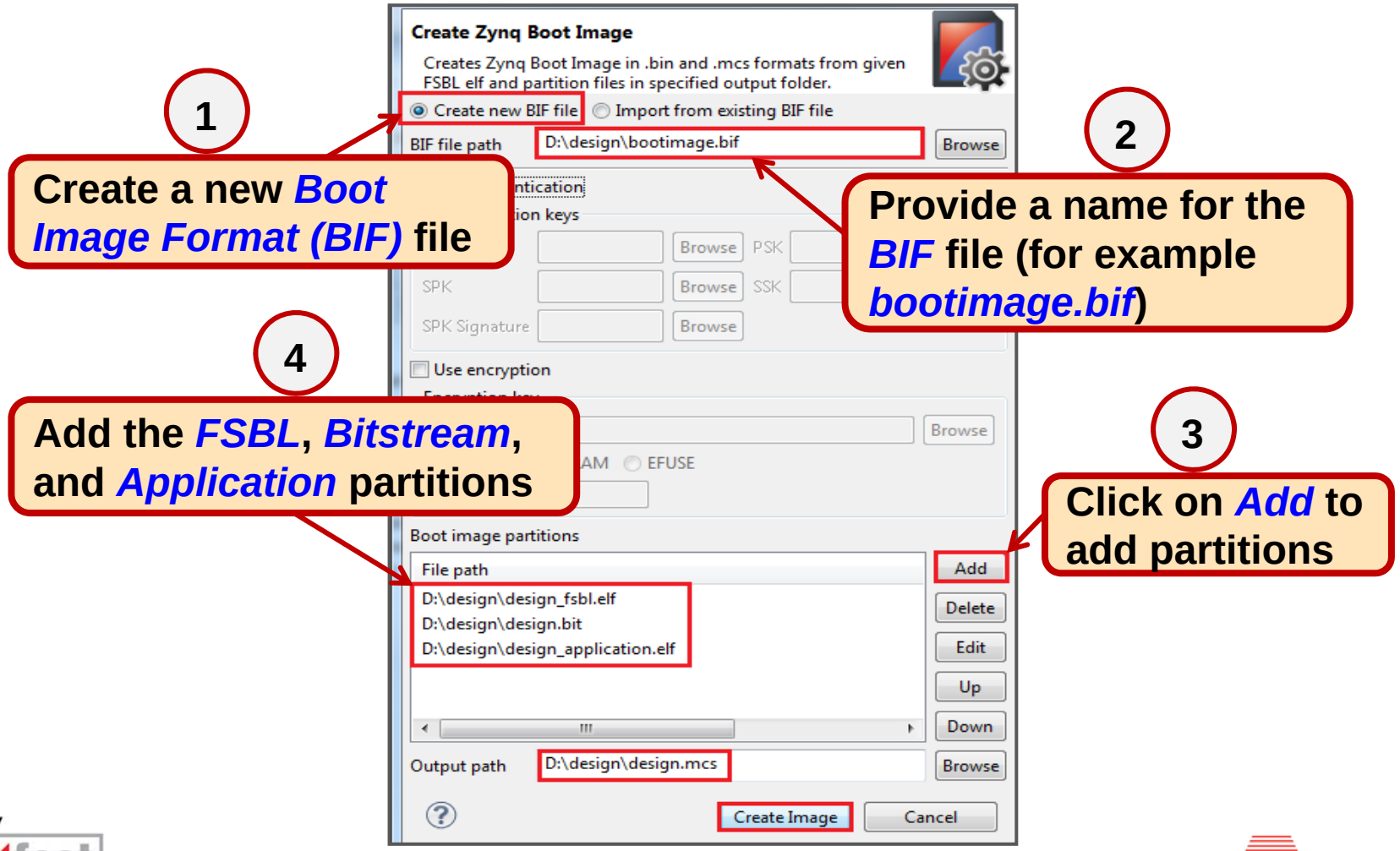
Non-Secure Boot Image Generation (Standalone OS)

- Invoke the ***Bootgen*** tool via ***Xilinx Tools > Create Zynq Boot Image*** from the SDK GUI



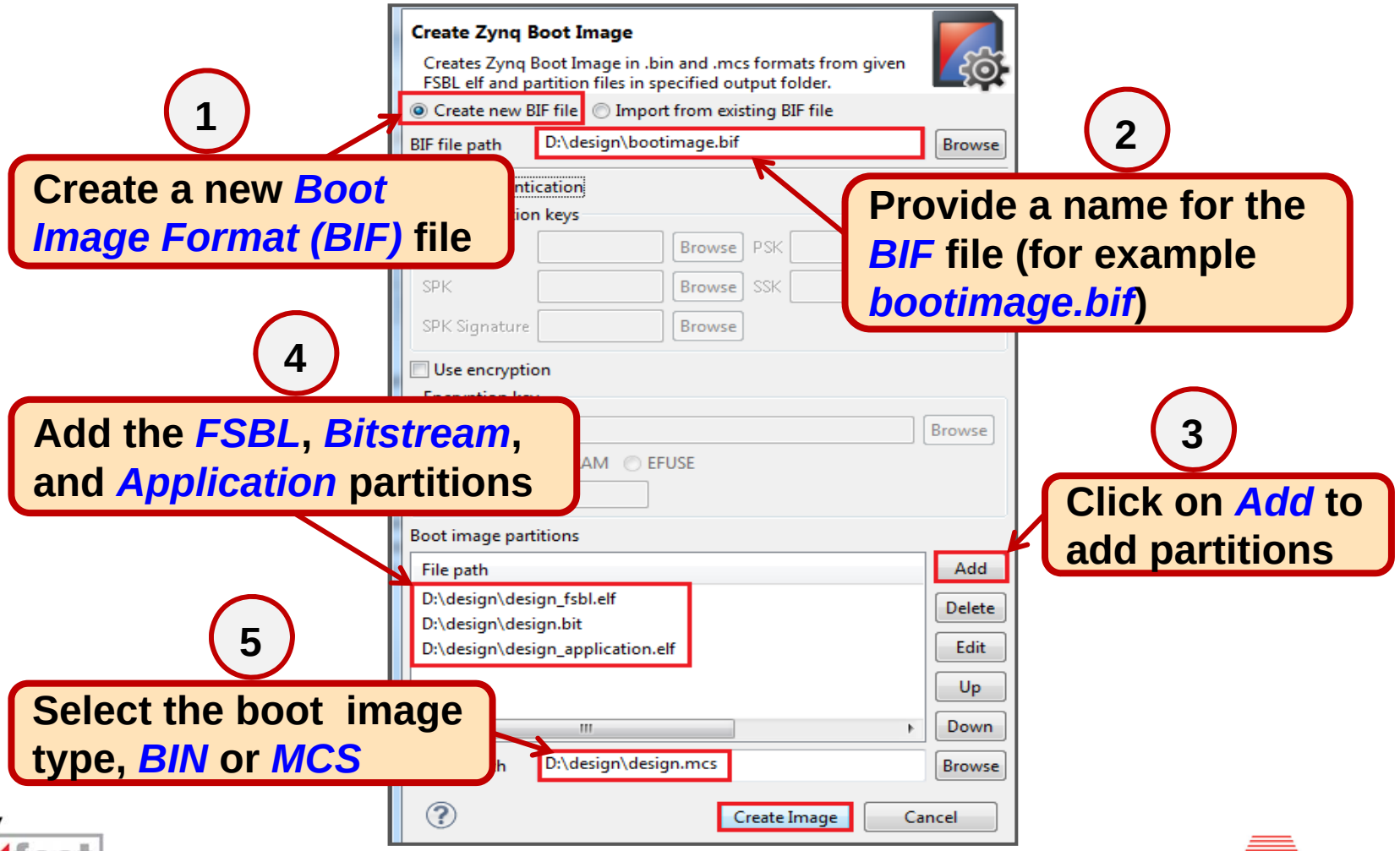
Non-Secure Boot Image Generation (Standalone OS)

- Invoke the *Bootgen* tool via *Xilinx Tools > Create Zynq Boot Image* from the SDK GUI



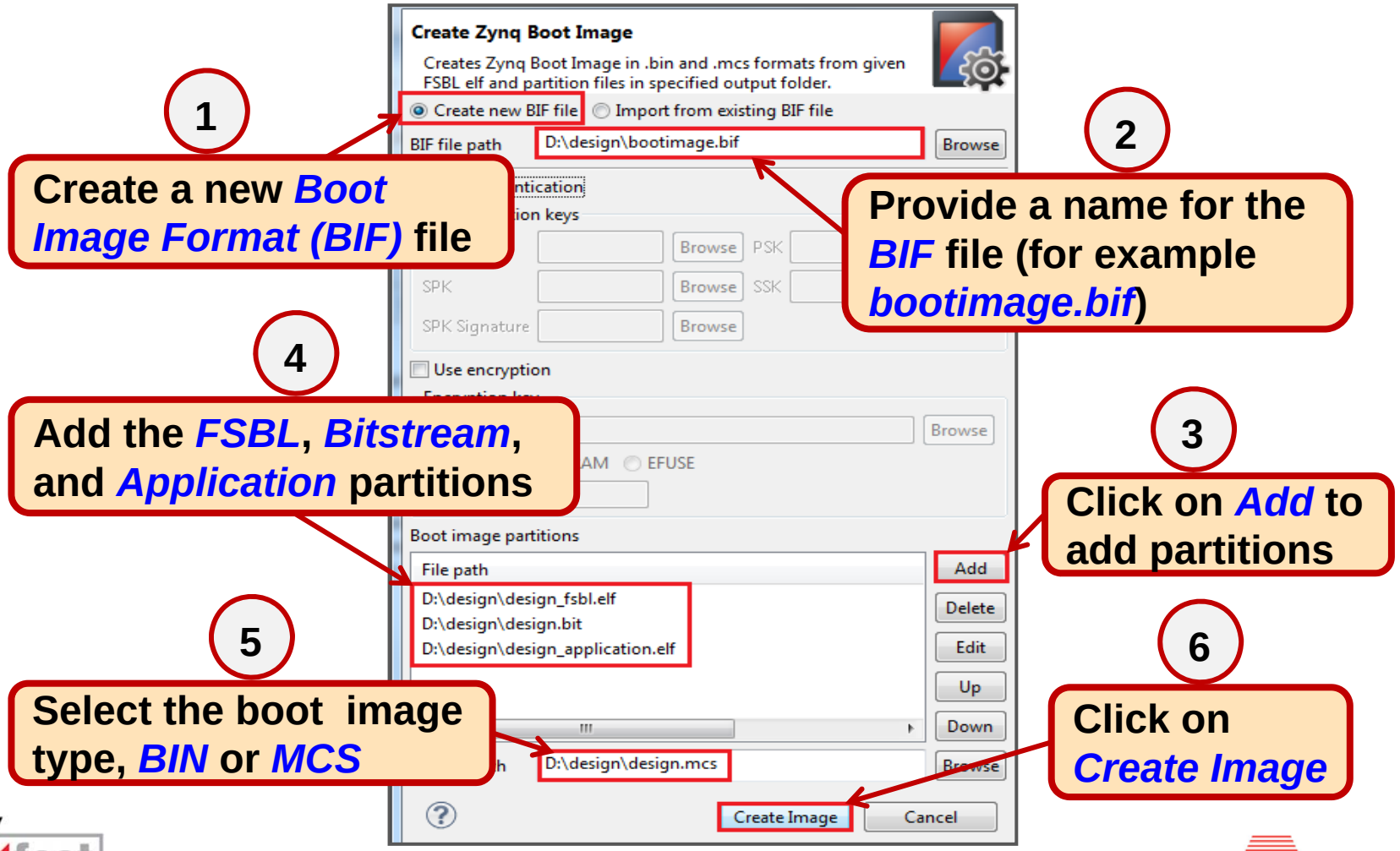
Non-Secure Boot Image Generation (Standalone OS)

- Invoke the *Bootgen* tool via *Xilinx Tools > Create Zynq Boot Image* from the SDK GUI



Non-Secure Boot Image Generation (Standalone OS)

- Invoke the *Bootgen* tool via *Xilinx Tools > Create Zynq Boot Image* from the SDK GUI



Non-Secure SD Card Boot Image Generation (Linux OS)

- **Example of using Bootgen in command line mode to generate a boot image for a Linux system booting from SD card**
  - Create a *Boot Image Format* (BIF) file describing the image partitions (for example, *bootimage.bif* file with the following contents)

```
image:
      {
        [bootloader] fsbl.elf
        system.bit
        u-boot.elf
      }
```

  - Use the following Bootgen command to generate the *boot.bin* boot image

```
bootgen -image bootimage.bif -o boot.bin
```

  - Copy the *boot.bin*, Linux OS image (*uImage*, *devicetree.dtb*, *uramdisk.image.gz*), and *application* to the root directory of the SD card

Boot Header Register Initialization Procedure

- **The set of registers to be initialized via *Boot Header* are included in the Bootgen flow using a *.INIT* input text file**
  - The register initialization syntax consists of an *operation directive* followed by the *register address* and *register data* and terminated with a *semicolon*

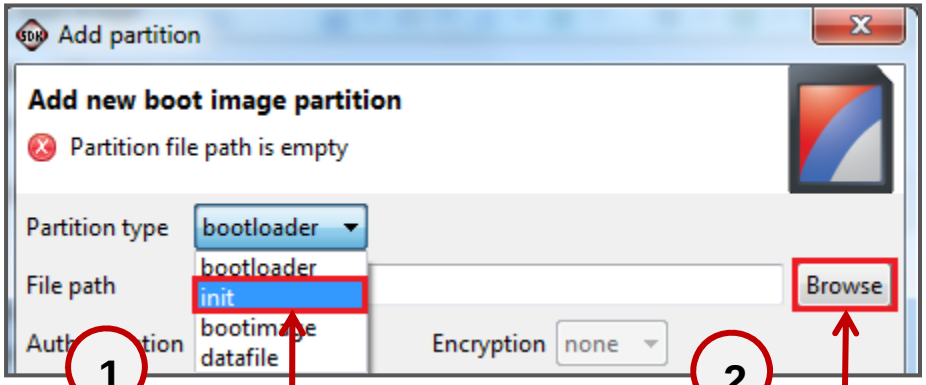
.set. <register_address> = <register_data>;

Boot Header Register Initialization Procedure

- **The set of registers to be initialized via *Boot Header* are included in the Bootgen flow using a *.INIT* input text file**
  - The register initialization syntax consists of an *operation directive* followed by the *register address* and *register data* and terminated with a *semicolon*

.set. <register_address> = <register_data>;

  - The *.INIT* file can be specified in the Bootgen GUI or in the *.BIF* file



Use the drop-down menu and select the *init* option

Browse to the *.INIT* text file

Boot Header Register Initialization Procedure

- **The set of registers to be initialized via *Boot Header* are included in the Bootgen flow using a *.INIT* input text file**
  - The register initialization syntax consists of an *operation directive* followed by the *register address* and *register data* and terminated with a *semicolon*

.set. <register_address> = <register_data>;

  - The *.INIT* file can be specified in the Bootgen GUI or in the *.BIF* file

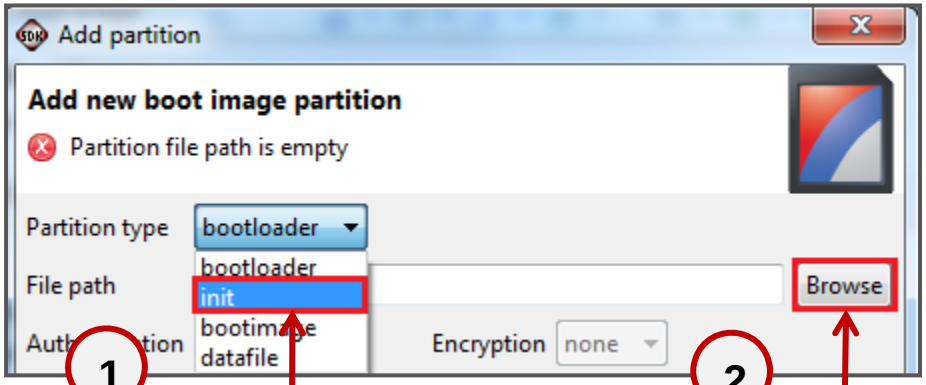
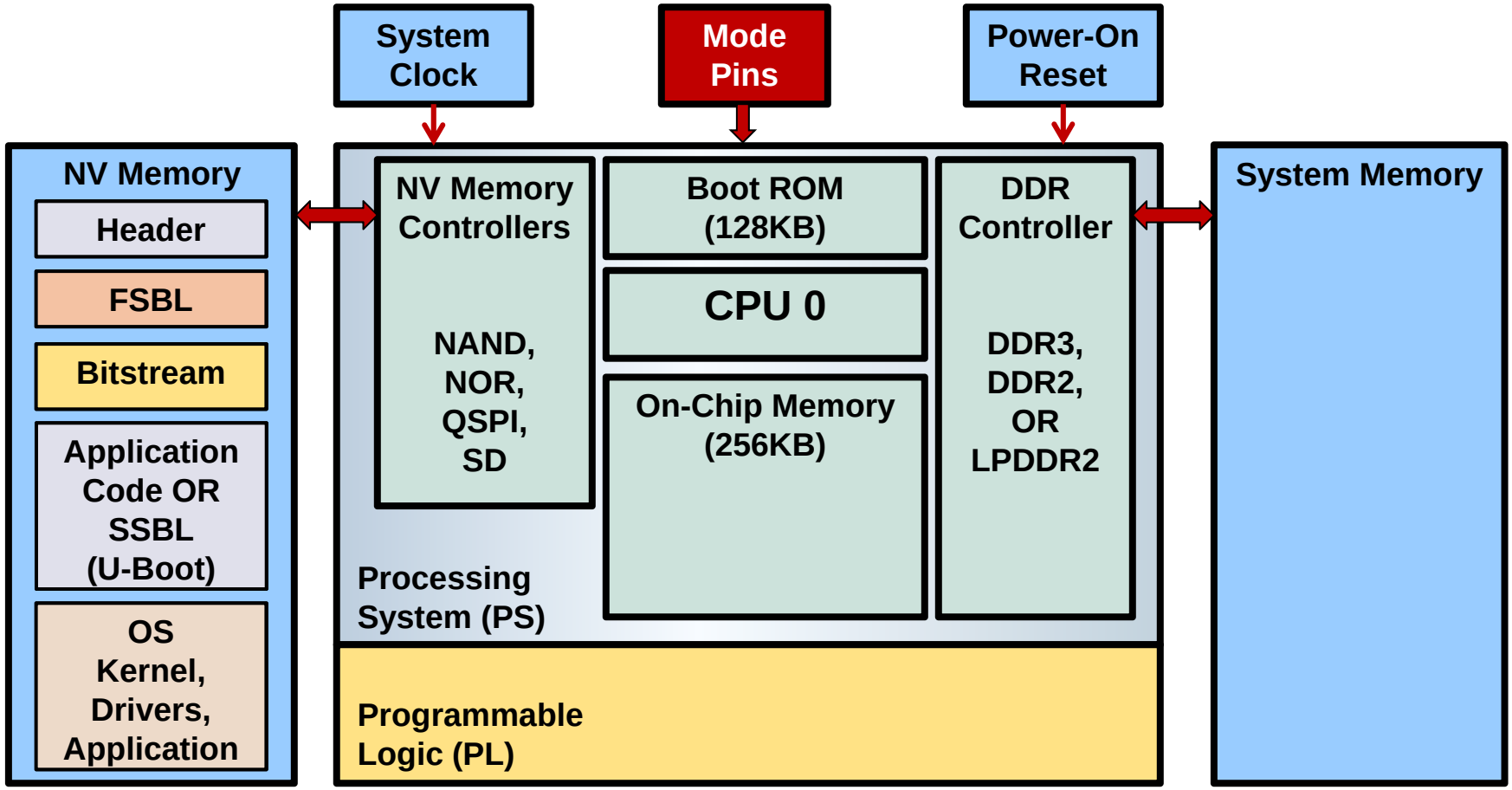


image: {
[INIT] my_regs.INIT
[bootloader] fsbl.elf
system.bit
u-boot.elf }

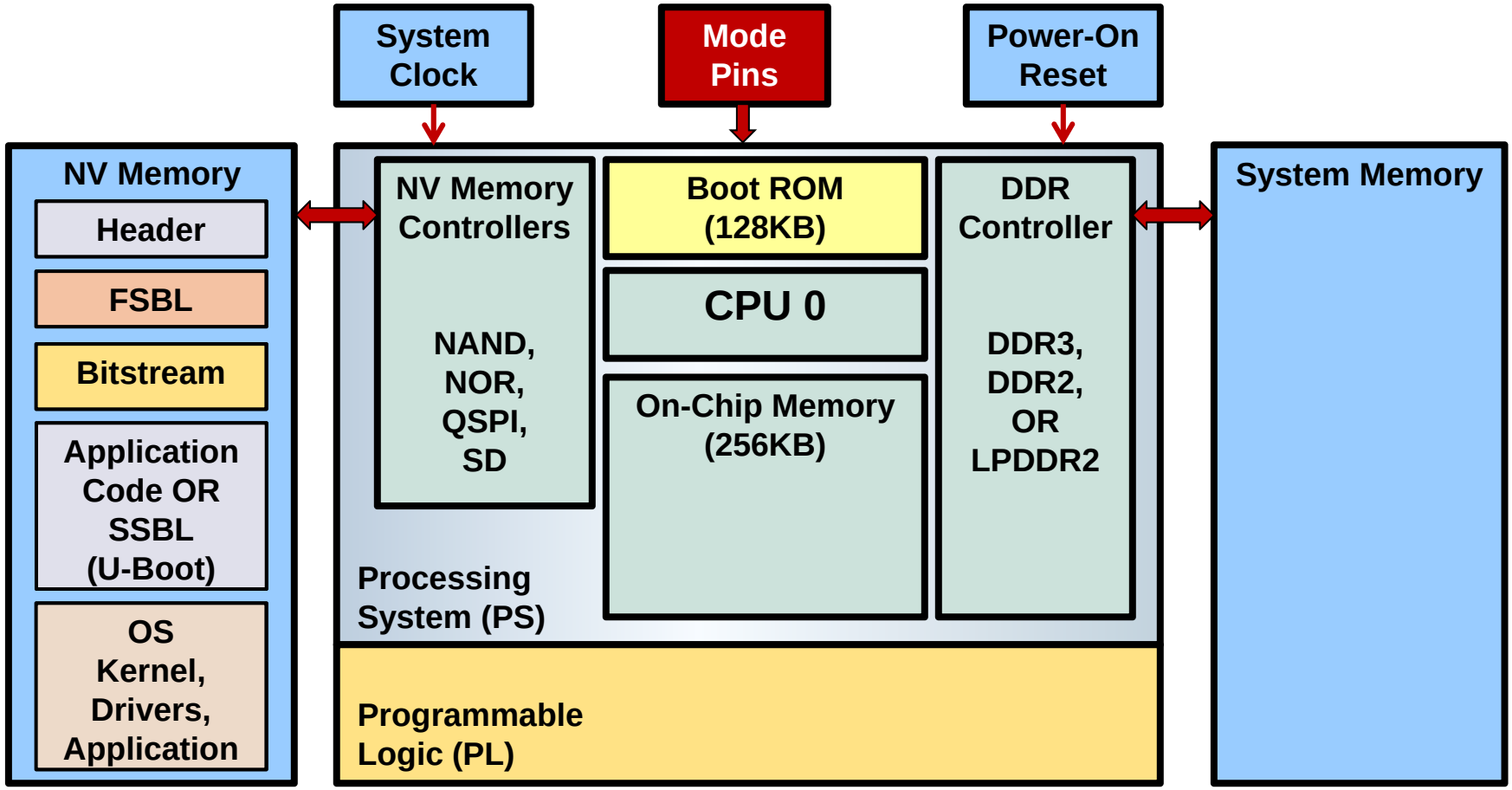
Use the drop-down menu and select the *init* option

Browse to the *.INIT* text file

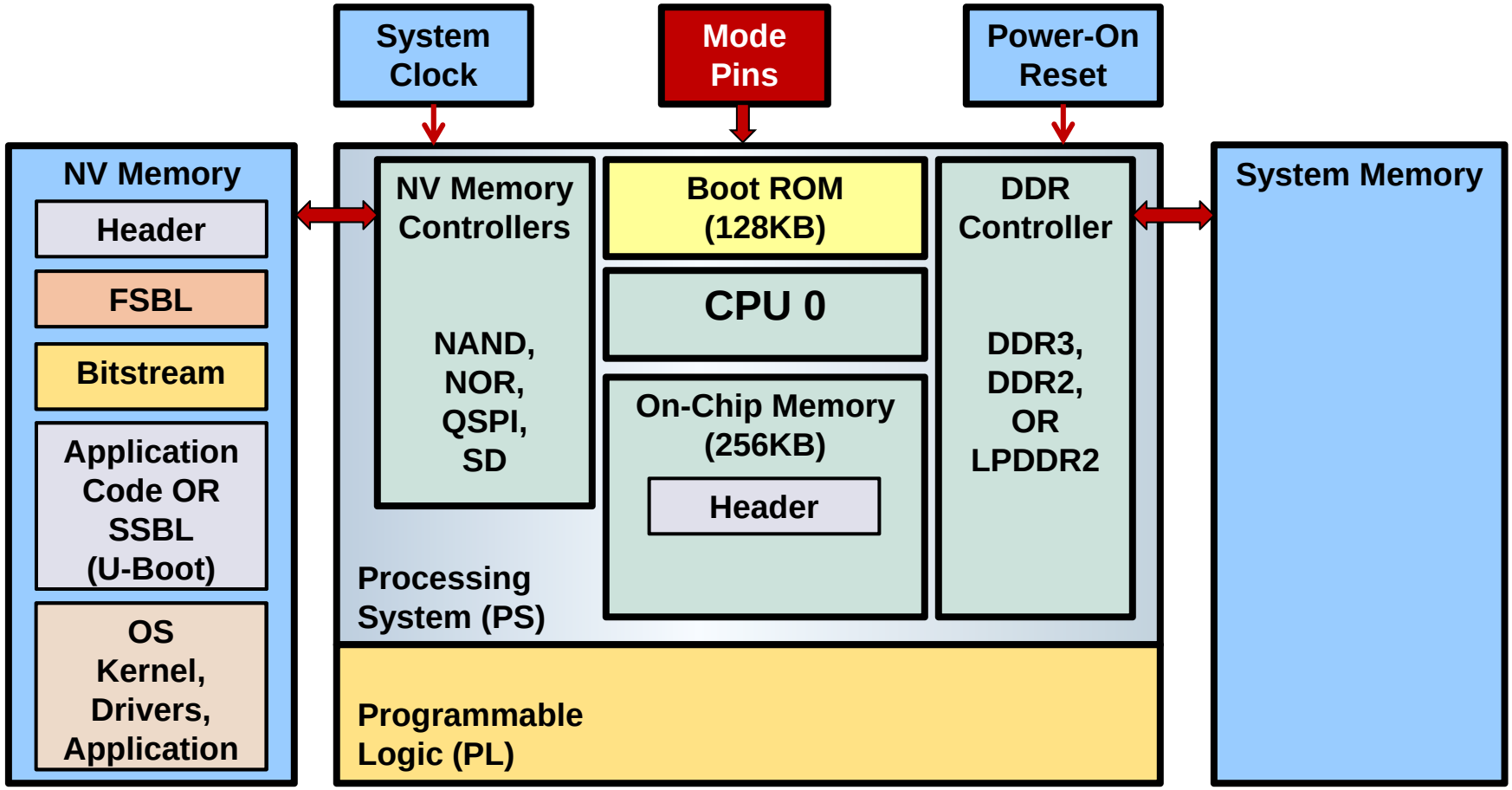
Typical Master Non-Secure Boot and Configuration Flow



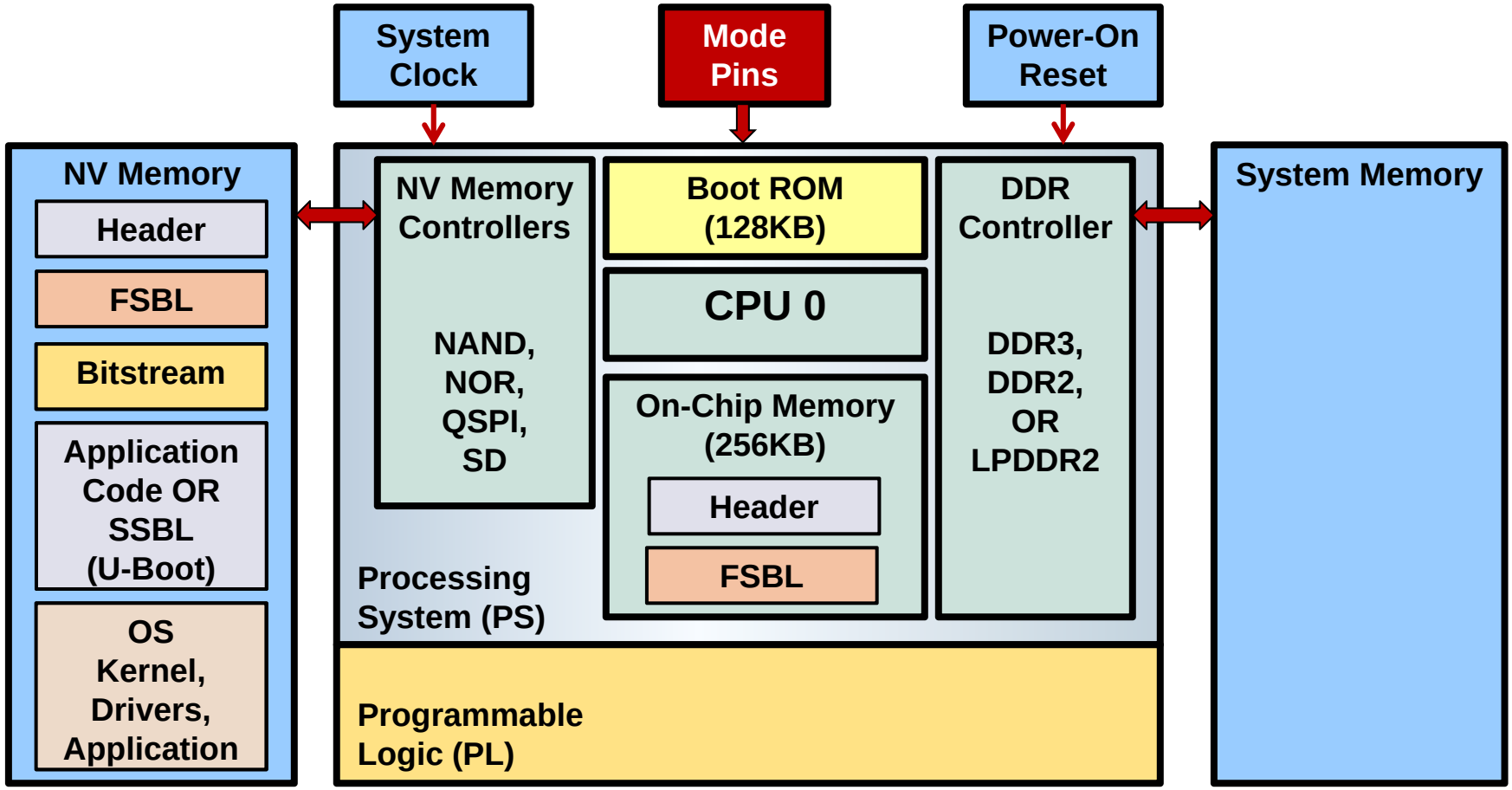
Typical Master Non-Secure Boot and Configuration Flow



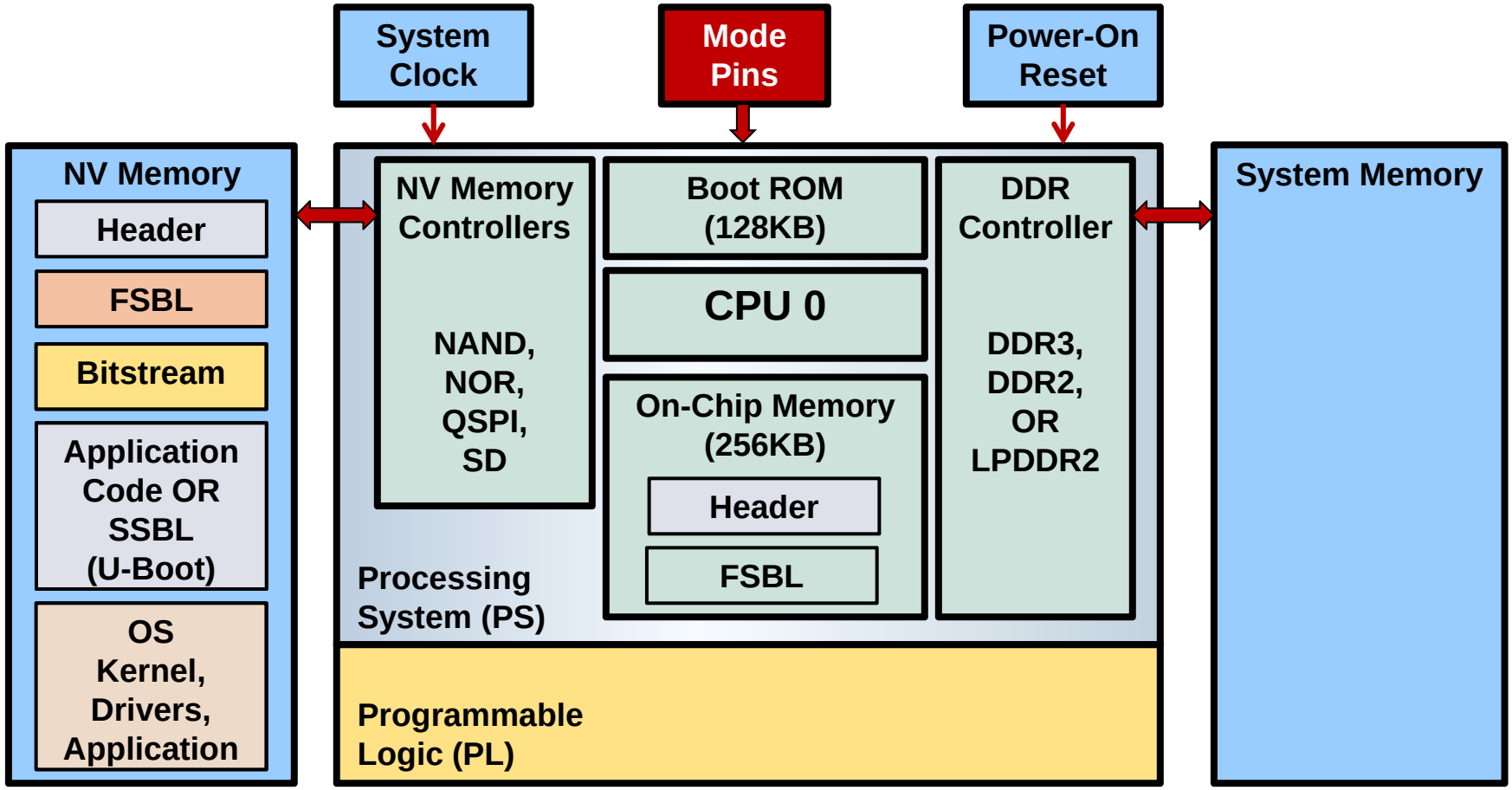
Typical Master Non-Secure Boot and Configuration Flow



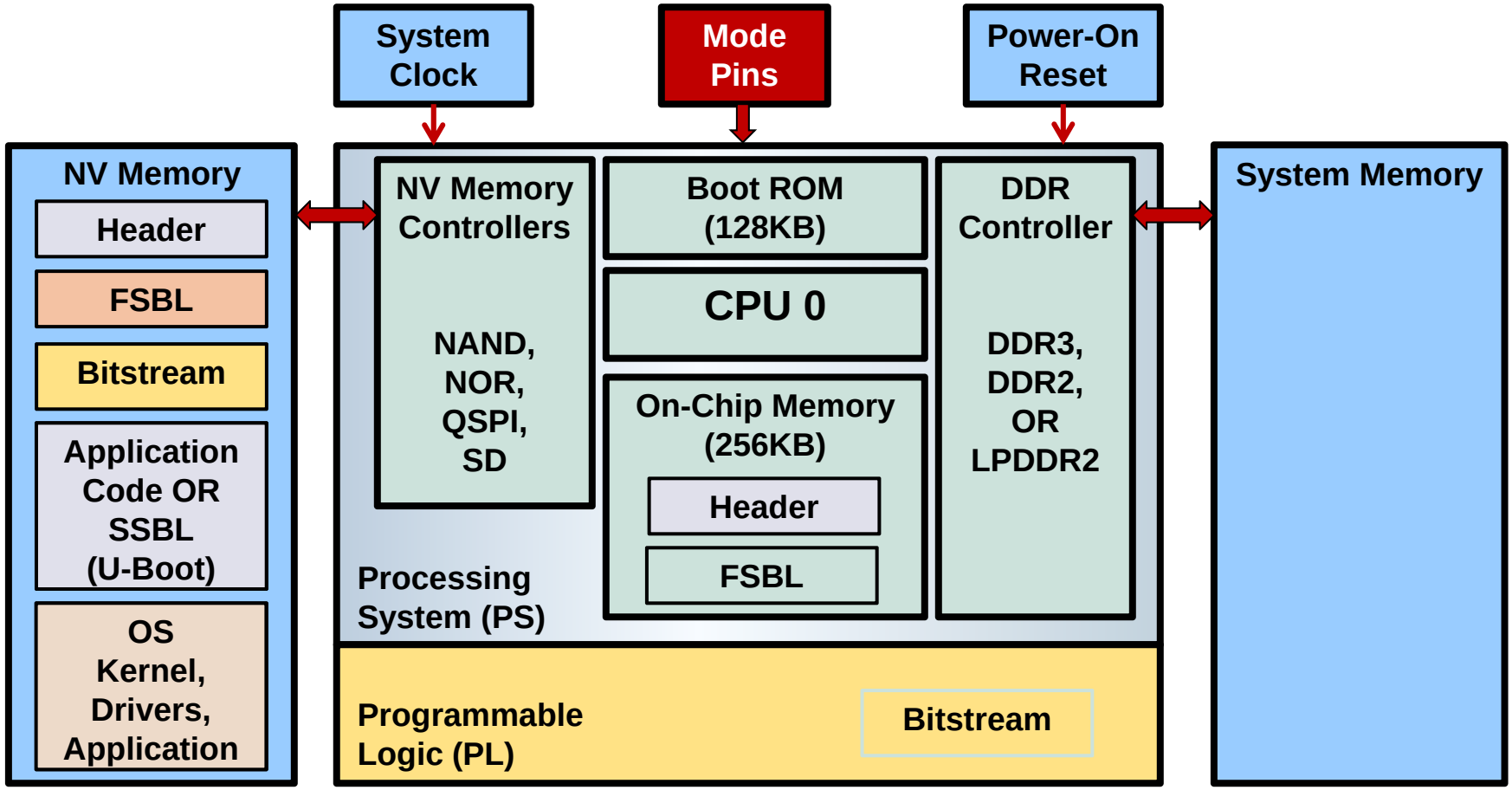
Typical Master Non-Secure Boot and Configuration Flow



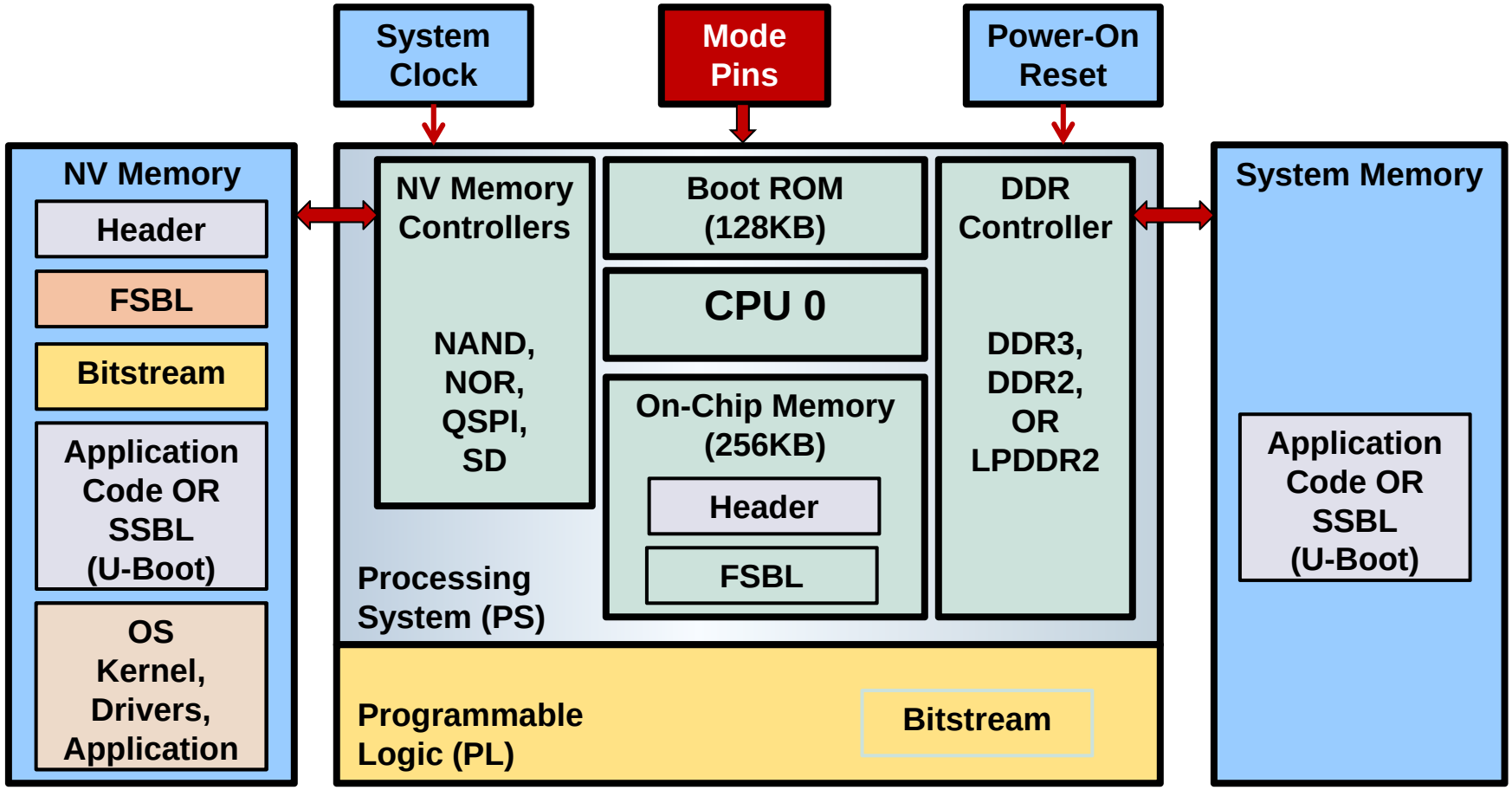
Typical Master Non-Secure Boot and Configuration Flow



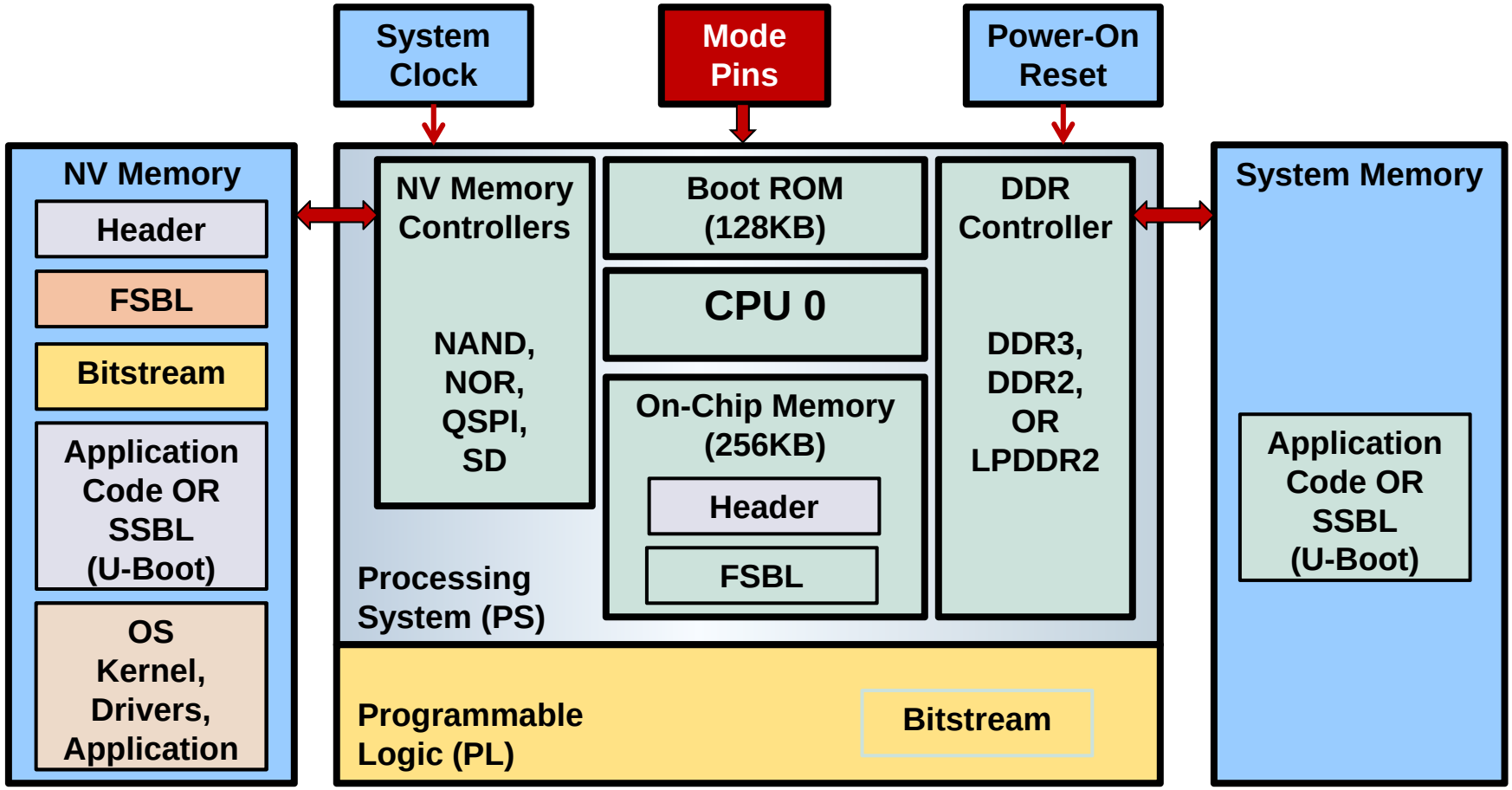
Typical Master Non-Secure Boot and Configuration Flow



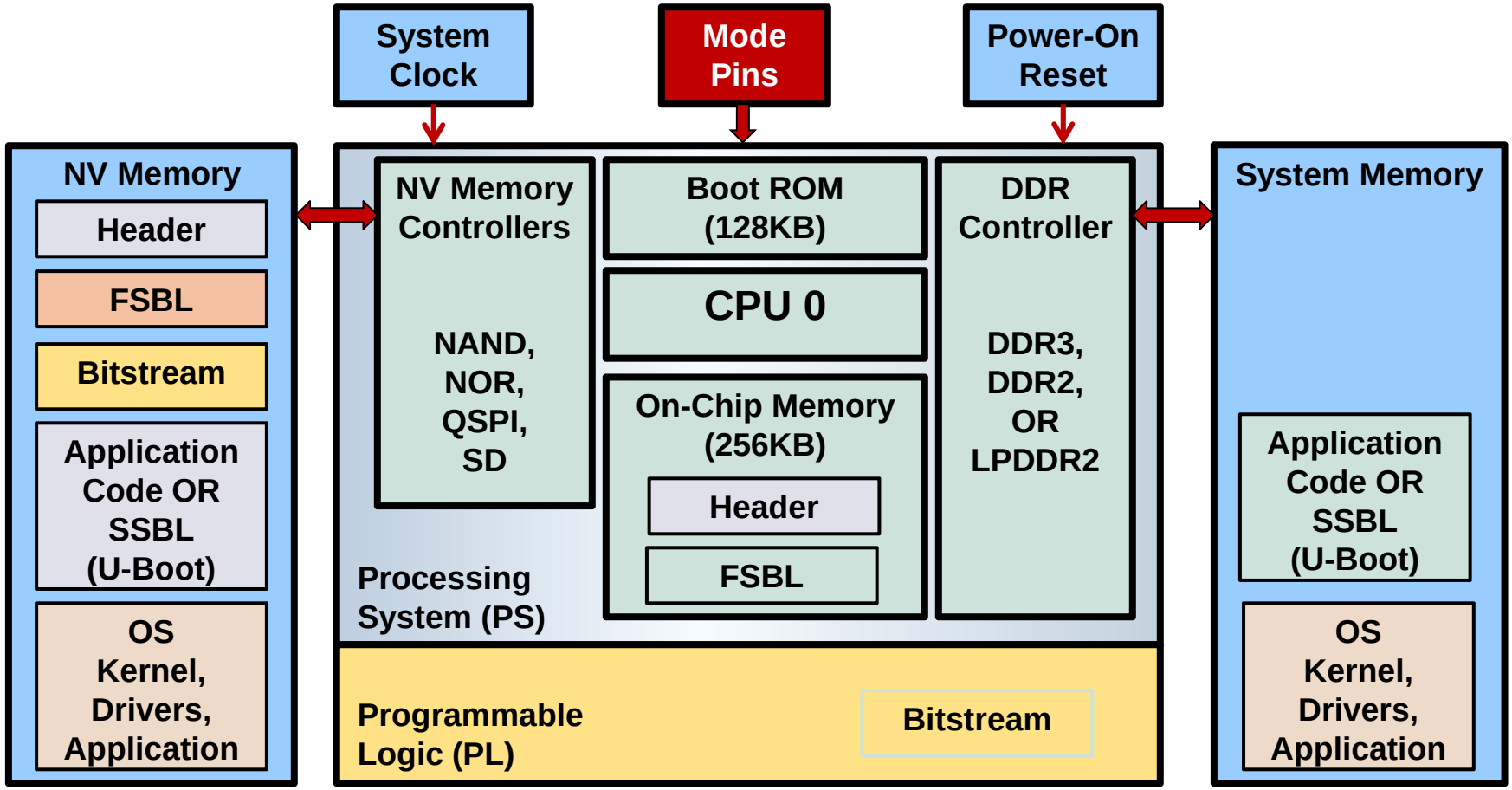
Typical Master Non-Secure Boot and Configuration Flow



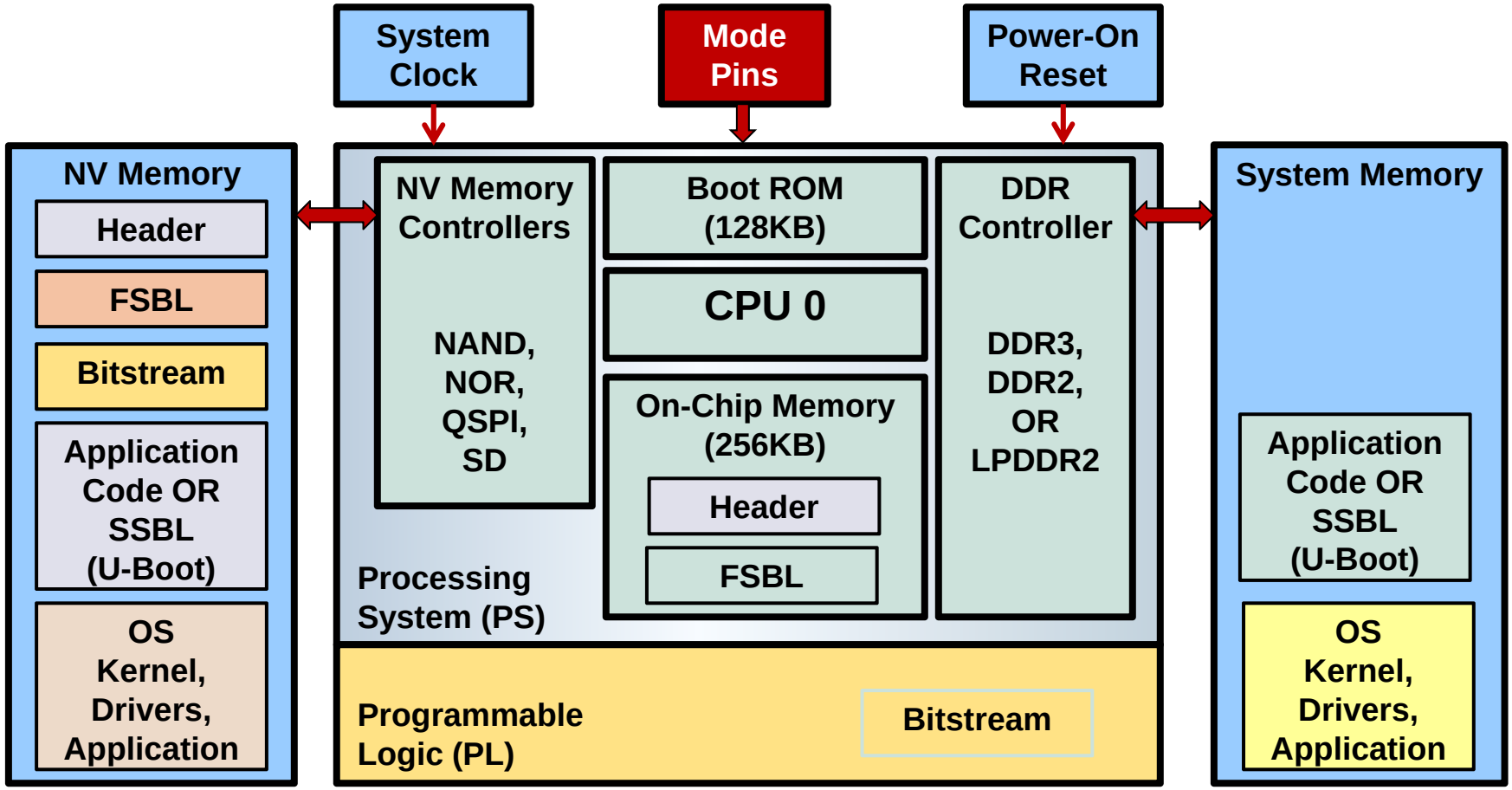
Typical Master Non-Secure Boot and Configuration Flow



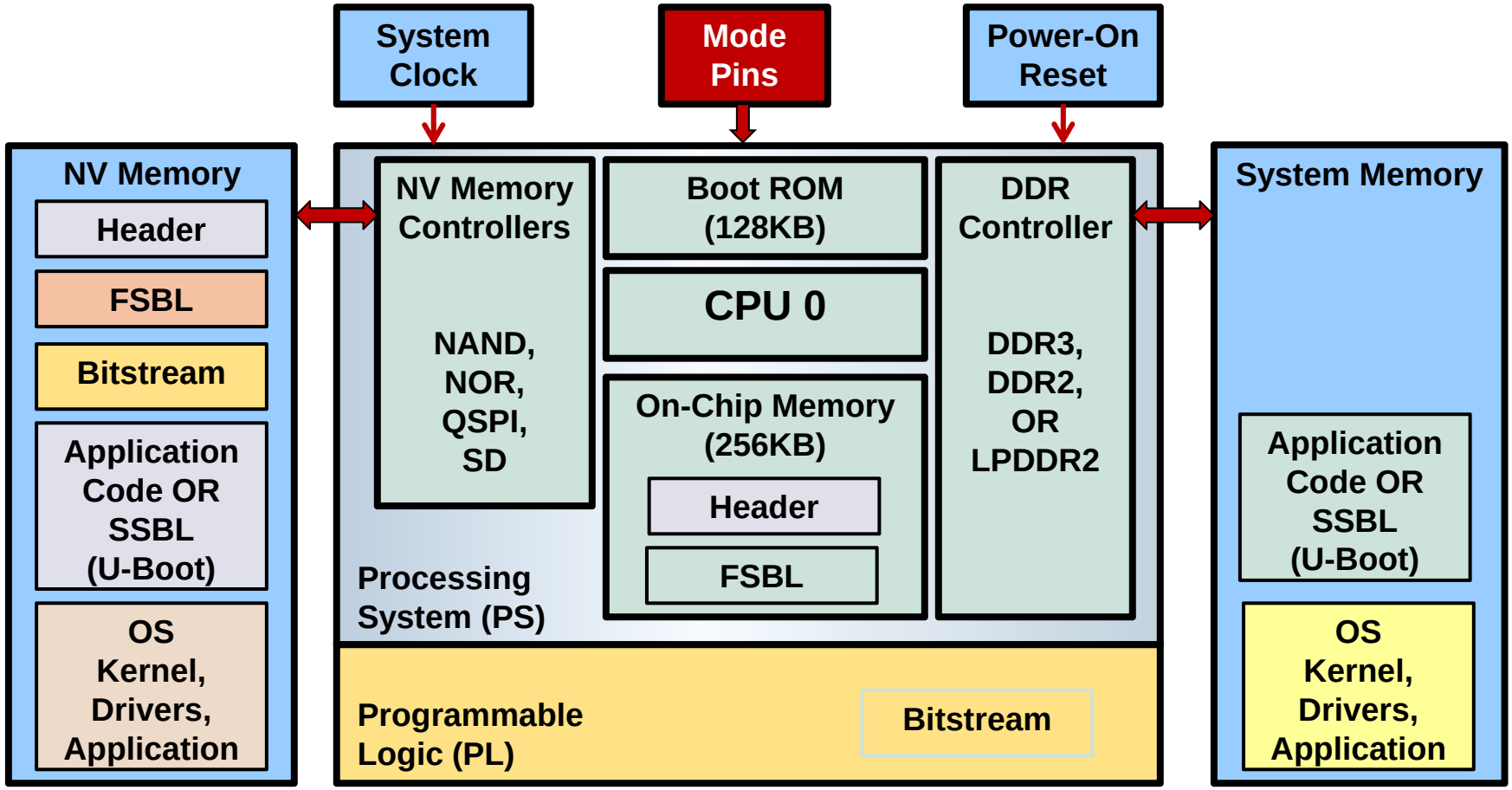
Typical Master Non-Secure Boot and Configuration Flow



Typical Master Non-Secure Boot and Configuration Flow



Typical Master Non-Secure Boot and Configuration Flow

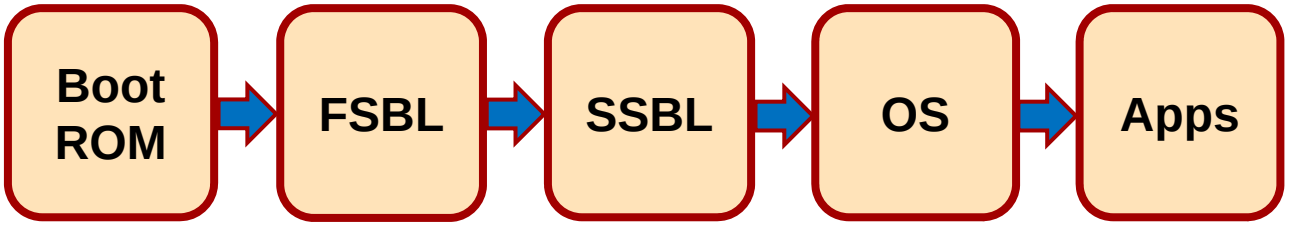


Compile FSBL with *FSBL_DEBUG_INFO*, *DEBUG*, and *FSBL_DEBUG_GENERAL* debug symbols to monitor Boot and Configuration progress on UART1

Secure Boot and Configuration

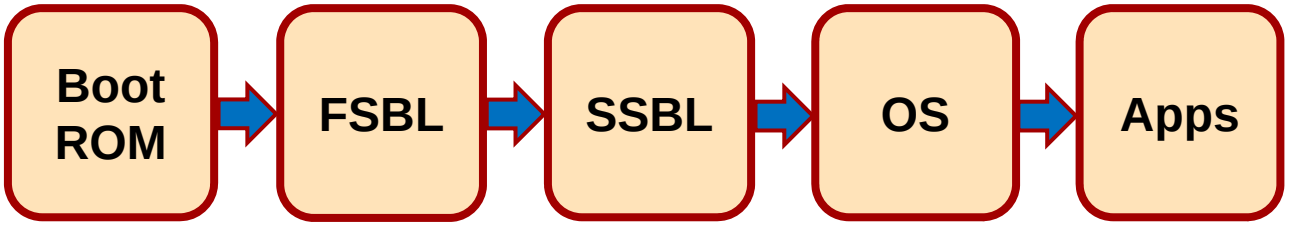
Zynq Secure Boot Overview

- **Zynq supports the ability to perform a secure boot to load authenticated and encrypted PS images and PL bitstreams**
  - Secure booting typically requires multiple phases
  - Each phase must hand off security responsibility to the next successive phase without compromising security
  - *Boot ROM* sets the root of trust by securing all access points and then loading the *FSBL*
  - *FSBL* and *SSBL* are required to maintain the chain of trust both in operation and in handoffs



Zynq Secure Boot Overview

- **Zynq supports the ability to perform a secure boot to load authenticated and encrypted PS images and PL bitstreams**
  - Secure booting typically requires multiple phases
  - Each phase must hand off security responsibility to the next successive phase without compromising security
  - *Boot ROM* sets the root of trust by securing all access points and then loading the *FSBL*
  - *FSBL* and *SSBL* are required to maintain the chain of trust both in operation and in handoffs



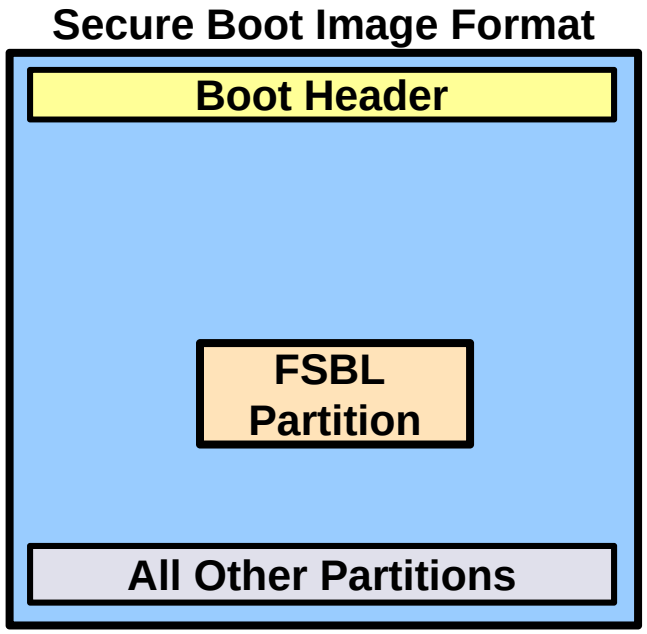
Boot ROM enables Secure Boot only if the FSBL partition is encrypted

Cryptographic Keys Used by Zynq

- **Zynq uses the following cryptographic keys**
  - AES 256-bit key
  - HMAC 256-bit key (SHA-256)
  - RSA Primary/Secondary Secret Keys (PSK, SSK)
  - RSA Primary/Secondary Public Keys (PPK, SPK)

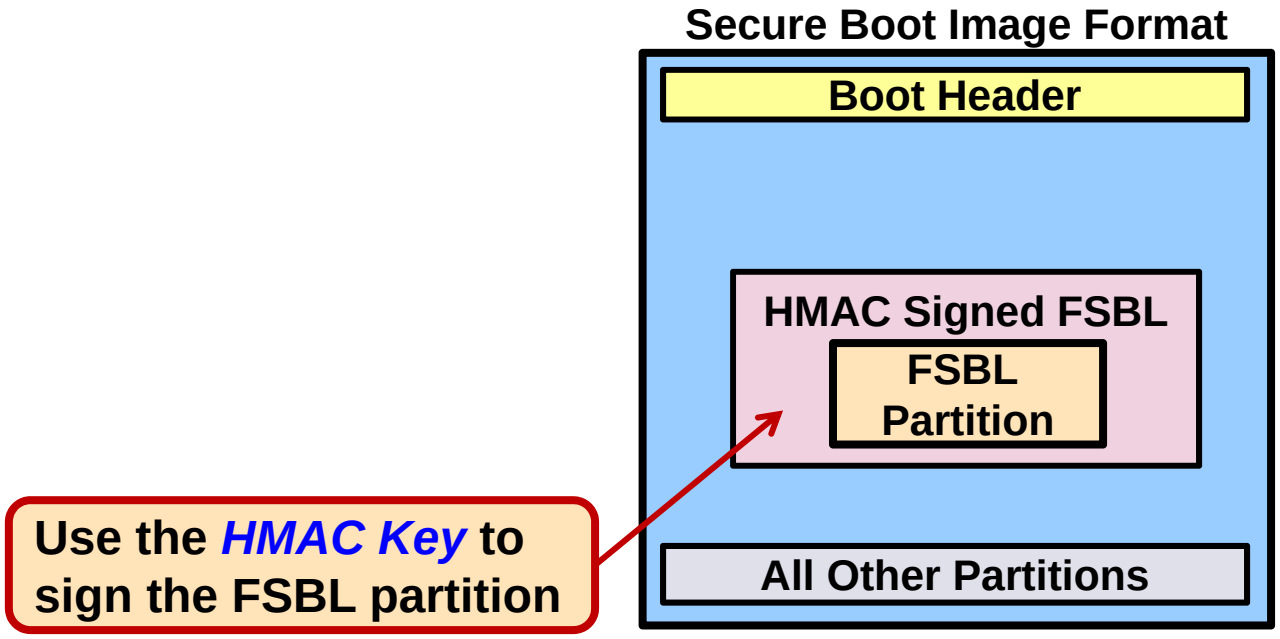
Cryptographic Keys Used by Zynq

- **Zynq uses the following cryptographic keys**
  - AES 256-bit key
  - HMAC 256-bit key (SHA-256)
  - RSA Primary/Secondary Secret Keys (PSK, SSK)
  - RSA Primary/Secondary Public Keys (PPK, SPK)



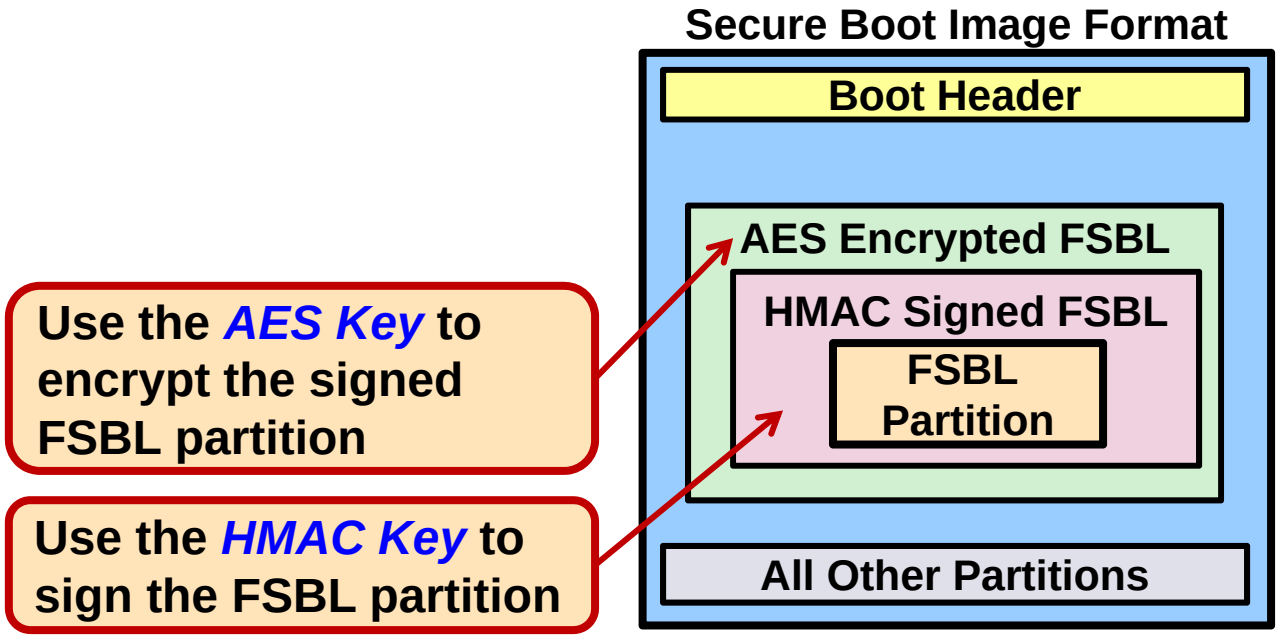
Cryptographic Keys Used by Zynq

- **Zynq uses the following cryptographic keys**
  - AES 256-bit key
  - HMAC 256-bit key (SHA-256)
  - RSA Primary/Secondary Secret Keys (PSK, SSK)
  - RSA Primary/Secondary Public Keys (PPK, SPK)



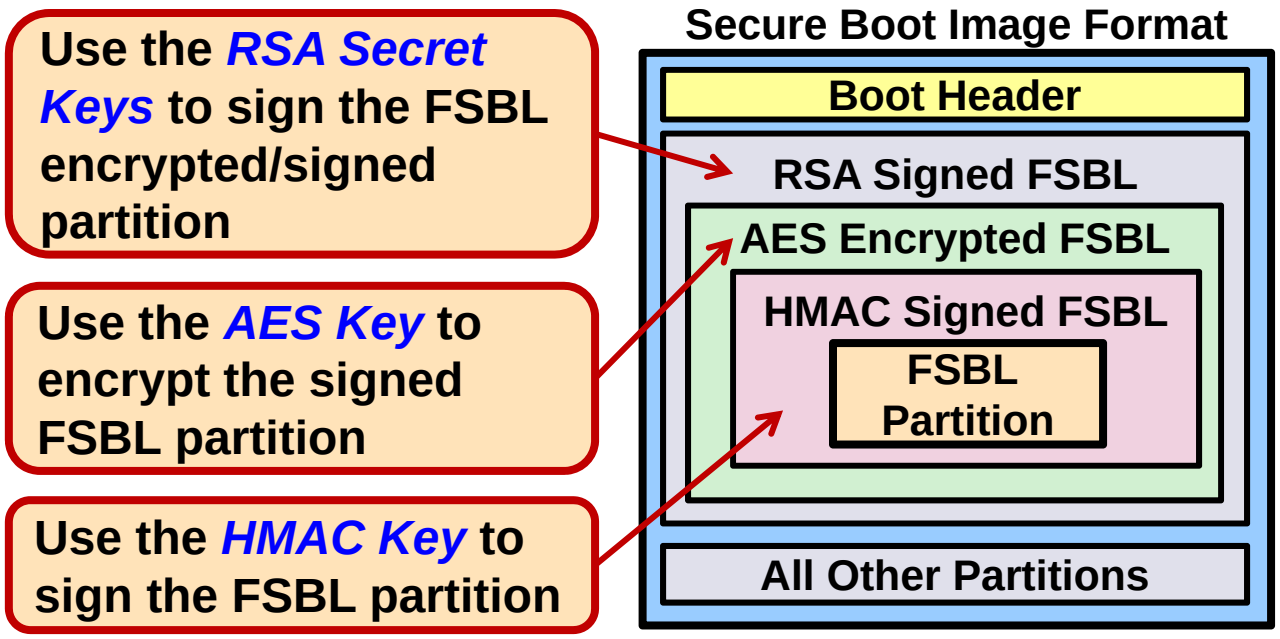
Cryptographic Keys Used by Zynq

- **Zynq uses the following cryptographic keys**
  - AES 256-bit key
  - HMAC 256-bit key (SHA-256)
  - RSA Primary/Secondary Secret Keys (PSK, SSK)
  - RSA Primary/Secondary Public Keys (PPK, SPK)



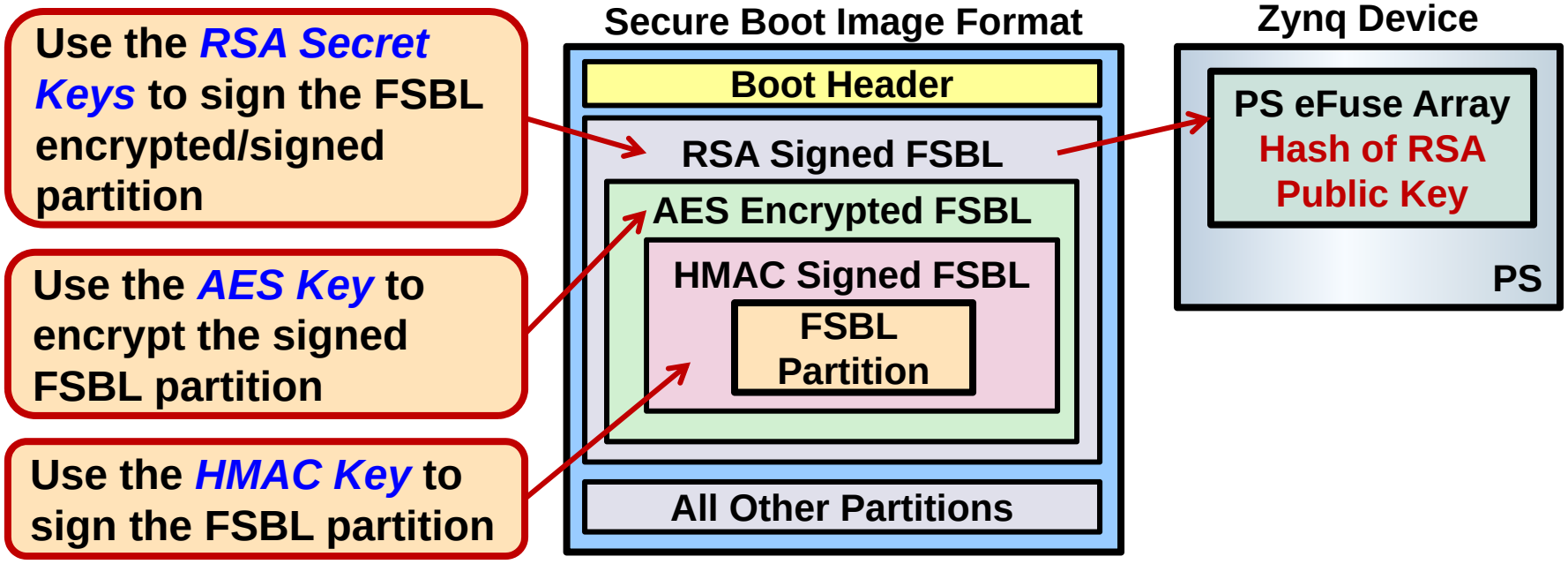
Cryptographic Keys Used by Zynq

- **Zynq uses the following cryptographic keys**
  - AES 256-bit key
  - HMAC 256-bit key (SHA-256)
  - RSA Primary/Secondary Secret Keys (PSK, SSK)
  - RSA Primary/Secondary Public Keys (PPK, SPK)



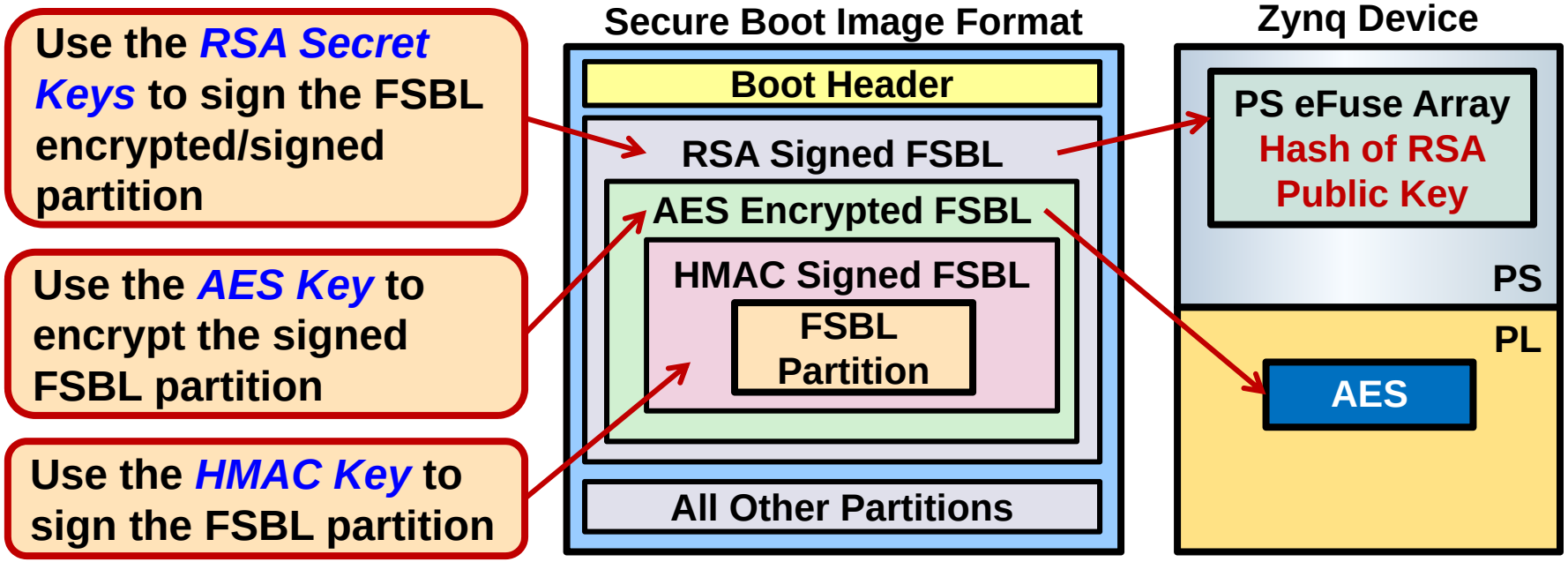
Cryptographic Keys Used by Zynq

- **Zynq uses the following cryptographic keys**
  - AES 256-bit key
  - HMAC 256-bit key (SHA-256)
  - RSA Primary/Secondary Secret Keys (PSK, SSK)
  - RSA Primary/Secondary Public Keys (PPK, SPK)



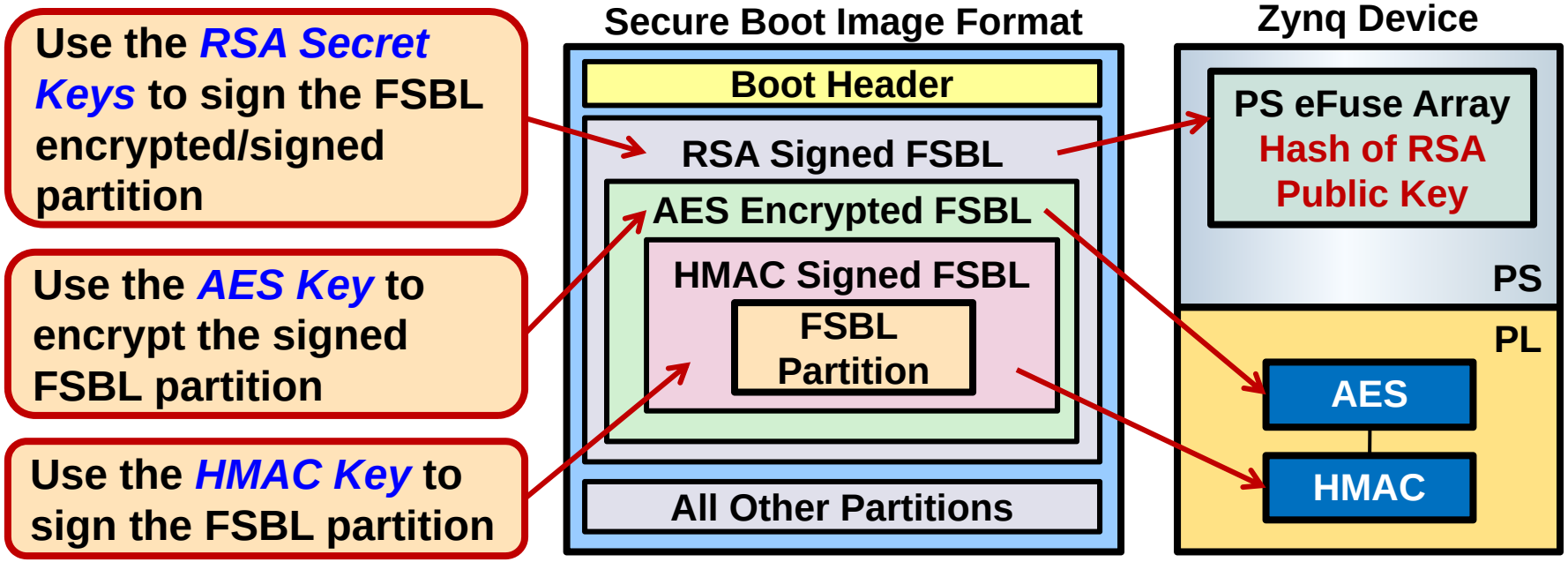
Cryptographic Keys Used by Zynq

- **Zynq uses the following cryptographic keys**
  - AES 256-bit key
  - HMAC 256-bit key (SHA-256)
  - RSA Primary/Secondary Secret Keys (PSK, SSK)
  - RSA Primary/Secondary Public Keys (PPK, SPK)



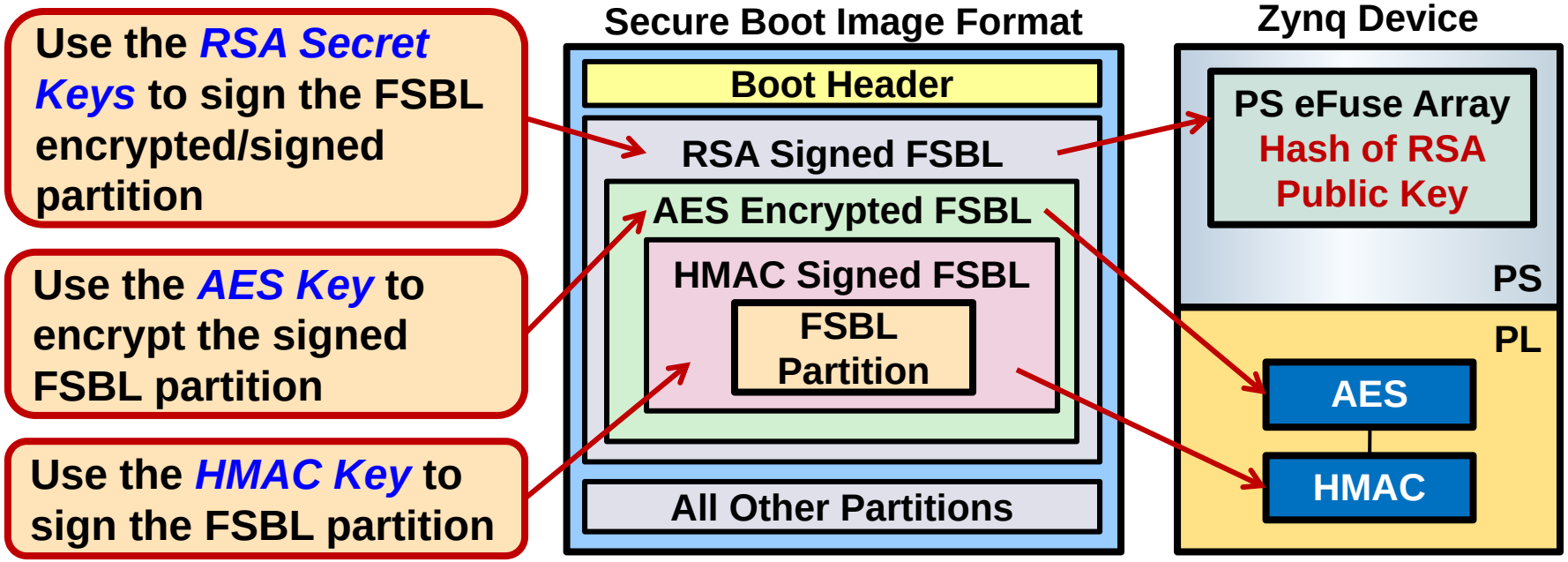
Cryptographic Keys Used by Zynq

- **Zynq uses the following cryptographic keys**
  - AES 256-bit key
  - HMAC 256-bit key (SHA-256)
  - RSA Primary/Secondary Secret Keys (PSK, SSK)
  - RSA Primary/Secondary Public Keys (PPK, SPK)



Cryptographic Keys Used by Zynq

- **Zynq uses the following cryptographic keys**
  - AES 256-bit key
  - HMAC 256-bit key (SHA-256)
  - RSA Primary/Secondary Secret Keys (PSK, SSK)
  - RSA Primary/Secondary Public Keys (PPK, SPK)

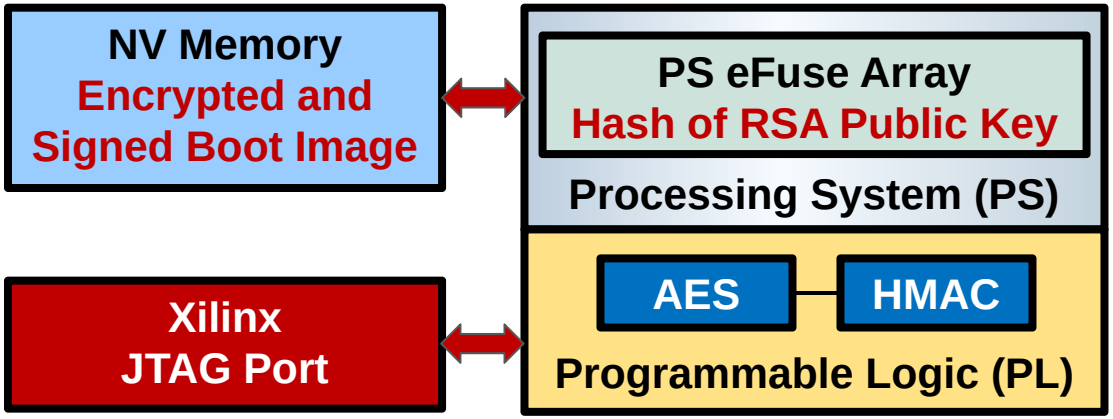


The *FSBL RSA Authentication* is enabled via PS eFuse array while RSA authentication of all other partitions is enabled via *Partition Headers*

Secure Boot Image Generation and Programming

- **Generate the keys and encrypted/signed boot image partitions**
  - Generate the *AES/HMAC Key*
  - Generate the *RSA Authentication Secret and Public Keys*
  - Generate the *Hash of RSA Public Key*
  - Use the above keys to encrypt and sign the boot image partitions

- **Program the keys and encrypted/signed boot image**
  - Program the *AES/HMAC Key* into the PL eFuse array or BBRAM
  - Program the *Hash of RSA Public Key* into the PS eFuse array
  - Program the encrypted/signed boot image into the NV memory



AES/HMAC Key Generation

- **Xilinx Bootgen tool can be used to generate the AES/HMAC key**
  - Create a *Boot Image Format* file (for example, *generate_aeskey.bif* file with the following contents)

```
generate_aeskey_image:
      {
        [aeskeyfile] bbram.nky
        [bootloader, encryption=aes] fsbl.elf
      }
```

  - Use the following Bootgen command to generate the AES/HMAC key

```
bootgen -image generate_aeskey.bif -o temp.mcs -encrypt bbram
```

    - The *–encrypt* option can be specified with *bbram* or *efuse*

  - Bootgen will generate the *bbram.nky* file containing the AES/HMAC key
    - Use iMPACT/Vivado to program the *AES/HMAC Key* into the Zynq PL

RSA Key Generation

- **The OpenSSL tool can be used to generate the RSA keys**
  - The OpenSSL tool is in Linux distributions. Windows users can use Cygwin OpenSSL or download it from www.openssl.org
  - The primary and secondary secret RSA keys are generated using the following OpenSSL command

```
openssl genrsa -out psk.pk1 2048
openssl genrsa -out ssk.pk1 2048
```

  - In RSA, the public key is contained in the secret key. The following OpenSSL command is used to extract the public key from the secret key

```
openssl rsa -pubout -in psk.pk1 -out ppk.pub
openssl rsa -pubout -in ssk.pk1 -out spk.pub
```

Generating the Hash of RSA Primary Public Key

- **After generating the RSA keys using OpenSSL, *Bootgen* is used to generate the *Hash of RSA Primary Public key***
  - Create a *gen_hash_ppk.bif* file with the following content

```
gen_hash_ppk:
      {
        [pskfile] psk.pk1
        [sskfile] ssk.pk1
        [bootloader, authentication=rsa] fsbl.elf
      }
```

  - Use the following bootgen command to generate the *hash_ppk.txt* file

```
bootgen -image gen_hash_ppk.bif -efuseppkbits hash_ppk.txt
```

    - The *hash_ppk.txt* file contains the *Hash of RSA Primary Public Key*
    - Bootgen uses SHA-256 hash algorithm to generate a 256-bit long *Hash of RSA Primary Public Key*

Secure Boot Image Generation

- **Create a *Boot Image Format* file describing the image partitions**
  - For example, *bootimage.bif* file with the following contents

```
image: {
  [aeskeyfile] bbram.nky
  [pskfile] psk.pk1
  [sskfile] ssk.pk1
  [bootloader,encryption=aes,authentication=rsa] fsbl.elf
  [encryption=aes, authentication=rsa] system.bit
  [authentication=rsa] u-boot.elf
  [authentication=rsa,load=0x3000000,offset=0x500000] uImage.bin
  [authentication=rsa,load=0x2A00000,offset=0xA00000] devicetree.dtb
  [authentication=rsa,load=0x2000000,offset=0xA20000] uramdisk.image.gz
  [authentication=rsa, encryption=aes] application.elf }
```

  - Use the following Bootgen command to generate the boot image

```
bootgen -image bootimage.bif -o <design>.mcs -encrypt bbram
```

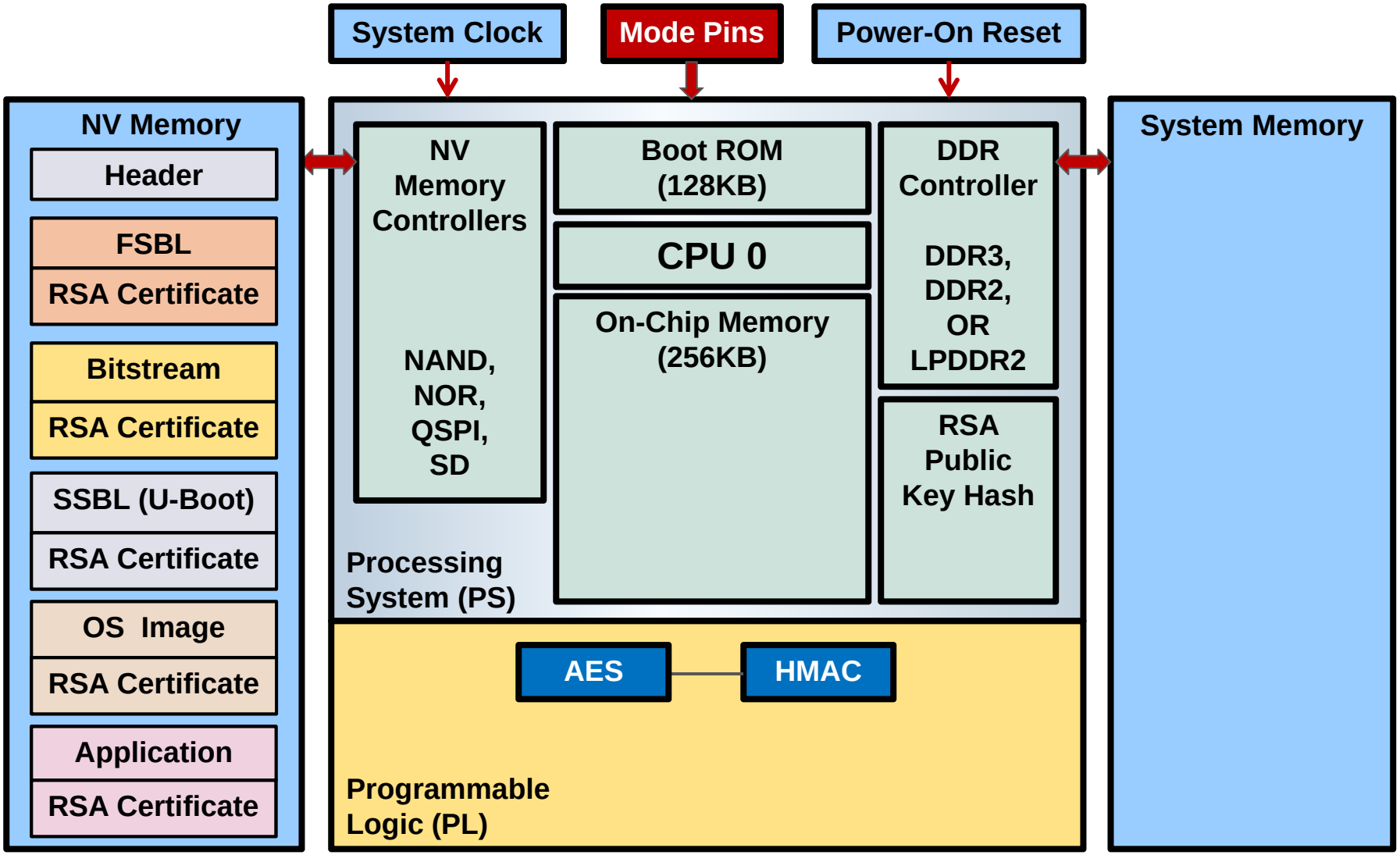
Programming the Hash of RSA Public Key

- **SDK *Secure Key Driver* software project is used to program the *RSA Enable* control bit and the *Hash of RSA Public Key* into the PS eFuse array**
  - Source files for this software project are located in the **UG1025** zip file (*xilskey_efuse_example.c* and *xilskey_input.h* files)
  - Edit the *xilskey_input.h* file as follows

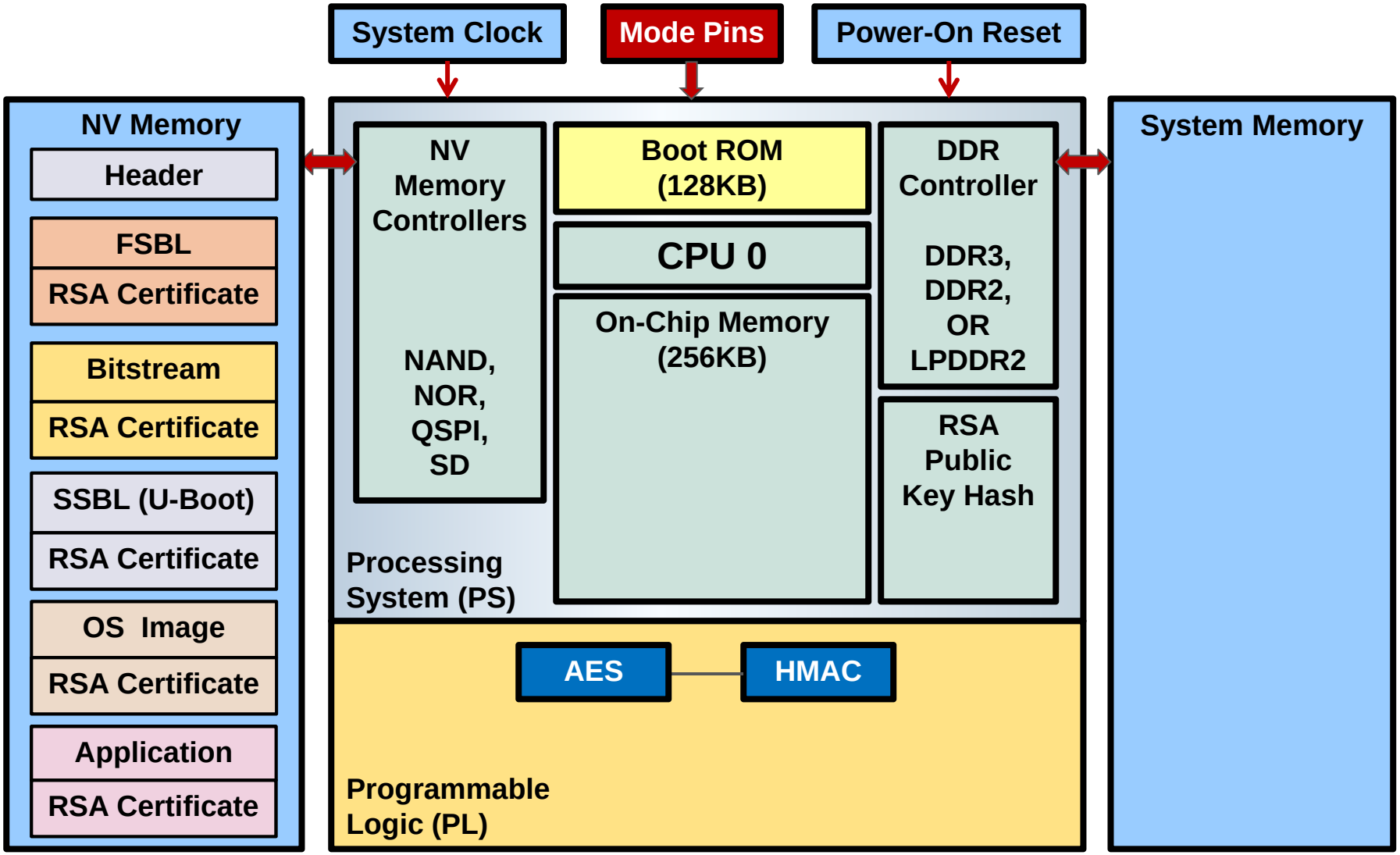
    1) **Define *XSK_EFUSEPS_DRIVER***
    2) **Define *XSK_EFUSEPS_RSA_KEY_HASH_VALUE* as Has of PPK from *hash_ppk.txt* file**
    3) **Set *XSK_EFUSEPS_ENABLE_RSA_KEY_HASH TRUE***
    4) **Set *XSK_EFUSEPS_ENABLE_RSA_AUTH* TRUE**

  - Build the *Secure Key Driver* software project in SDK and use *XMD* to download the code to the PS OCM and run it (this will program the PS eFuse array)

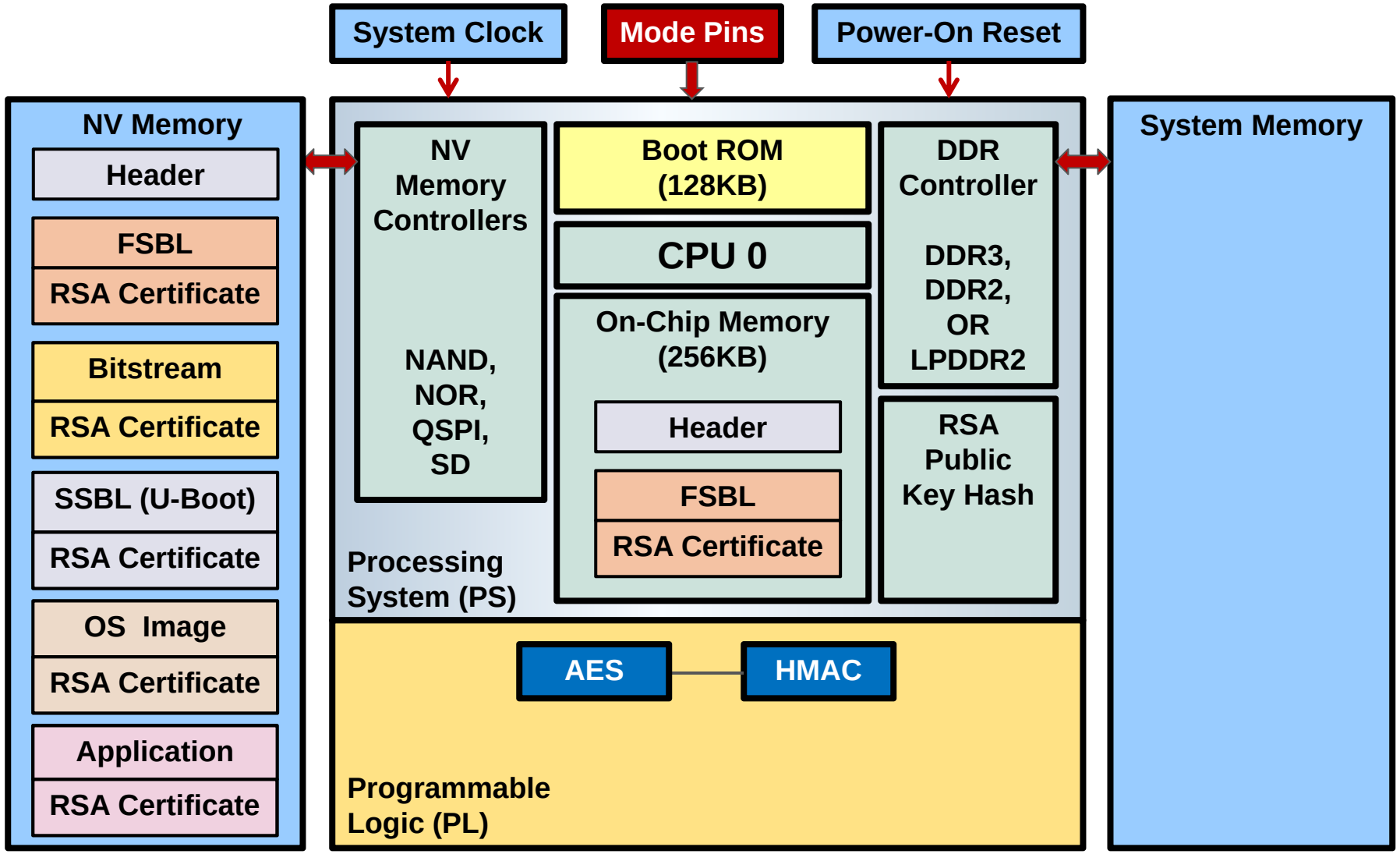
Typical Secure Boot and Configuration Flow



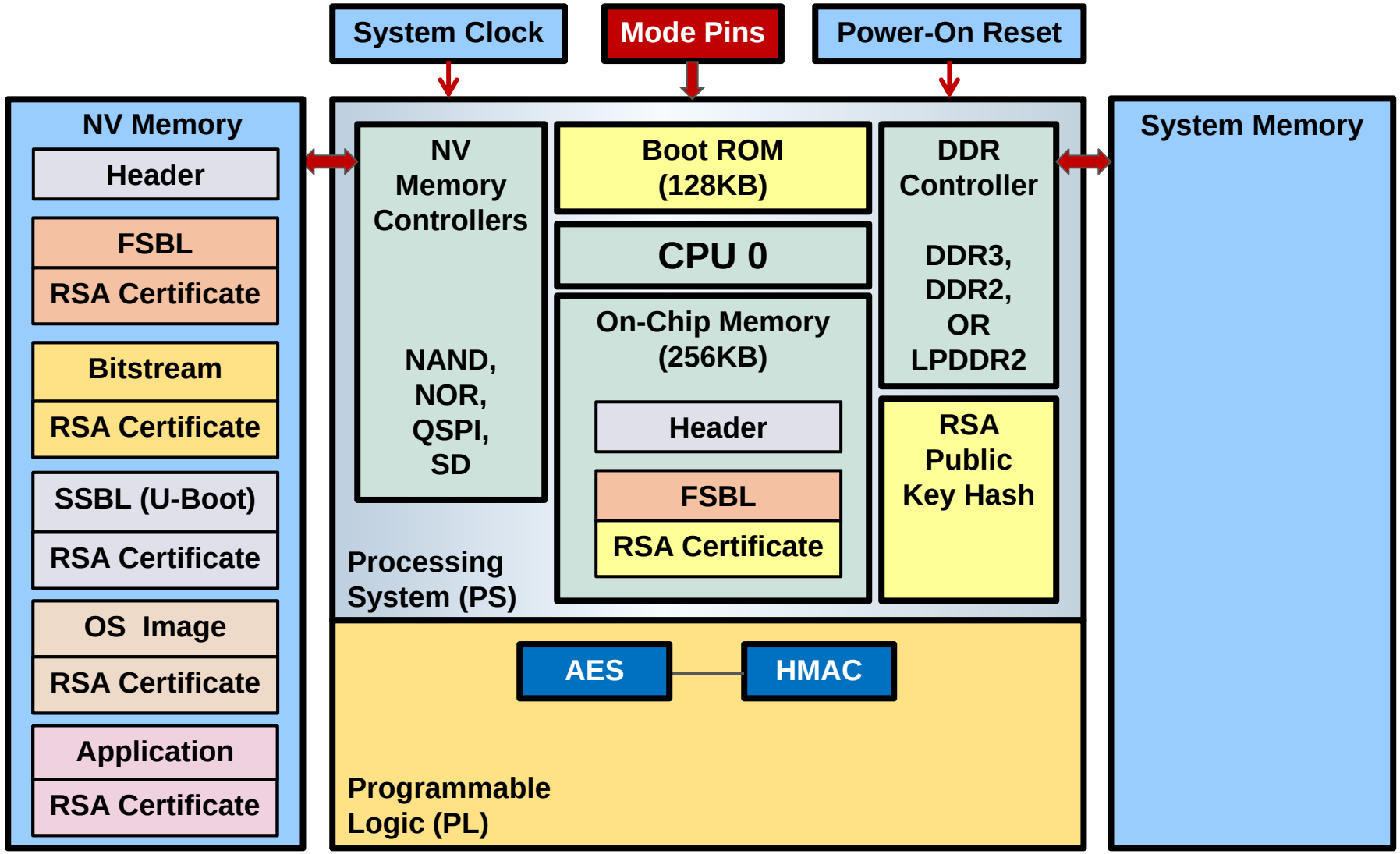
Typical Secure Boot and Configuration Flow



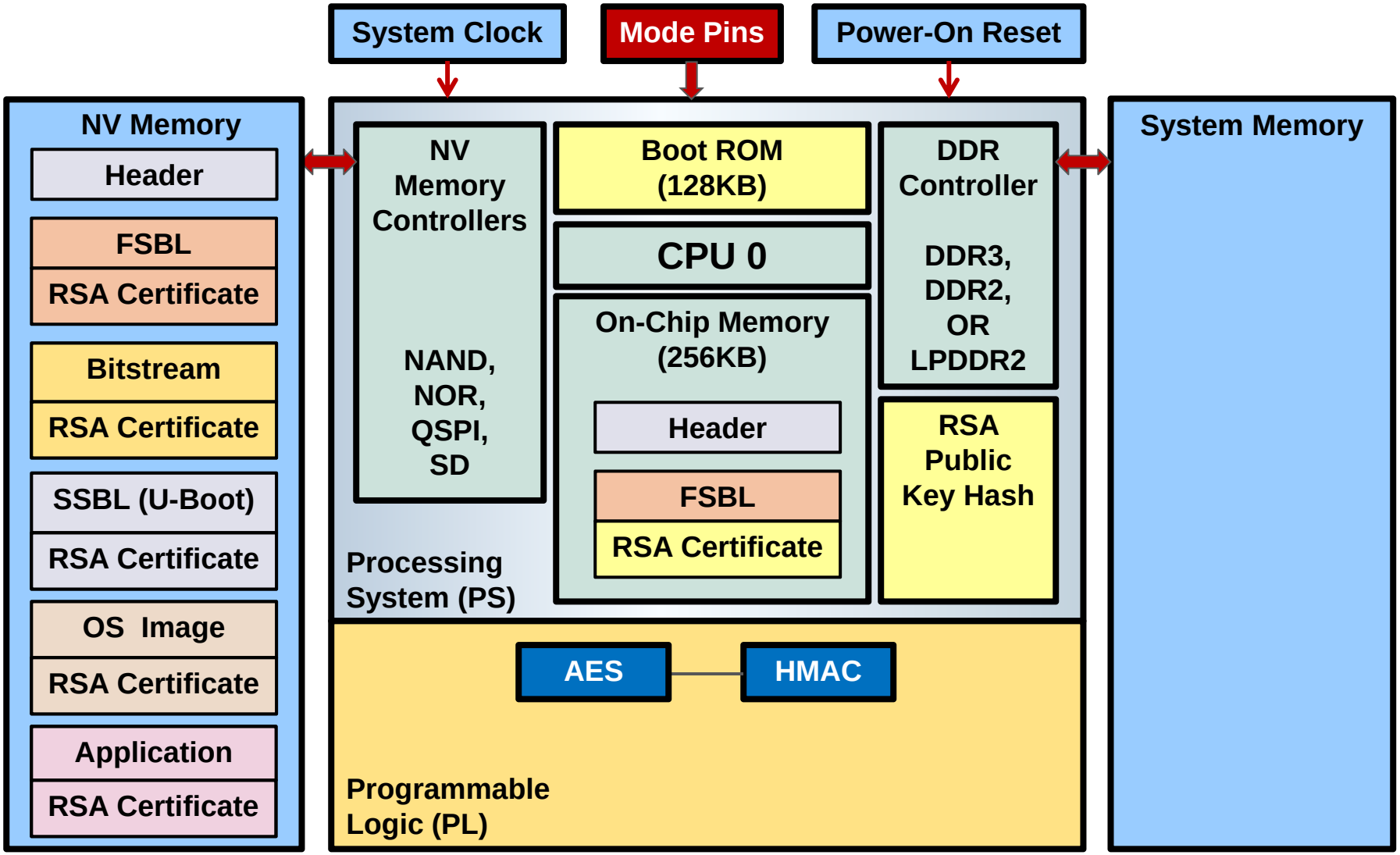
Typical Secure Boot and Configuration Flow



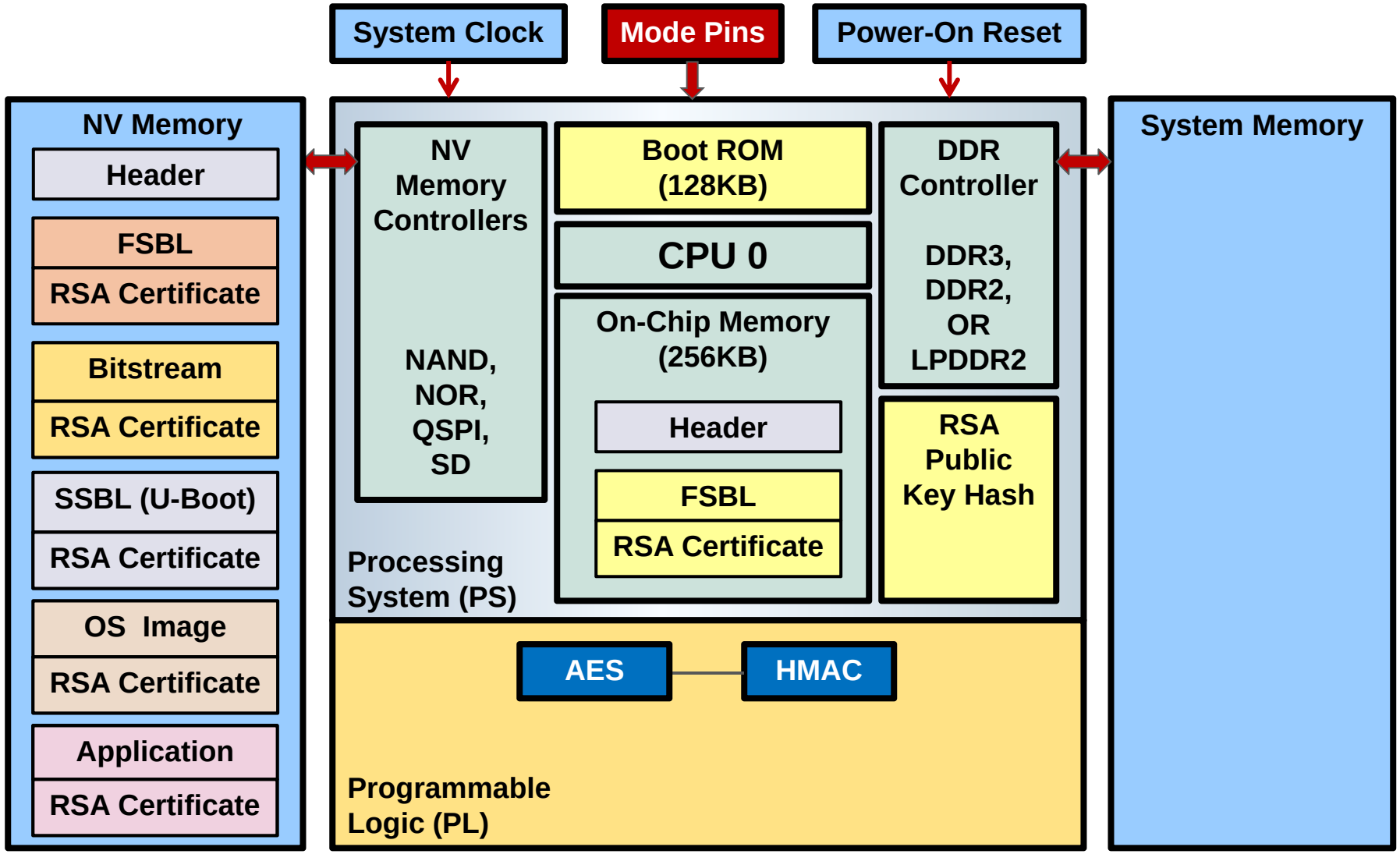
Typical Secure Boot and Configuration Flow



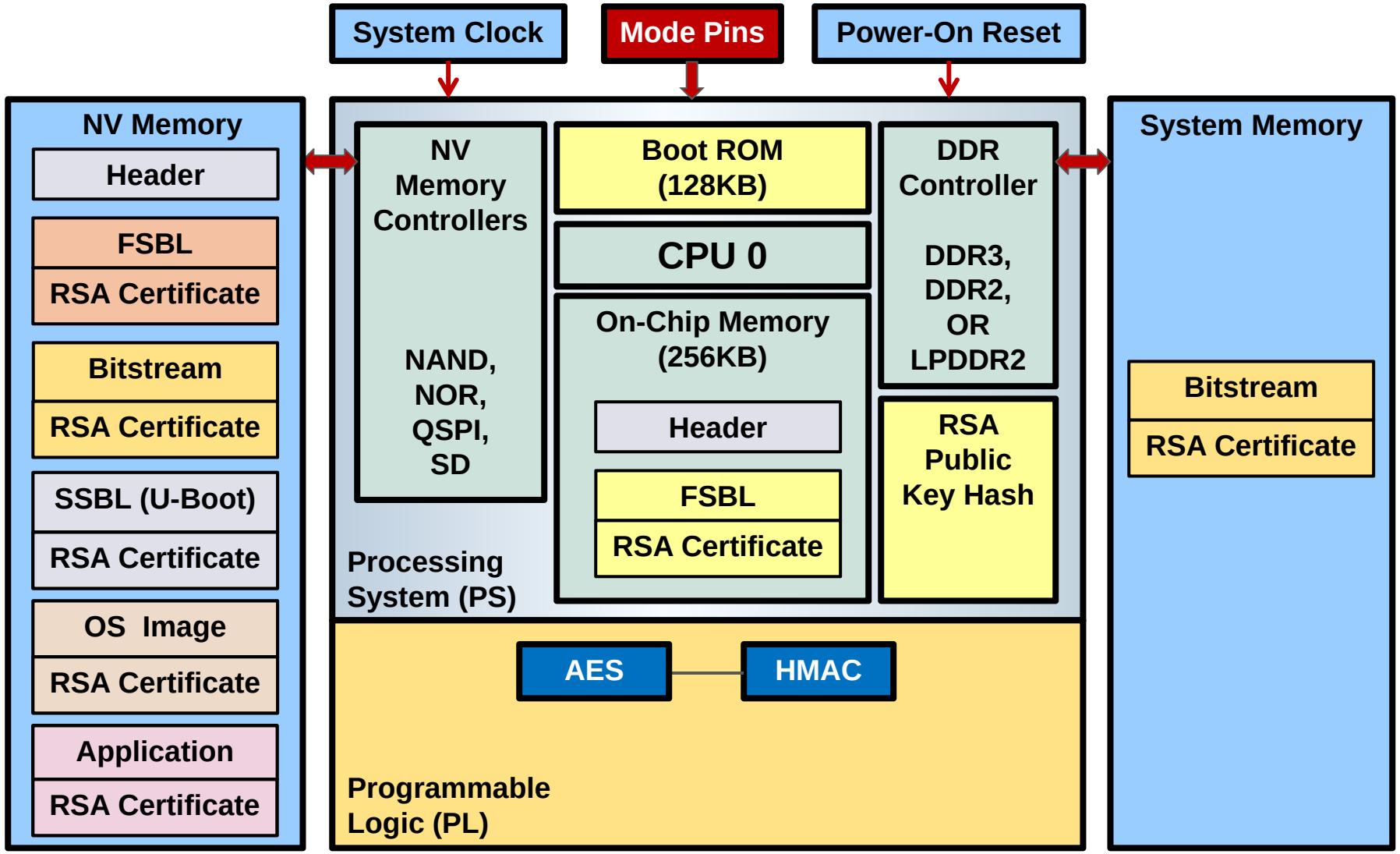
Typical Secure Boot and Configuration Flow



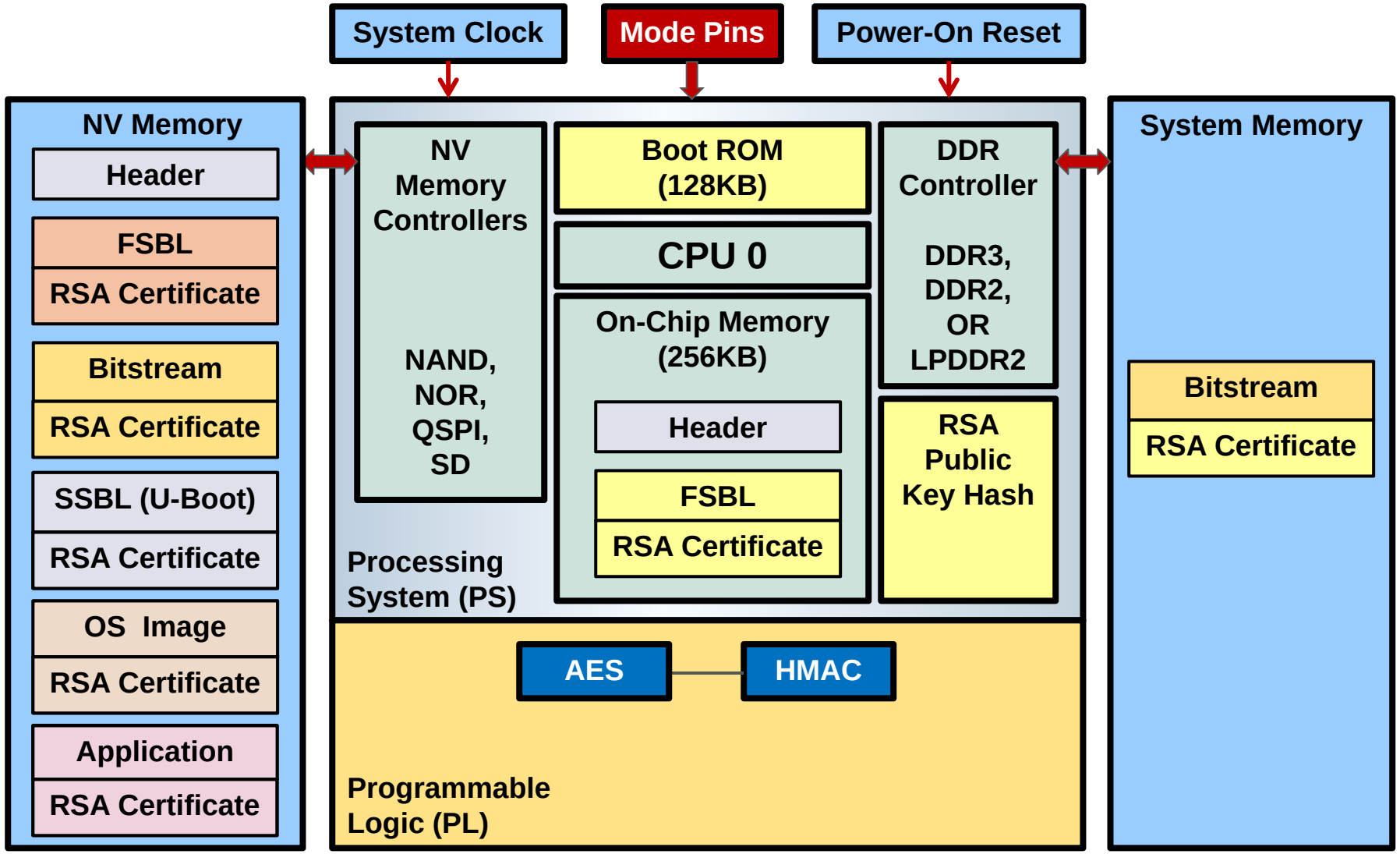
Typical Secure Boot and Configuration Flow



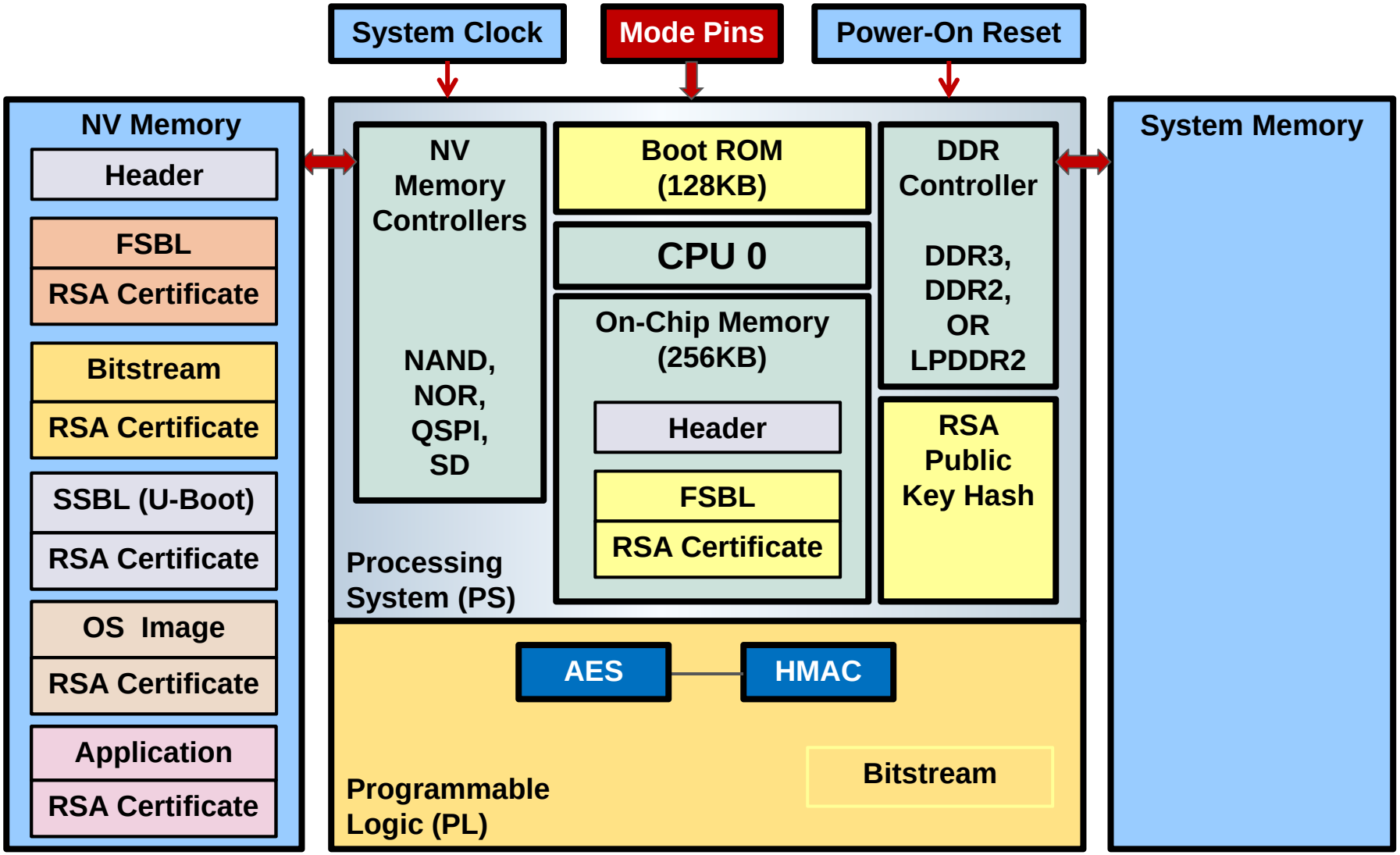
Typical Secure Boot and Configuration Flow



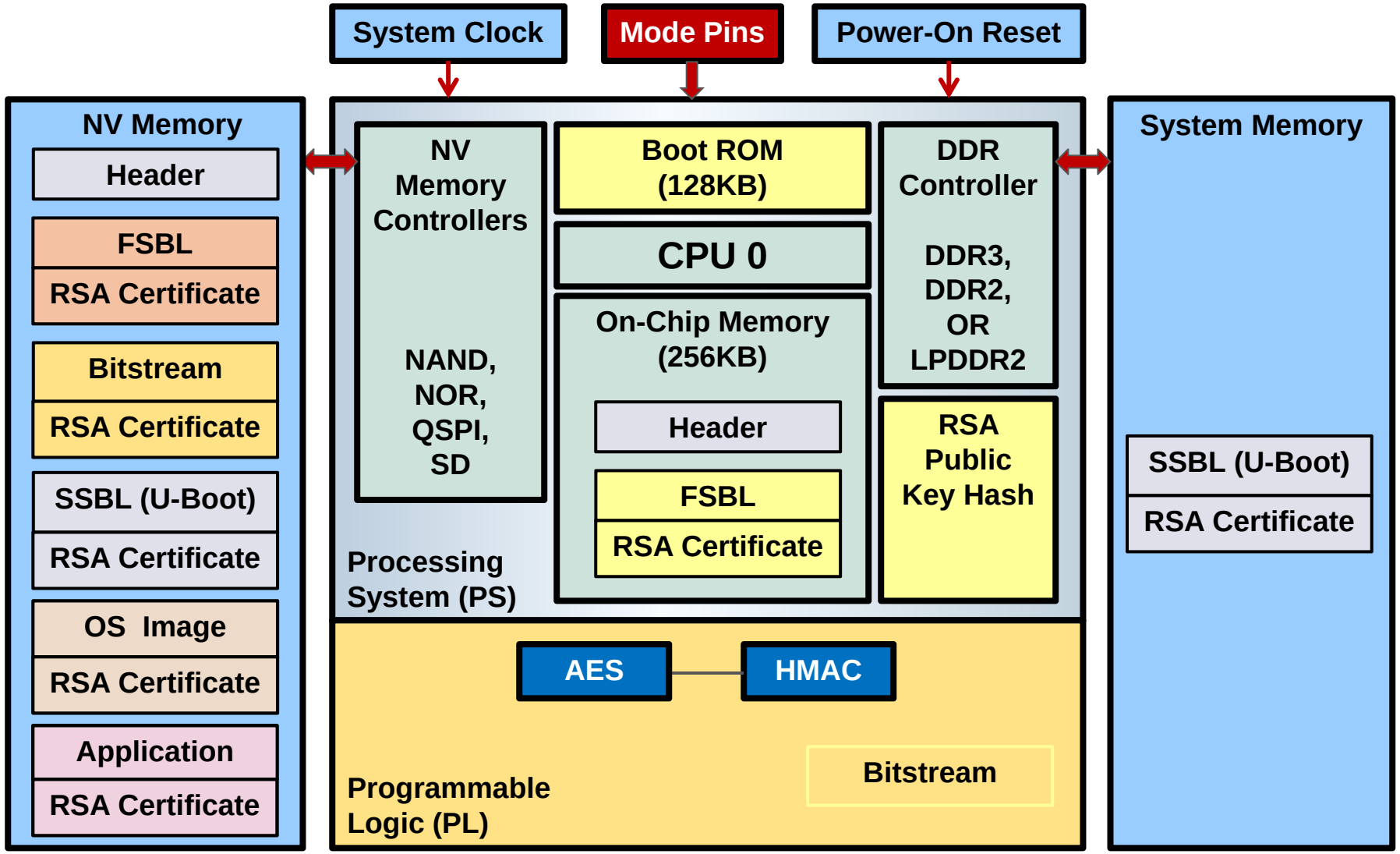
Typical Secure Boot and Configuration Flow



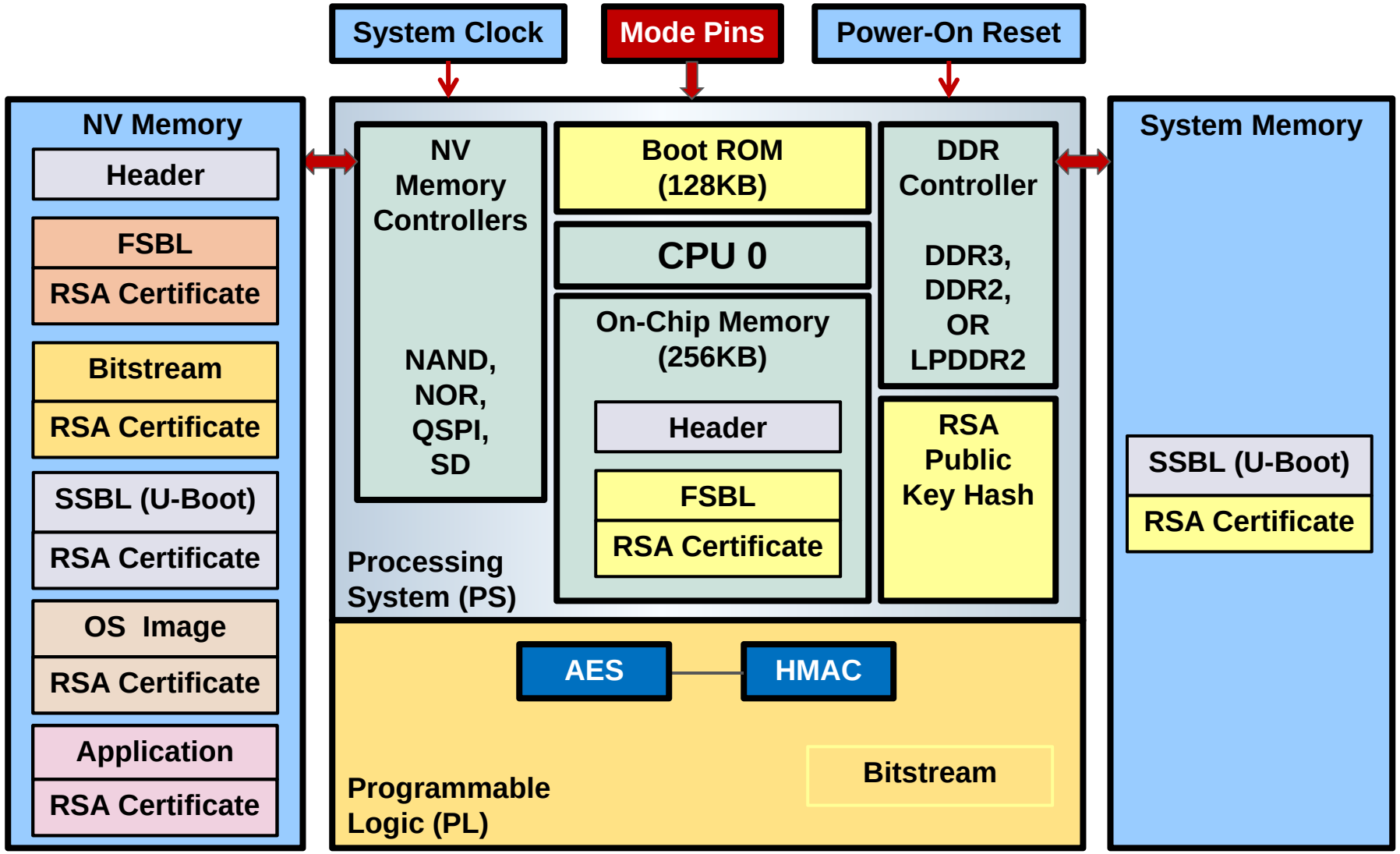
Typical Secure Boot and Configuration Flow



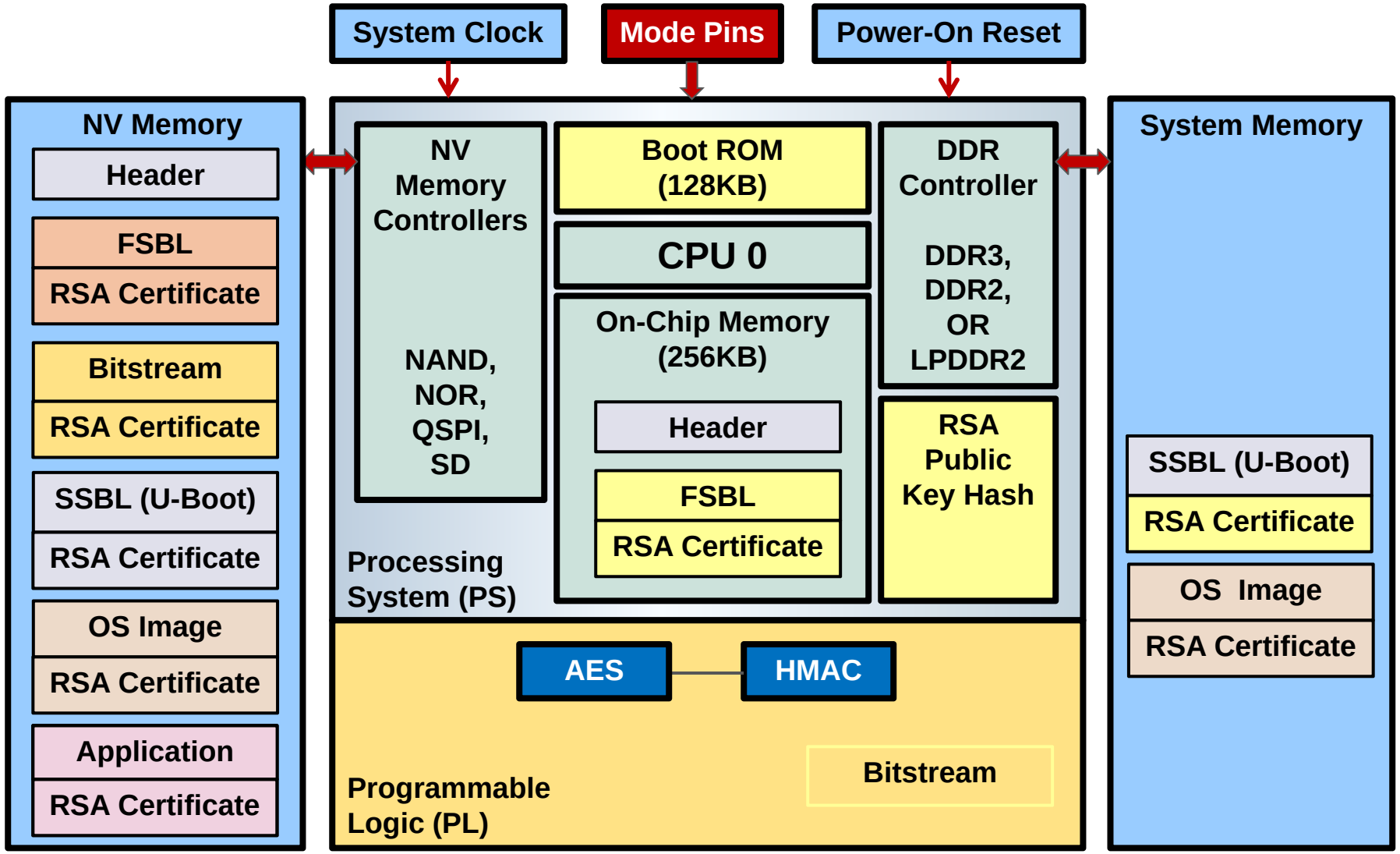
Typical Secure Boot and Configuration Flow



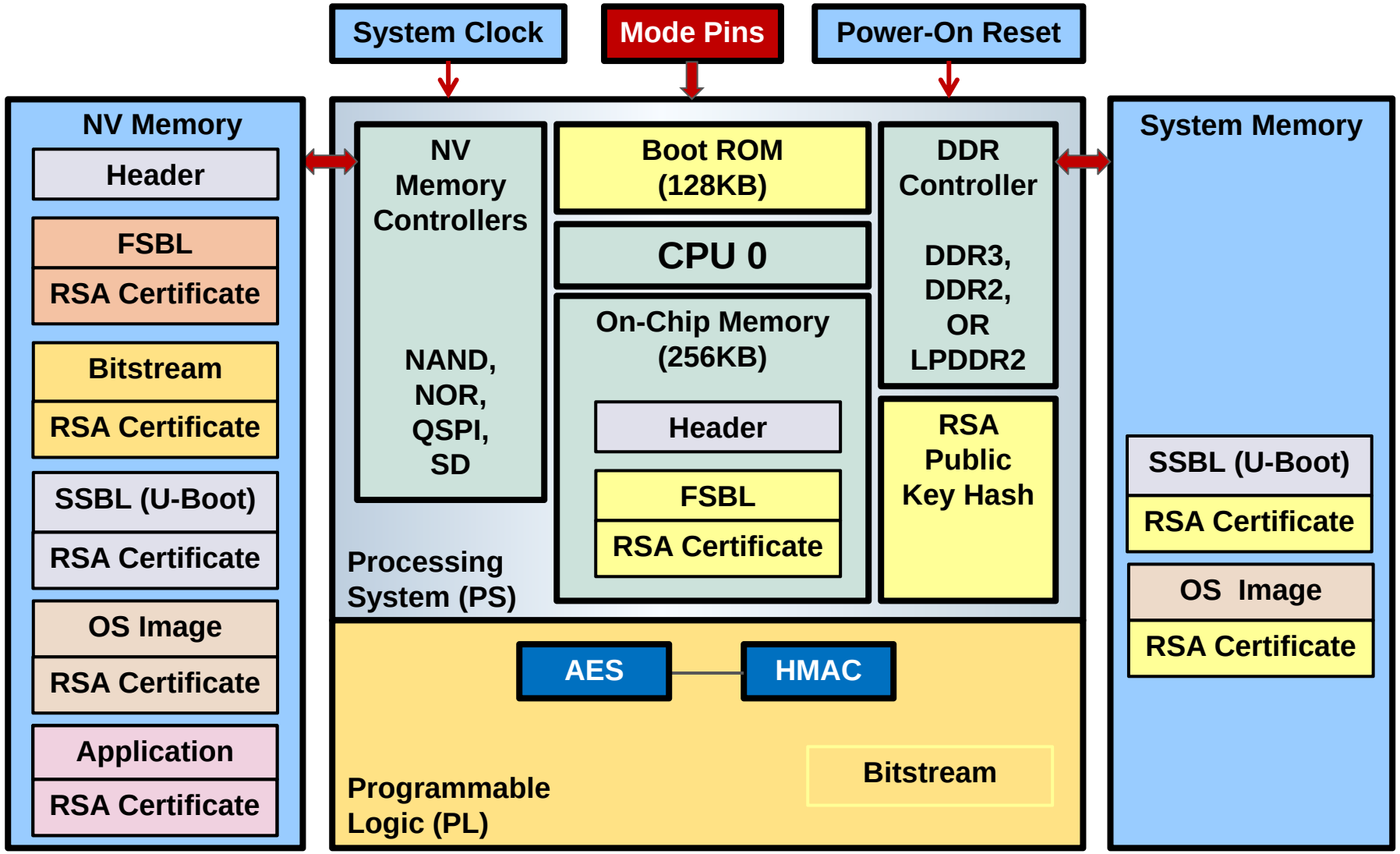
Typical Secure Boot and Configuration Flow



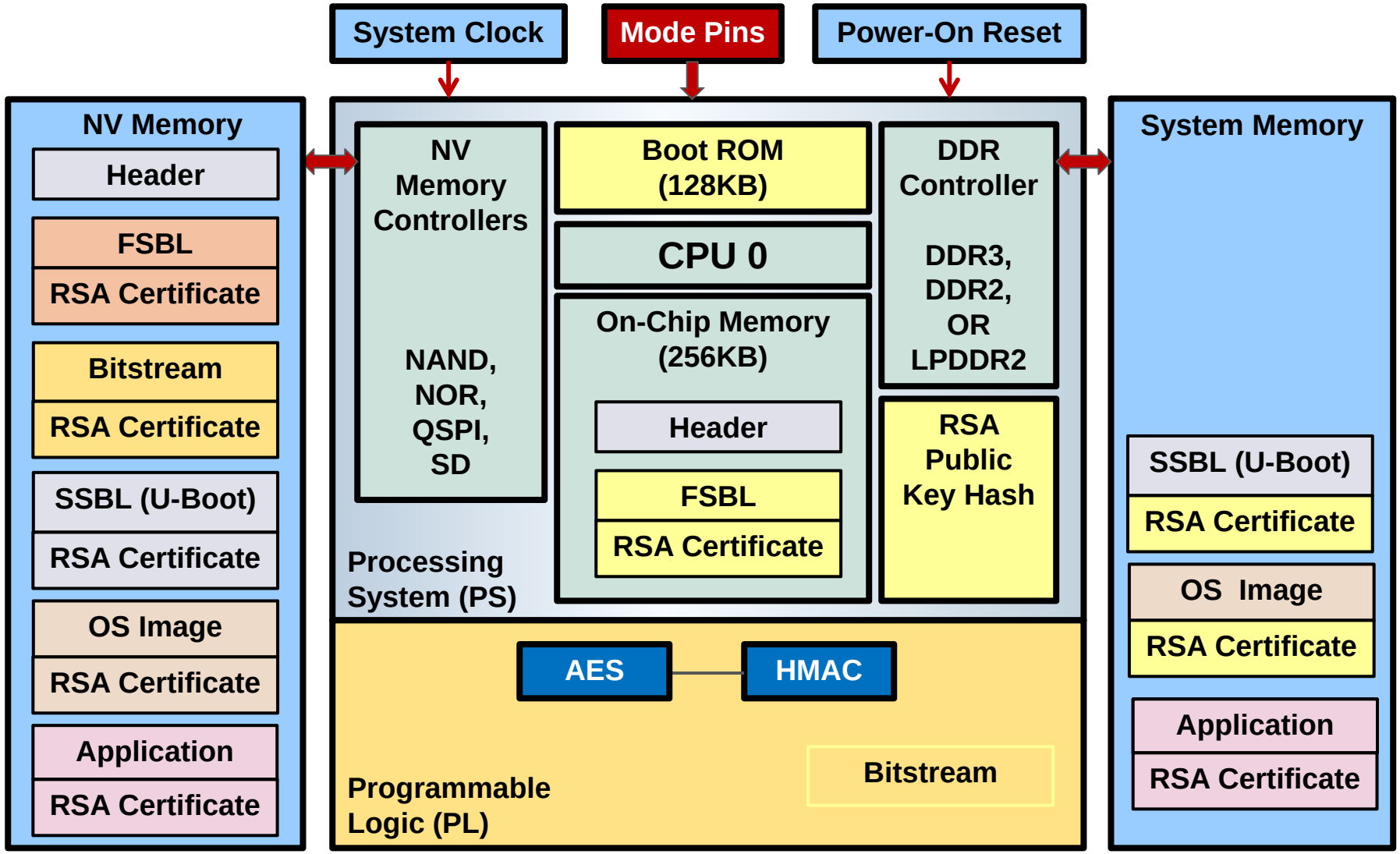
Typical Secure Boot and Configuration Flow



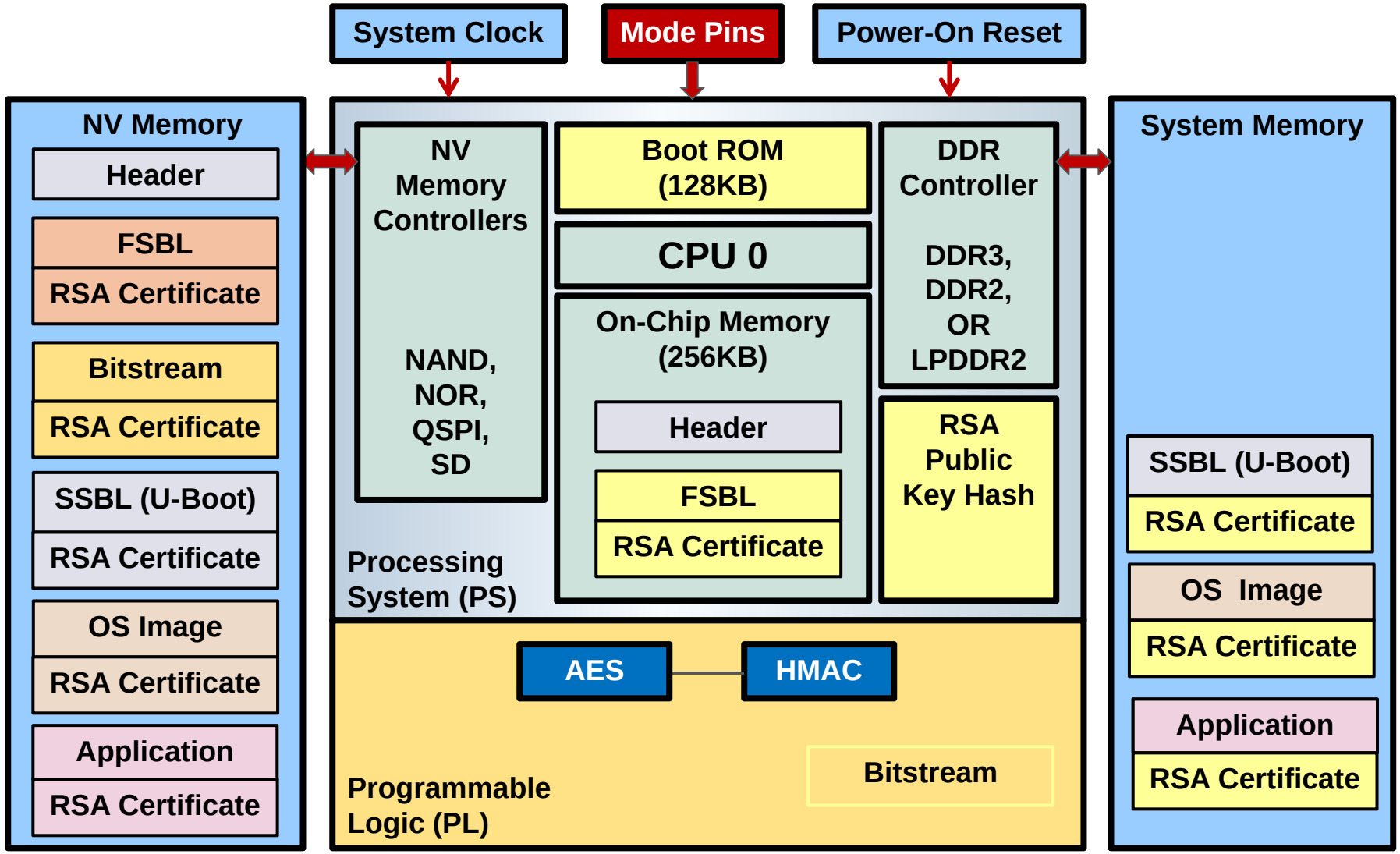
Typical Secure Boot and Configuration Flow



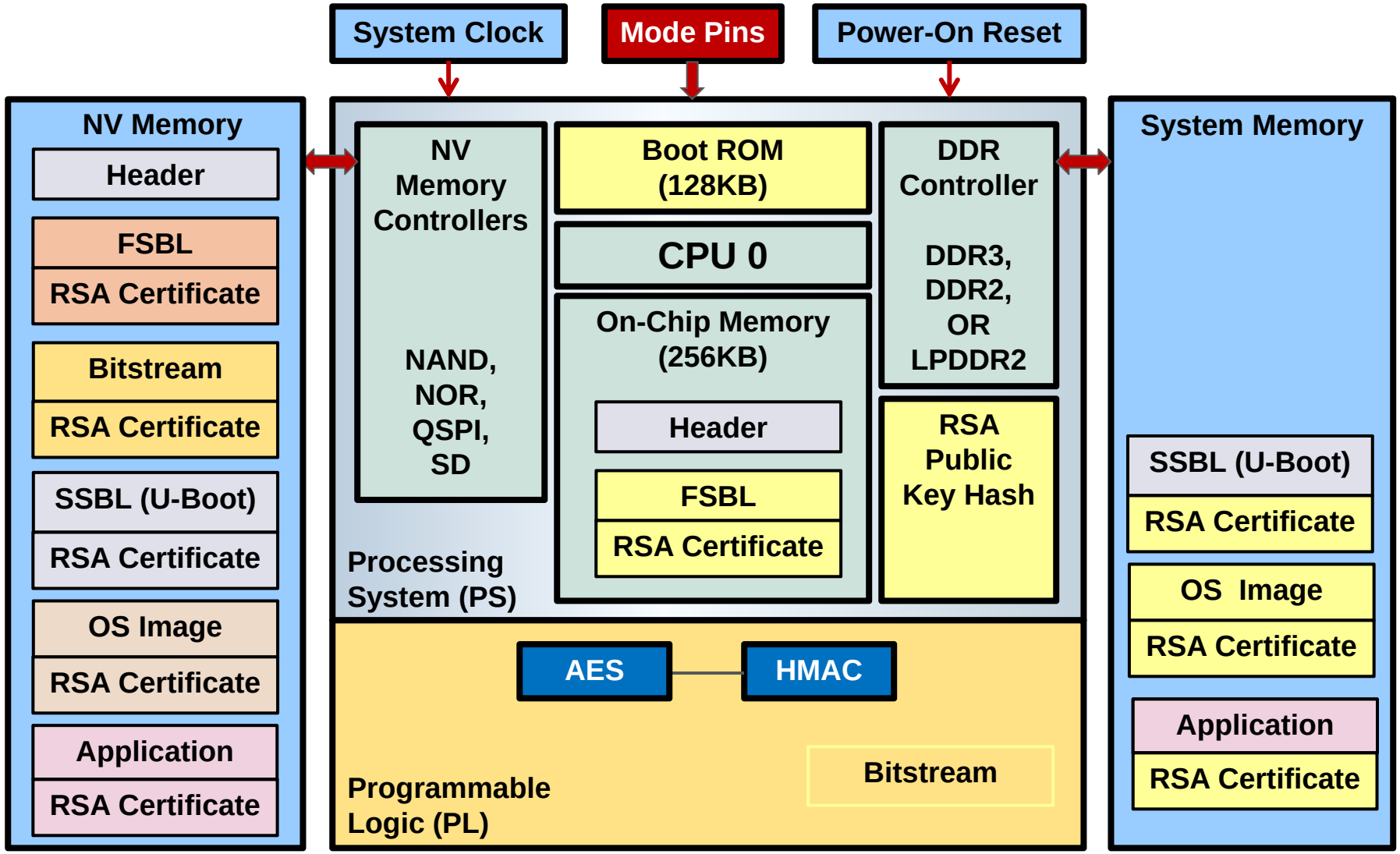
Typical Secure Boot and Configuration Flow



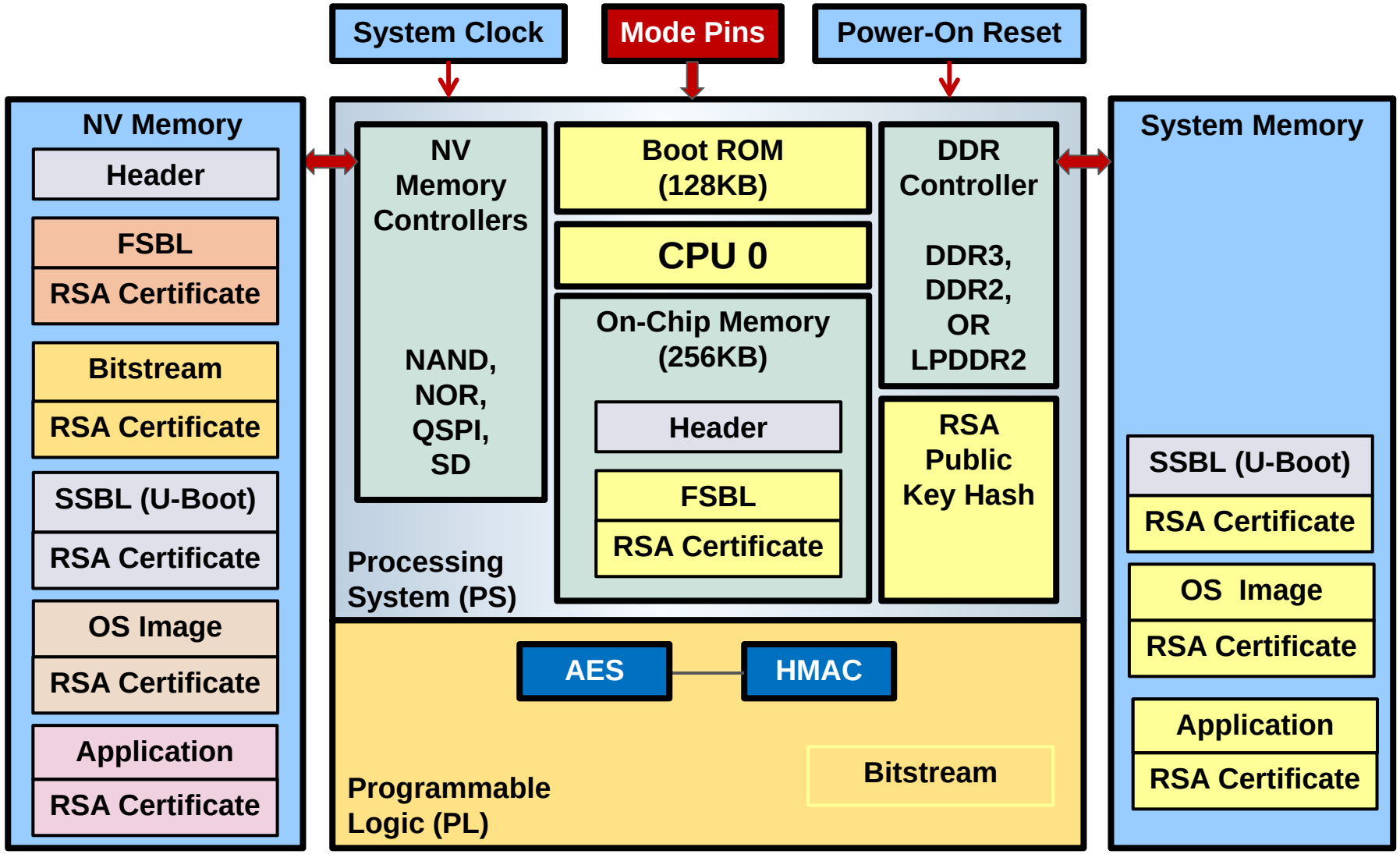
Typical Secure Boot and Configuration Flow



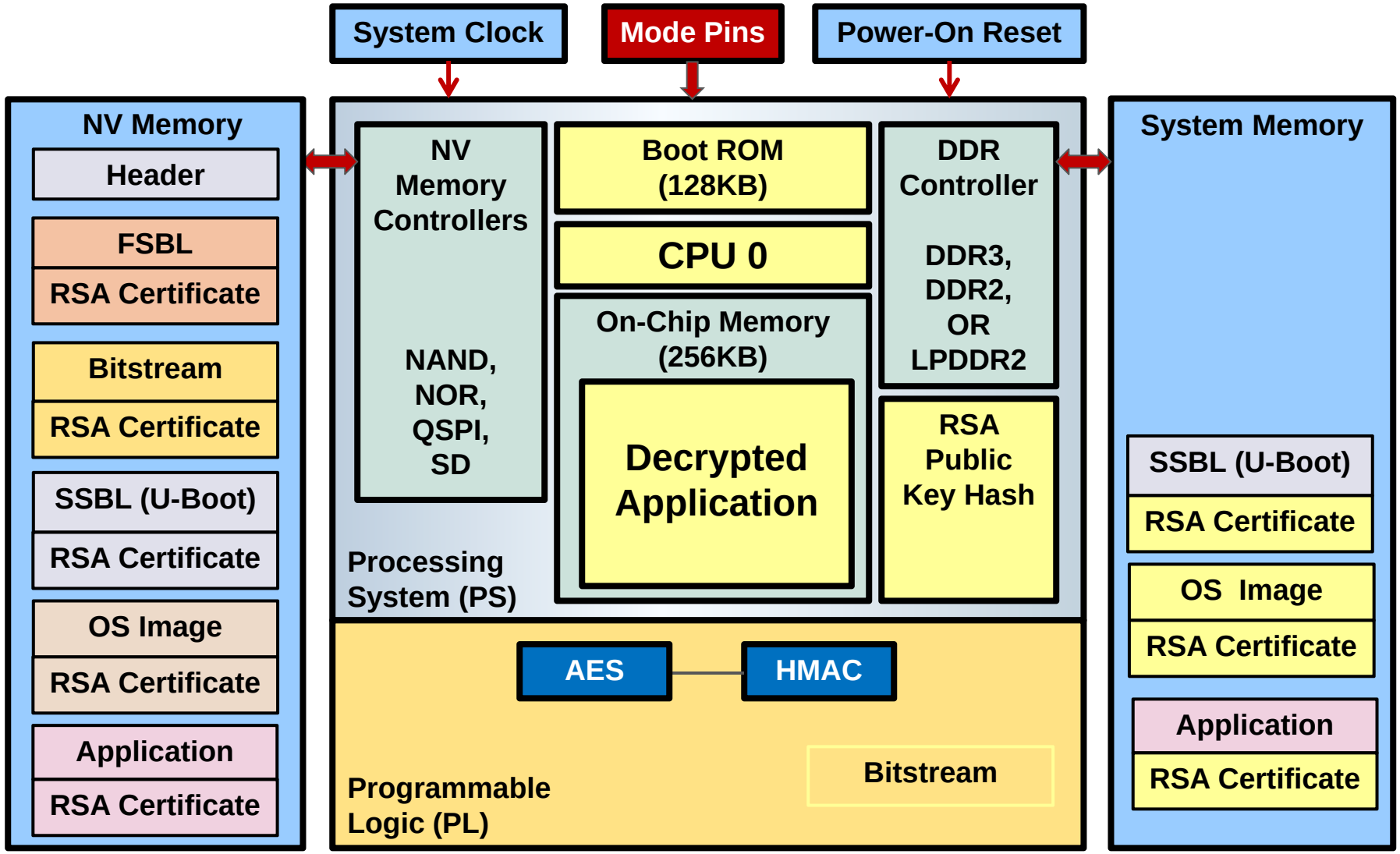
Typical Secure Boot and Configuration Flow



Typical Secure Boot and Configuration Flow



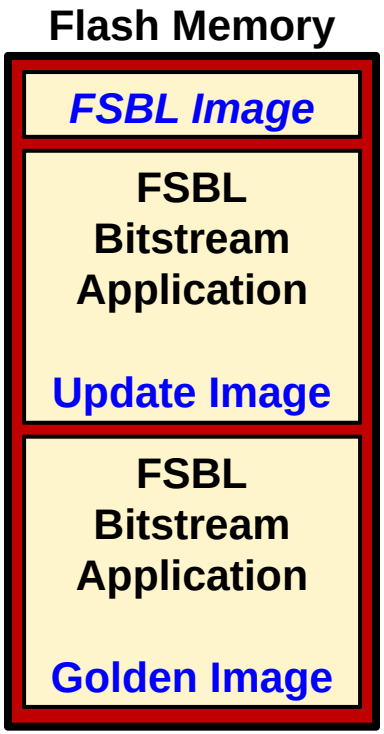
Typical Secure Boot and Configuration Flow



Multi-Boot

Multi-boot Overview

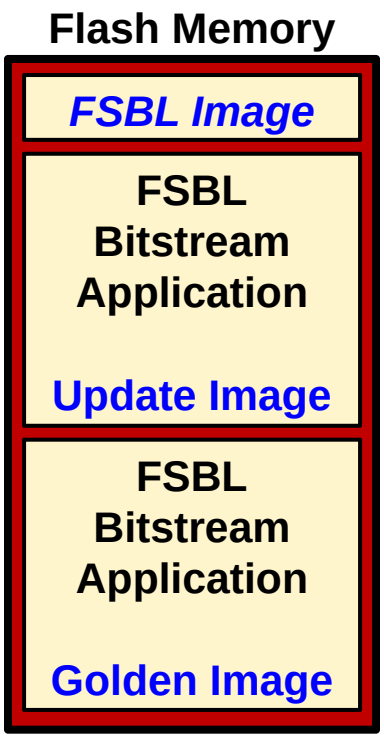
- **Multi-boot is used to ensure that the device boots a *Golden Image* in the event of a failure to boot the *Update Image***



Multi-boot Overview

- **Multi-boot is used to ensure that the device boots a *Golden Image* in the event of a failure to boot the *Update Image***

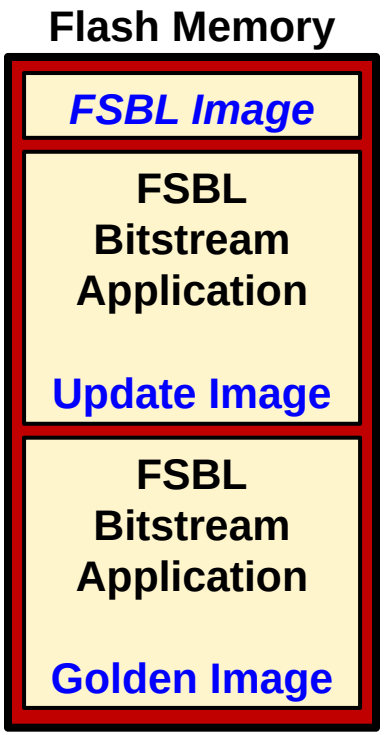
  1) Boot ROM loads the *FSBL Image* into the OCM



Multi-boot Overview

- **Multi-boot is used to ensure that the device boots a *Golden Image* in the event of a failure to boot the *Update Image***

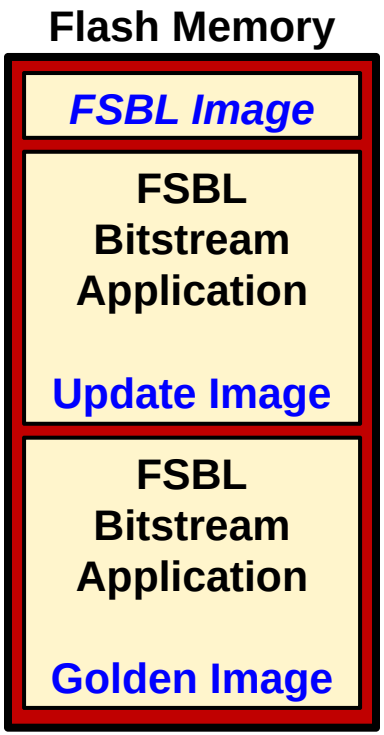
1) Boot ROM loads the *FSBL Image* into the OCM
2) The *FSBL* will set the boot address to the *Update Image* and issues a soft reset



Multi-boot Overview

- **Multi-boot is used to ensure that the device boots a *Golden Image* in the event of a failure to boot the *Update Image***

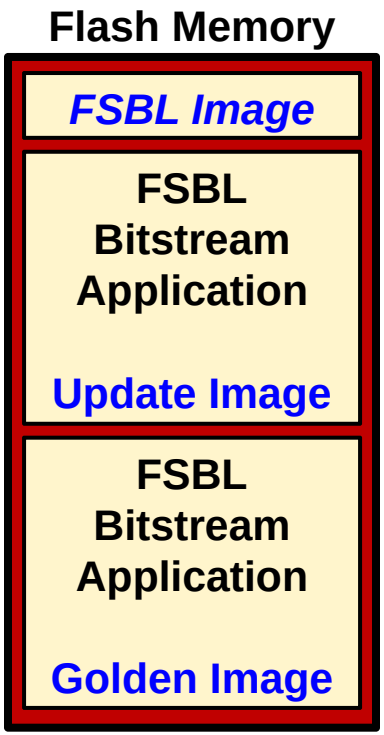
1) Boot ROM loads the *FSBL Image* into the OCM
2) The *FSBL* will set the boot address to the *Update Image* and issues a soft reset
3) Boot ROM loads the *Update Image FSBL* into the OCM



Multi-boot Overview

- **Multi-boot is used to ensure that the device boots a *Golden Image* in the event of a failure to boot the *Update Image***

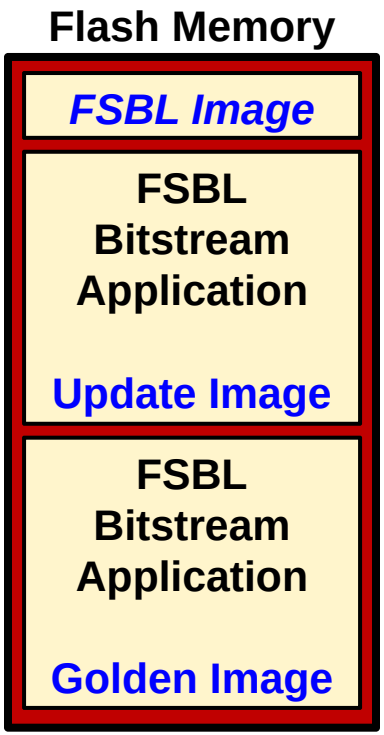
1) Boot ROM loads the *FSBL Image* into the OCM
2) The *FSBL* will set the boot address to the *Update Image* and issues a soft reset
3) Boot ROM loads the *Update Image FSBL* into the OCM
4) The *Update Image FSBL* loads the Bitstream to the PL and the Application to the PS memory



Multi-boot Overview

- **Multi-boot is used to ensure that the device boots a *Golden Image* in the event of a failure to boot the *Update Image***

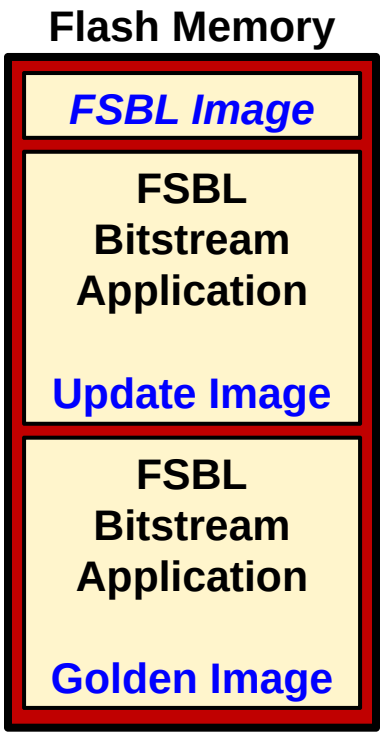
1) Boot ROM loads the *FSBL Image* into the OCM
2) The *FSBL* will set the boot address to the *Update Image* and issues a soft reset
3) Boot ROM loads the *Update Image FSBL* into the OCM
4) The *Update Image FSBL* loads the Bitstream to the PL and the Application to the PS memory
5) If the *Update Image* Bitstream or Application download is not successful, *Update Image FSBL* sets the boot address to the *Golden Image FSBL* and issues a reset



Multi-boot Overview

- **Multi-boot is used to ensure that the device boots a *Golden Image* in the event of a failure to boot the *Update Image***

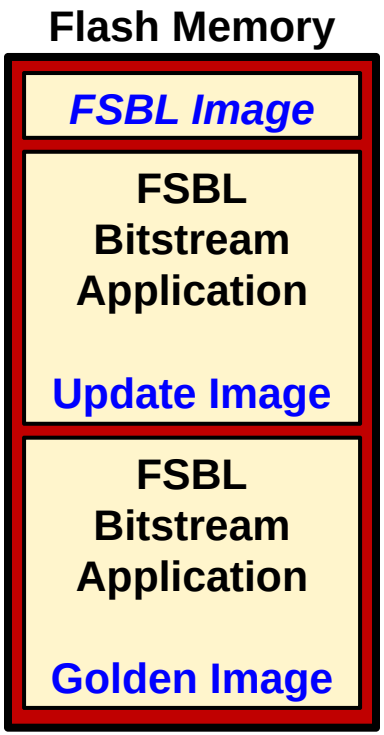
1) Boot ROM loads the *FSBL Image* into the OCM
2) The *FSBL* will set the boot address to the *Update Image* and issues a soft reset
3) Boot ROM loads the *Update Image FSBL* into the OCM
4) The *Update Image FSBL* loads the Bitstream to the PL and the Application to the PS memory
5) If the *Update Image* Bitstream or Application download is not successful, *Update Image FSBL* sets the boot address to the *Golden Image FSBL* and issues a reset
6) Boot ROM loads the *Golden Image FSBL* into the OCM



Multi-boot Overview

- **Multi-boot is used to ensure that the device boots a *Golden Image* in the event of a failure to boot the *Update Image***

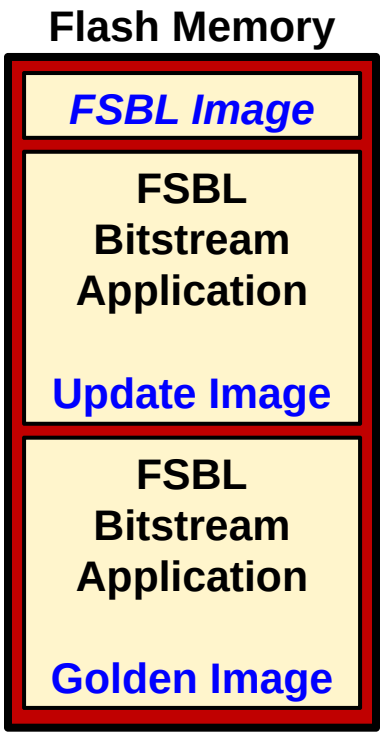
1) Boot ROM loads the *FSBL Image* into the OCM
2) The *FSBL* will set the boot address to the *Update Image* and issues a soft reset
3) Boot ROM loads the *Update Image FSBL* into the OCM
4) The *Update Image FSBL* loads the Bitstream to the PL and the Application to the PS memory
5) If the *Update Image* Bitstream or Application download is not successful, *Update Image FSBL* sets the boot address to the *Golden Image FSBL* and issues a reset
6) Boot ROM loads the *Golden Image FSBL* into the OCM
7) If *Update Image* is completely corrupted, Boot ROM can find the *Golden Image* via *Boot Header Search*



Multi-boot Overview

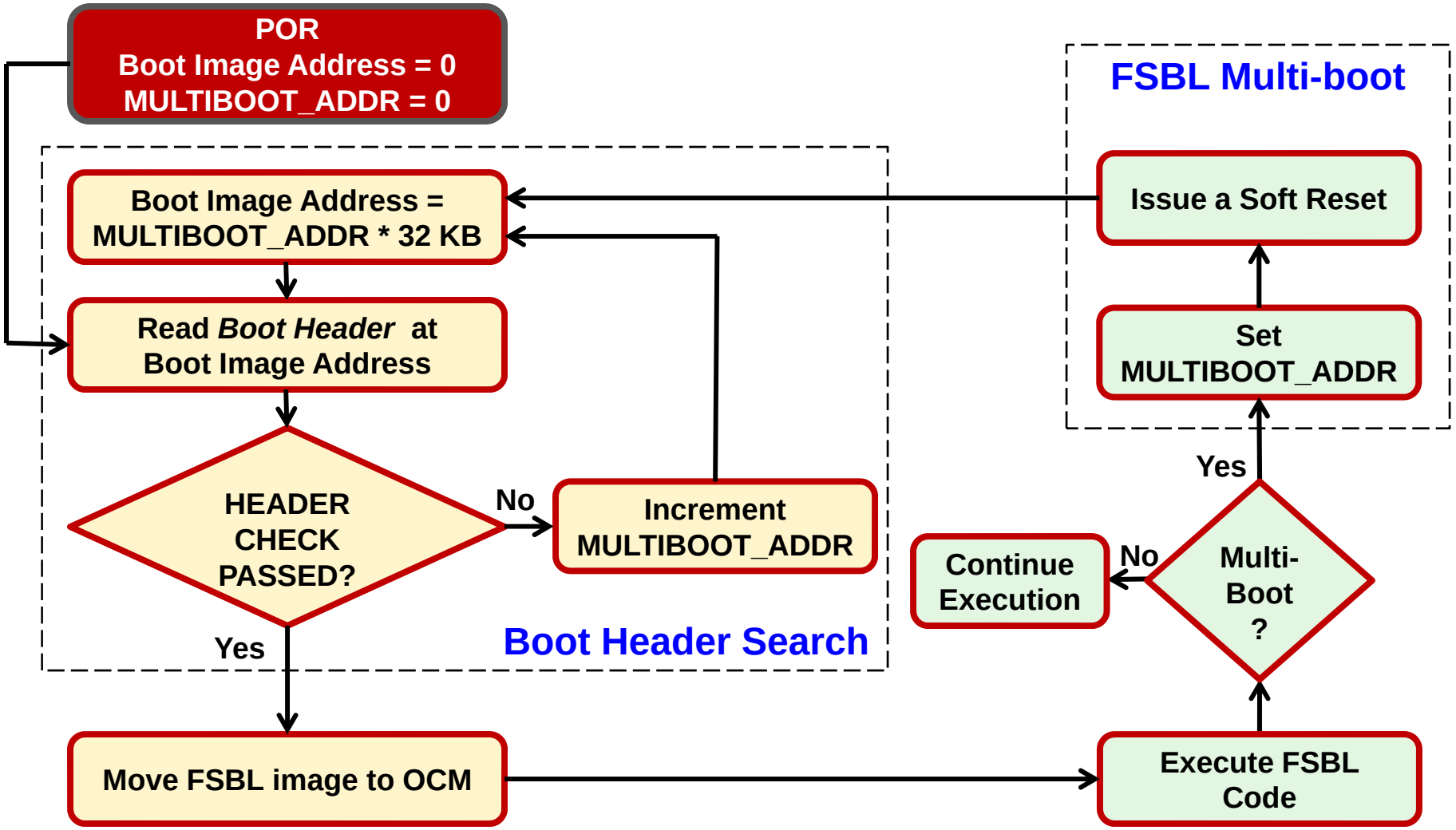
- **Multi-boot is used to ensure that the device boots a *Golden Image* in the event of a failure to boot the *Update Image***

1) Boot ROM loads the *FSBL Image* into the OCM
2) The *FSBL* will set the boot address to the *Update Image* and issues a soft reset
3) Boot ROM loads the *Update Image FSBL* into the OCM
4) The *Update Image FSBL* loads the Bitstream to the PL and the Application to the PS memory
5) If the *Update Image* Bitstream or Application download is not successful, *Update Image FSBL* sets the boot address to the *Golden Image FSBL* and issues a reset
6) Boot ROM loads the *Golden Image FSBL* into the OCM
7) If *Update Image* is completely corrupted, Boot ROM can find the *Golden Image* via *Boot Header Search*



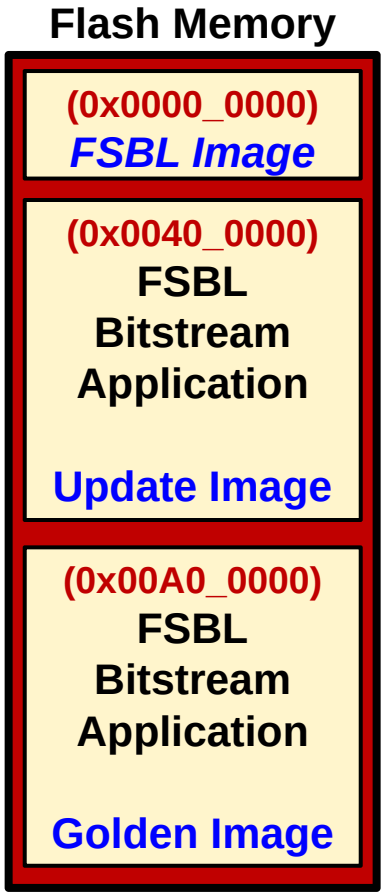
Look for multi-boot example designs in *UG1025*, *XAPP1175*, and http://www.wiki.xilinx.com/Zynq-7000+AP+SoC+Multiboot+Tech+Tip

Multi-boot Flow



Multi-boot Example

- **Use the Bootgen tool to generate *fsbl.MCS*, *update_image.MCS*, and *golden_image.MCS***
  - SDK Flash programmer can be used three times to program the Flash with the above MCS files at offsets 0x0000_0000, 0x0040_0000, and 0x00A0_0000

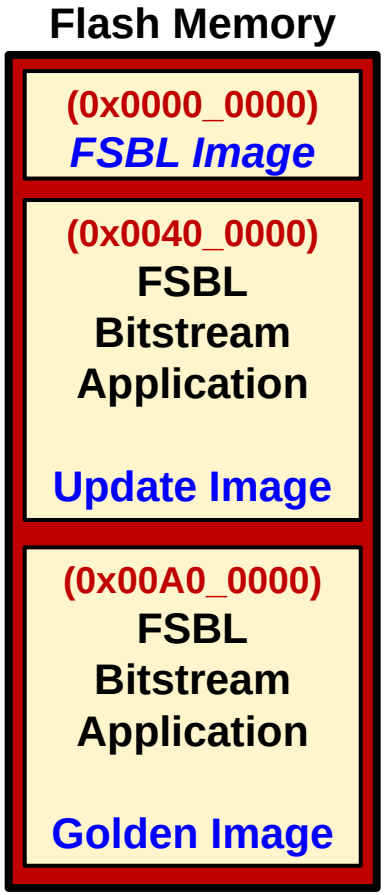


Multi-boot Example

- **Use the Bootgen tool to generate *fsbl.MCS*, *update_image.MCS*, and *golden_image.MCS***
  - SDK Flash programmer can be used three times to program the Flash with the above MCS files at offsets 0x0000_0000, 0x0040_0000, and 0x00A0_0000

- **U-Boot can also be used to program the Flash**
  - Use the Bootgen tool to generate *Boot.BIN*, *fsbl.BIN*, *update_image.BIN*, and *golden_image.BIN* images
  - Place these images on an SD card and boot the target board (*Boot.BIN* image consists of FSBL and U-Boot)

```
zynq-uboot> mmcinfo
zynq-uboot> fatload mmc 0 0x100000 fsbl.bin
zynq-uboot> sf probe 0 0 0
zynq-uboot> sf write 0x100000 0 0x20000
zynq-uboot> fatload mmc 0 0x100000 update_image.bin
zynq-uboot> sf write 0x100000 0x400000 ${filesize}
zynq-uboot> fatload mmc 0 0x100000 golden_image.bin
zynq-uboot> sf write 0x100000 0xA00000 ${filesize}
```



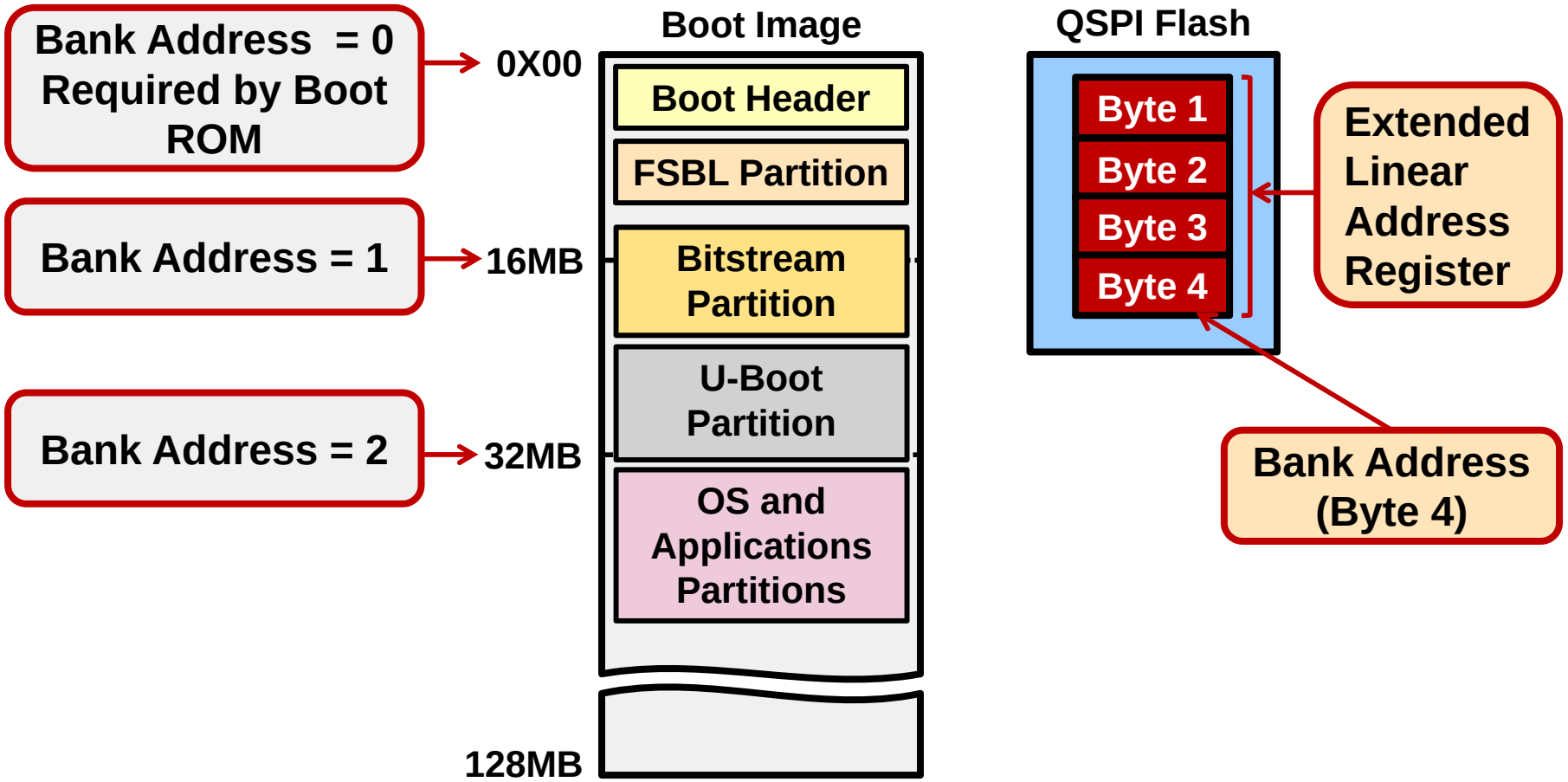
Boot and Configuration Devices

Booting From QSPI Flash

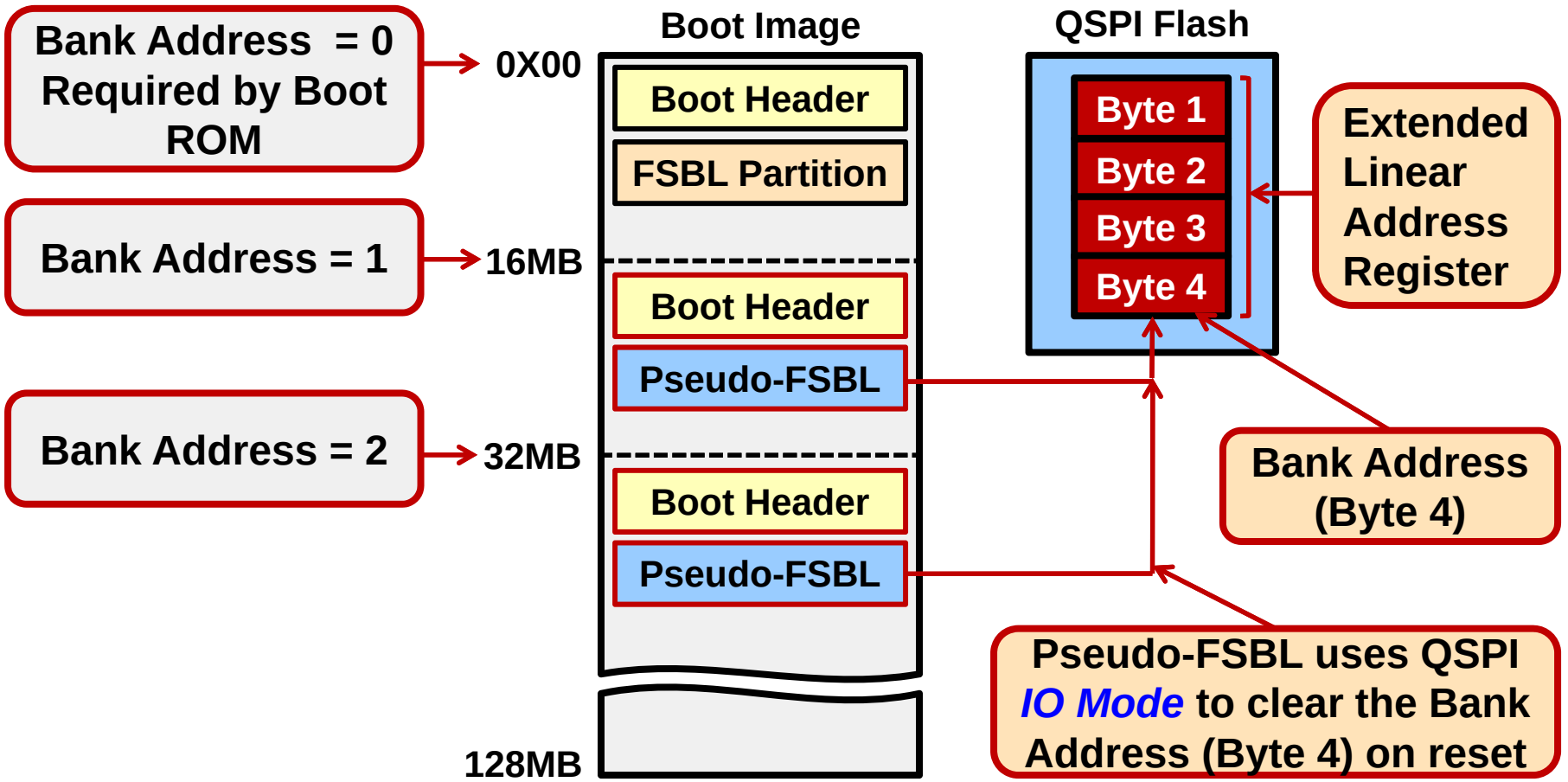
- **Advantages of QSPI Flash**
  - *High performance* - QSPI is the fastest boot/configuration solution
  - *Low pin count* - QSPI interface has low pin count
  - *Easy management* - QSPI can be accessed as linear memory in Zynq
  - *Execute-in-place (XIP)* – QSPI Flash supports Zynq XIP feature

- **Boot ROM uses the QSPI 24-bit (3 bytes) *Linear Addressing Mode* to load the FSBL**
  - This implies FSBL image must be placed in the first 16MB of a single QSPI or the first 32MB of a dual QSPI for devices larger than 16MB
  - Memory above 16MB for a single QSPI device and 32MB for dual QSPI configuration can be accessed after the Boot ROM passes control to FSBL
    - FSBL and SSBL use the QSPI *Extended Linear Addressing Mode* (4-byte address) or *IO Mode* to access the QSPI memory above 16MB/32MB

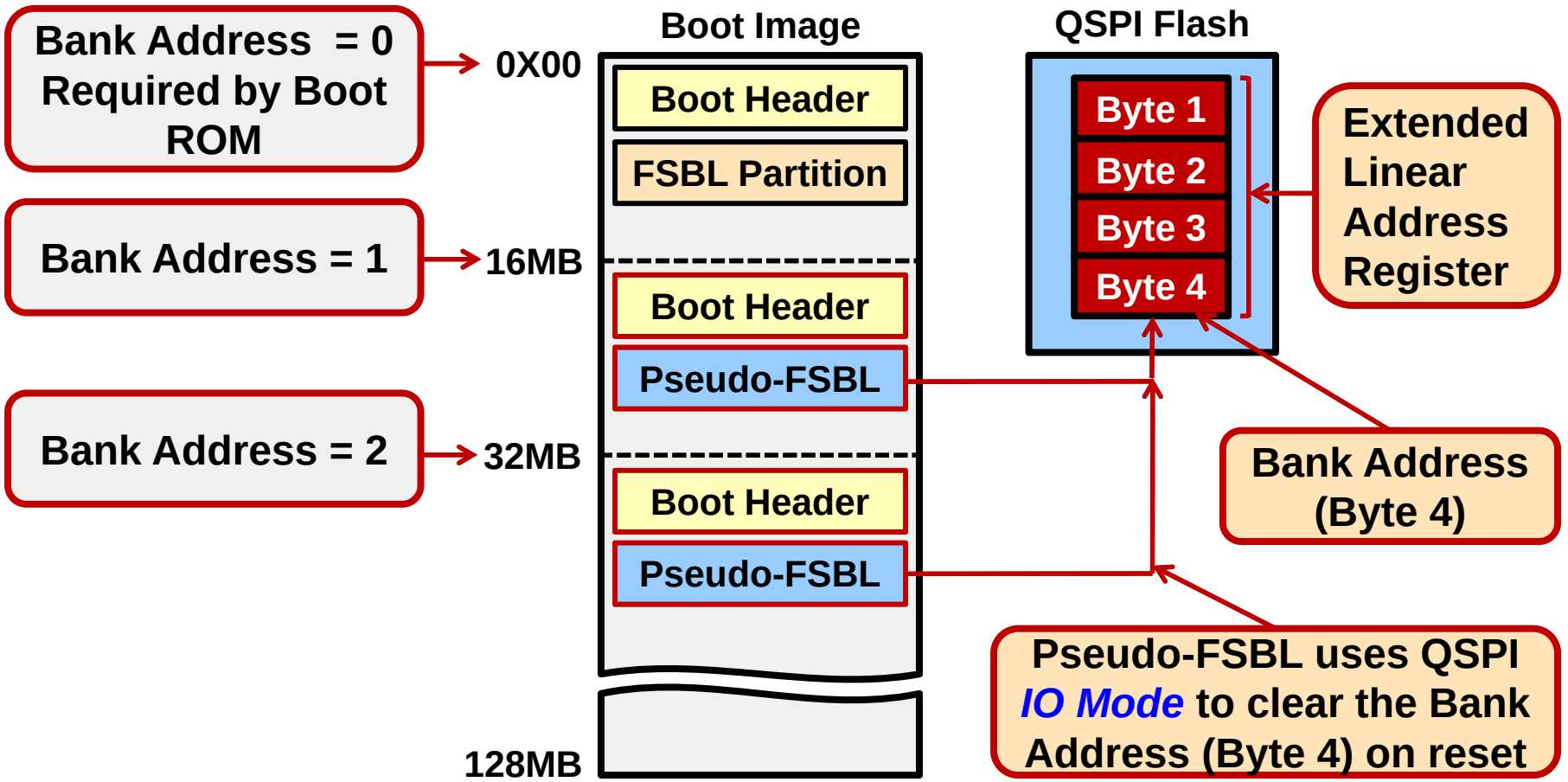
Booting From Larger than 16MB QSPI Devices



Booting From Larger than 16MB QSPI Devices



Booting From Larger than 16MB QSPI Devices



Spansion Alternative Software Solution to *Answer Record 57744*

Xilinx Supported QSPI Flash Devices

- **Xilinx supports the following families of QSPI Flash devices**

Vendor	QSPI Flash Families	Maximum Density
Micron	N25Q	128 MB
Spansion	S25FL and S70FL	128 MB

- **QSPI controller supports the following memory configurations**
  - *Single Mode* - QSPI device must be connected to MIO[1:6, 8]
  - *Dual Stacked Mode* - QSPI devices must be connected to MIO[0:6, 8]
  - *Dual Parallel Mode* - QSPI devices must be connected to MIO[0:6, 8:13]

Memory Configuration	Required MIO Pins	Max Memory Size (Linear Mode)	Max Memory Size (IO/Extended Linear Mode)
Single Mode	7	16 MB	128 MB
Dual Stacked Mode	8	32 MB	256 MB
Dual Parallel Mode	13	32 MB	256 MB

Improving QSPI Boot Time

- **Upon reset PS control registers are not initialized by default to operate the QSPI controller for optimal performance**
  - Normally, the register settings in the FSBL are used to change the PS default register settings and optimize the QSPI access time
  - The *Boot Header Register Initialization* feature can be used to optimize the PS control register settings prior to FSBL execution
    - This improves FSBL load time, XIP operation, and Boot ROM execution

Improving QSPI Boot Time

- **Upon reset PS control registers are not initialized by default to operate the QSPI controller for optimal performance**
  - Normally, the register settings in the FSBL are used to change the PS default register settings and optimize the QSPI access time
  - The *Boot Header Register Initialization* feature can be used to optimize the PS control register settings prior to FSBL execution
    - This improves FSBL load time, XIP operation, and Boot ROM execution

Example of PS Control Register Settings to Improve QSPI Boot Time

Register Name	Register Address	Example of Improved Value	Description
ARM_CLK_CTRL	0xF8000120	0x1F000200	CPU Clock = 433 MHz
LQSPI_CLK_CTRL	0xF800014C	0x00000521	QSPI Ref Clock = 173 MHz
Config_reg	0xE000D000	0x800238C1	QSPI Clock = 86 MHz

Improving QSPI Boot Time

- **Upon reset PS control registers are not initialized by default to operate the QSPI controller for optimal performance**
  - Normally, the register settings in the FSBL are used to change the PS default register settings and optimize the QSPI access time
  - The *Boot Header Register Initialization* feature can be used to optimize the PS control register settings prior to FSBL execution
    - This improves FSBL load time, XIP operation, and Boot ROM execution

Example of PS Control Register Settings to Improve QSPI Boot Time

Register Name	Register Address	Example of Improved Value	Description
ARM_CLK_CTRL	0xF8000120	0x1F000200	CPU Clock = 433 MHz
LQSPI_CLK_CTRL	0xF800014C	0x00000521	QSPI Ref Clock = 173 MHz
Config_reg	0xE000D000	0x800238C1	QSPI Clock = 86 MHz

For example, *.set. 0xF8000120 = 0x1F000200;* in the *.INIT* file will set the CPU clock to 433 MHz (Default CPU clock is 216 MHz)

Booting From SD Card

- **Advantages of SD Card**
  - *High density* – Up to 32 GB card density
  - *Easy Management* - Device is generally managed as a file system
  - *Low pin count* – SD card interface has low pin count
- **Disadvantages of SD Card**
  - *Slow performance* - SD is the slowest boot/configuration solution
  - *Mechanical considerations* - SD card requires a connector
  - SD boot mode does not support *Boot Header Search* or *Multi-boot*
- **SD controller supports the following memory configuration**

Memory Configuration	Required MIO Pins	Must be Connected to
SD Card	6	MIO[40:45] and SD 0 Controller

  - MIO pins for the *Card Detect (CD)* and optional *Write Protect (WP)* signals are user selectable

Choosing the Right SD Card

- **Not all SD cards are created equal**
  - Zynq SD controller starts the SD clock and issues CMD0 after 3.5 SD clocks (*AR52023*)
    - This can cause a boot failure as some SD cards require 74 clocks before CMD0 is issued (most SD cards work with 3.5 clocks)
  - microSD cards don't have the WP pin, SD boot will fail if the SD controller WP signal is not driven low (*AR59316*, fixed in 2014.1)
    - Use a spare MIO pin to emulate the WP pin (connect it to GND)
    - Assign the WP signal to an EMIO pin in the PS Configuration Wizard
  - SD card manufacturer and type will play a significant role in the SD card performance

SD Card Class	Class 4	Class 10
Performance (MB/s)	6.3	11.5

Choosing the Right SD Card

- **Not all SD cards are created equal**
  - Zynq SD controller starts the SD clock and issues CMD0 after 3.5 SD clocks (*AR52023*)
    - This can cause a boot failure as some SD cards require 74 clocks before CMD0 is issued (most SD cards work with 3.5 clocks)
  - microSD cards don't have the WP pin, SD boot will fail if the SD controller WP signal is not driven low (*AR59316*, fixed in 2014.1)
    - Use a spare MIO pin to emulate the WP pin (connect it to GND)
    - Assign the WP signal to an EMIO pin in the PS Configuration Wizard
  - SD card manufacturer and type will play a significant role in the SD card performance

SD Card Class	Class 4	Class 10
Performance (MB/s)	6.3	11.5

SanDisk **and** ***PNY*** **are our recommended SD cards for Zynq applications**

Booting From NAND or NOR Flash

- Xilinx supports the following families of NAND and NOR Flash devices

Vendor	NAND Flash Families/ Maximum Density	NOR Flash Families/ Maximum Density
Micron	MT29F/1GB	M29EW/64MB
Spansion	S34/512MB	29GL/64MB

- NAND controller supports the following memory configurations

Memory Configuration	Required MIO Pins	Must be Connected to
NAND x8	15	MIO[0:14]
NAND x16	23	MIO[0:14,16:23]

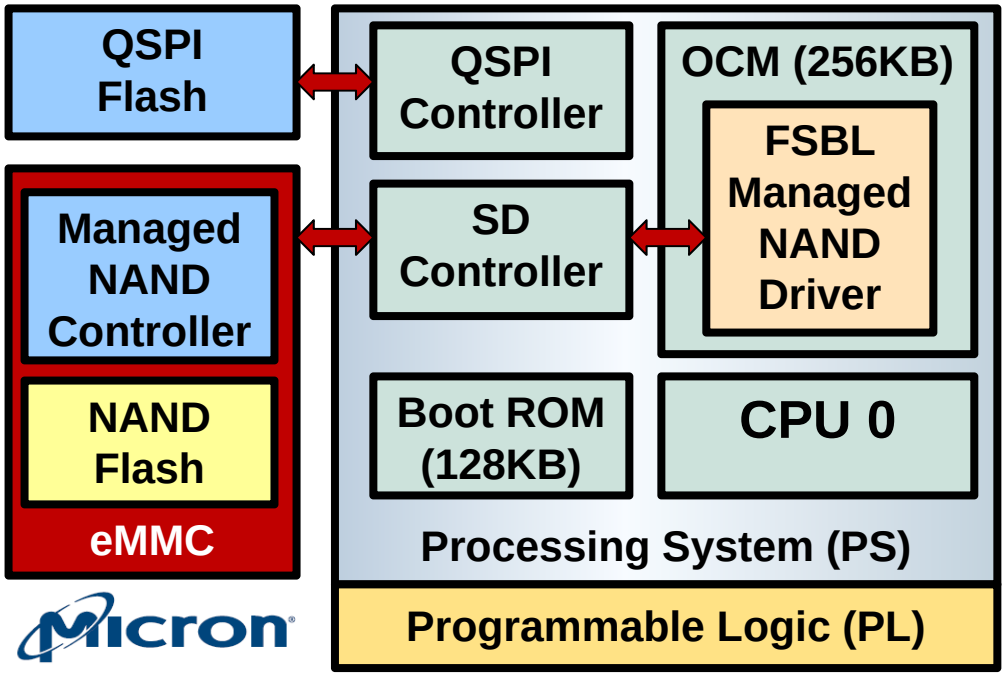
- NOR controller supports the following memory configuration

Memory Configuration	Required MIO Pins	Must be connected to
NOR x8	40	MIO[0:39]

Zynq eMMC Support

- **Zynq supports eMMC Flash in MLC and SLC configuration as a secondary boot source**
  - A small QSPI Flash is used to store the FSBL while all the other boot partitions are stored on the eMMC
  - Boot ROM loads FSBL from QSPI into OCM while FSBL loads all other partitions from eMMC into the system DDR memory (see *UG821*)

- **Micron MTFC eMMC**
  - MLC NAND Flash
  - eMMC v4.41 and v4.51 Managed NAND Controller
  - Host selectable x1, x4, x8 interface
  - Clock speed up to 200 MHz
  - Data rate up to 130 MB/s
  - Densities up to 64 GB



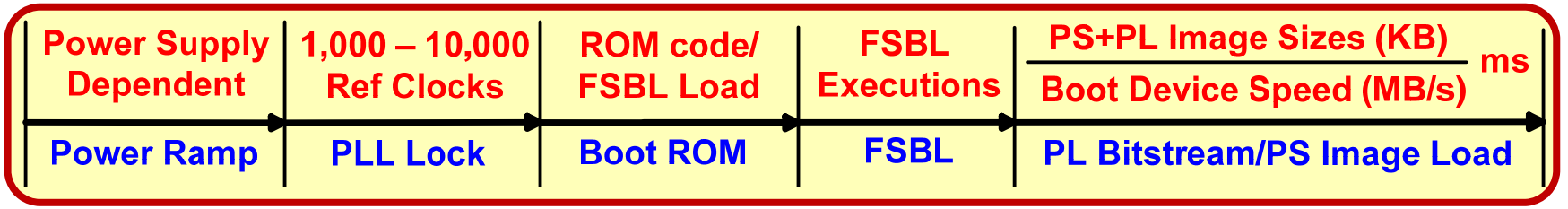
Boot ROM Execution and FSBL Image Copy Time

- **Boot ROM execution and FSBL image copy time for 128KB of FSBL image and a PS_CLK of 33.33 MHz**
  - Table shows Boot ROM execution and FSBL load times for default and optimized register values using the *Boot Header Register Initialization*
  - Boot time is from POR de-assertion until Boot ROM branches to the FSBL image in OCM

Non-Secure Boot Mode	Default Register Initialization (ms)	Optimized Register Initialization (ms)
QSPI Single	98.4	16
QSPI Dual	72	12
NAND x8	114	52
NAND x16	92	50
NOR	72	12
SD Card	216	196

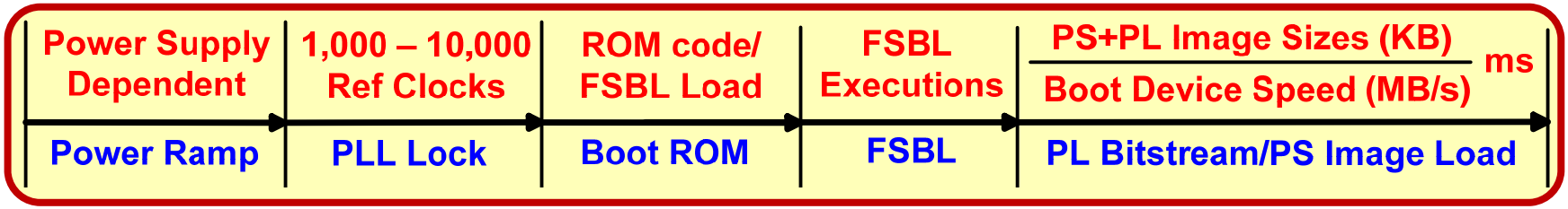
Zynq Boot and Configuration Time

- **Boot and configuration times are determined by the selected boot mode and non-volatile memory used, along with**
  - *Power supply ramp time* – power supply dependent, typically 50ms
  - *PLL Lock* – 300us max using a 33.33 MHz reference clock input
  - *Boot ROM code execution/FSBL load* – 16ms (single QSPI, 128KB FSBL)
  - *FSBL execution* – FSBL initializing memory/peripheral controllers (< 10ms)
  - *PL bitstream*, *PS image load time* – depends on bitstream and image sizes
  - *Boot Header Register Initialization* – optimizes NV memory accesses



Zynq Boot and Configuration Time

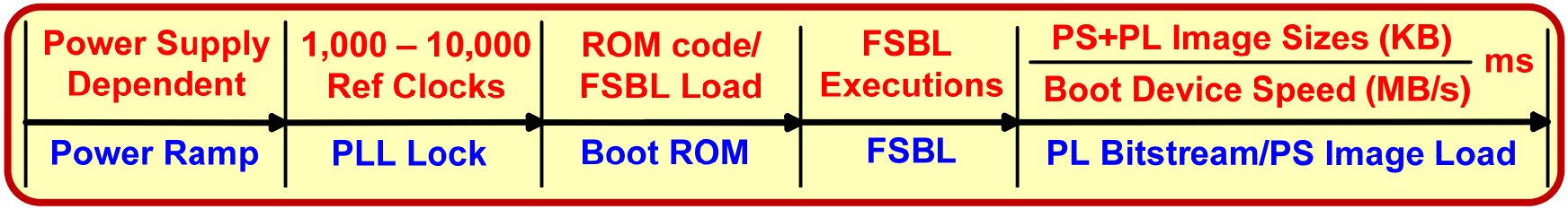
- **Boot and configuration times are determined by the selected boot mode and non-volatile memory used, along with**
  - *Power supply ramp time* – power supply dependent, typically 50ms
  - *PLL Lock* – 300us max using a 33.33 MHz reference clock input
  - *Boot ROM code execution/FSBL load* – 16ms (single QSPI, 128KB FSBL)
  - *FSBL execution* – FSBL initializing memory/peripheral controllers (< 10ms)
  - *PL bitstream*, *PS image load time* – depends on bitstream and image sizes
  - *Boot Header Register Initialization* – optimizes NV memory accesses



PL bitstream and PS image can be loaded in parallel using a *QSPI device for PL bitstream* and an *SD card/eMMC for PS image*

Zynq Boot and Configuration Time

- **Boot and configuration times are determined by the selected boot mode and non-volatile memory used, along with**
  - *Power supply ramp time* – power supply dependent, typically 50ms
  - *PLL Lock* – 300us max using a 33.33 MHz reference clock input
  - *Boot ROM code execution/FSBL load* – 16ms (single QSPI, 128KB FSBL)
  - *FSBL execution* – FSBL initializing memory/peripheral controllers (< 10ms)
  - *PL bitstream*, *PS image load time* – depends on bitstream and image sizes
  - *Boot Header Register Initialization* – optimizes NV memory accesses



PL bitstream and PS image can be loaded in parallel using a *QSPI device for PL bitstream* and an *SD card/eMMC for PS image*

Please refer to the *Answer Record 54833* for information on *Tandem Configuration* for PCIe applications

Next Steps

Next Step

- **For more information on *Zynq Boot and Configuration* process, please refer to the following documents/application notes**
  - *UG585*, *UG821*, *UG1025*, and *XAPP1175*

- **Please visit www.zedboard.org web site for information on Avnet Zynq developments boards and SoMs**

Mini Module Plus

Mini-ITX Motherboard

MicroZed

Next Step

- **For more information on *Zynq Boot and Configuration* process, please refer to the following documents/application notes**
  - *UG585*, *UG821*, *UG1025*, and *XAPP1175*

- **Please visit www.zedboard.org web site for information on Avnet Zynq developments boards and SoMs**

Mini Module Plus

Mini-ITX Motherboard

MicroZed

All X-Fest 2014 presentations will be available on www.xfest2014.com


Thank
You
Asante
Shukria
Dhanyavadagalu
Vinaka
köszönöm
Kiitos
Maake
Manana
Dankon
Matondo
Kam Sah Hammida
감사합니다
Dank Je
Dankscheen
спасибо
Maururu
Biyan
Terima Kasih
Taiku
Tack
Blagodaram
Ngiyabonga
Dziekuje
Chokrane
Diolch i Chi
Grazie
Juspaxar
Arigato
Mochchakkeram
Gracias
Khap
Tingki
நன்றி
धन्यवाद
cảm ơn bạn
Paldies
Gratias Tibi
Ua Tsaug Rau Koj
Bedankt
Dakujem
Grazas
Kop Khun
Obrigado
Děkuji
Nirringrazzjak
Di Ou Mèsi
Kia Ora
ありがとう
Suksama
Nuwun
Hvala
Djiere Dieuf
Rahmat
谢
Welalin
Matur
Misaotra
Danke
Go Raibh Maith Agat
Eskerrik Asko
Merci
ขอขอบคุณคุณ
Salamat
Najis Tuke
谢
хвала