PlayStation 3 Emulation

(Re)implementing the impossible

Alexandro Sanchez Bach

$ whoami

- Independent security consultant
- Hypervisor/emulator developer
  - 2013 – 2015 RPCS3
  - 2014 – 2017 Nucleus
  - 2017 – Now Orbital

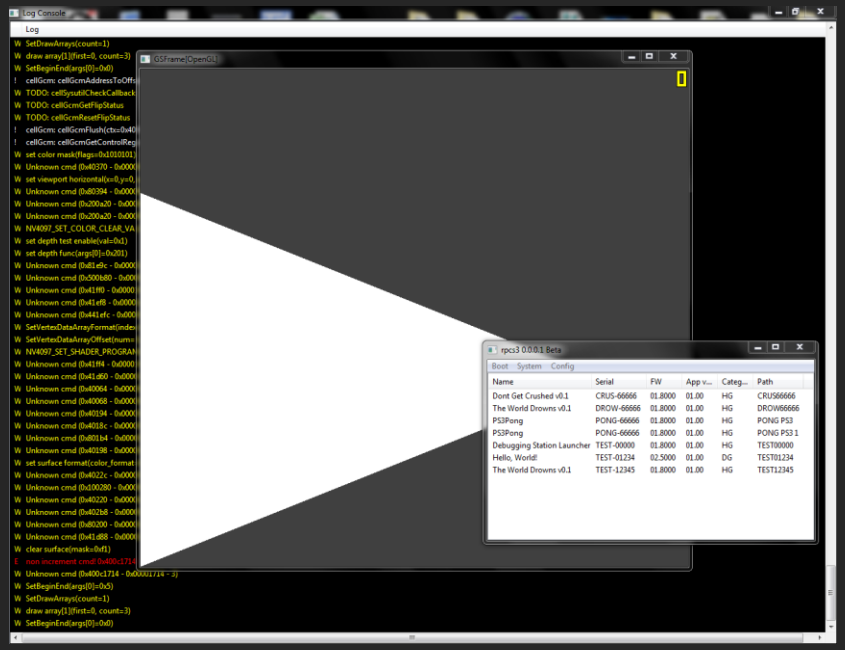
PlayStation 3 emulators

FOSS

$ uptime

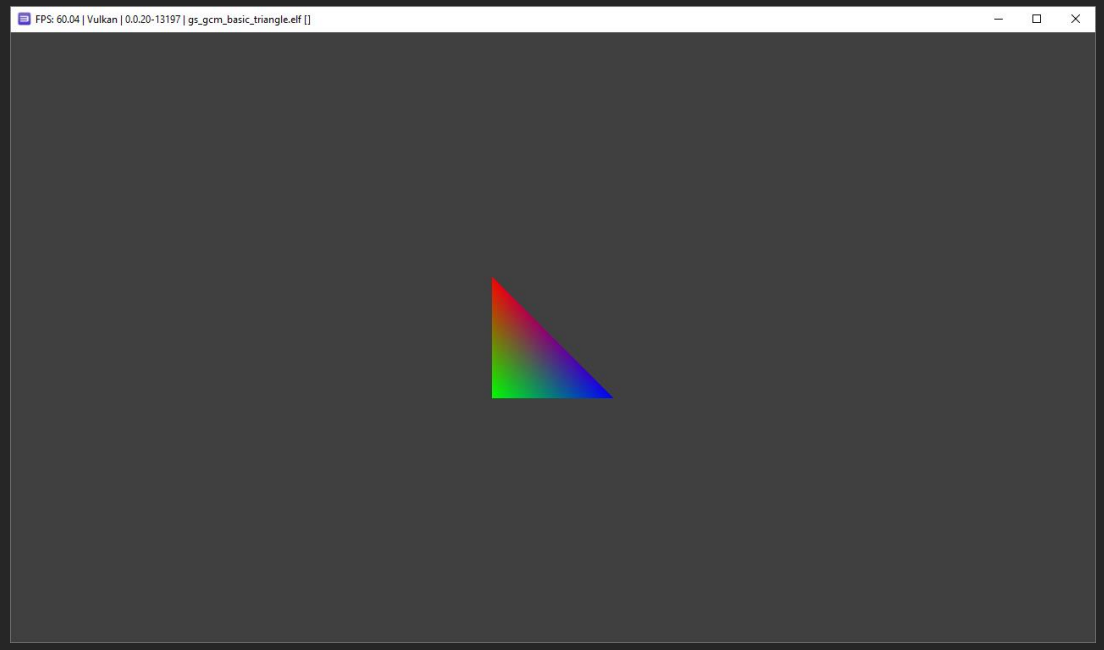
- RPCS3 became 10 years old!

such progress 😮



~2011

RPCS3 v0.0.0.1 Beta

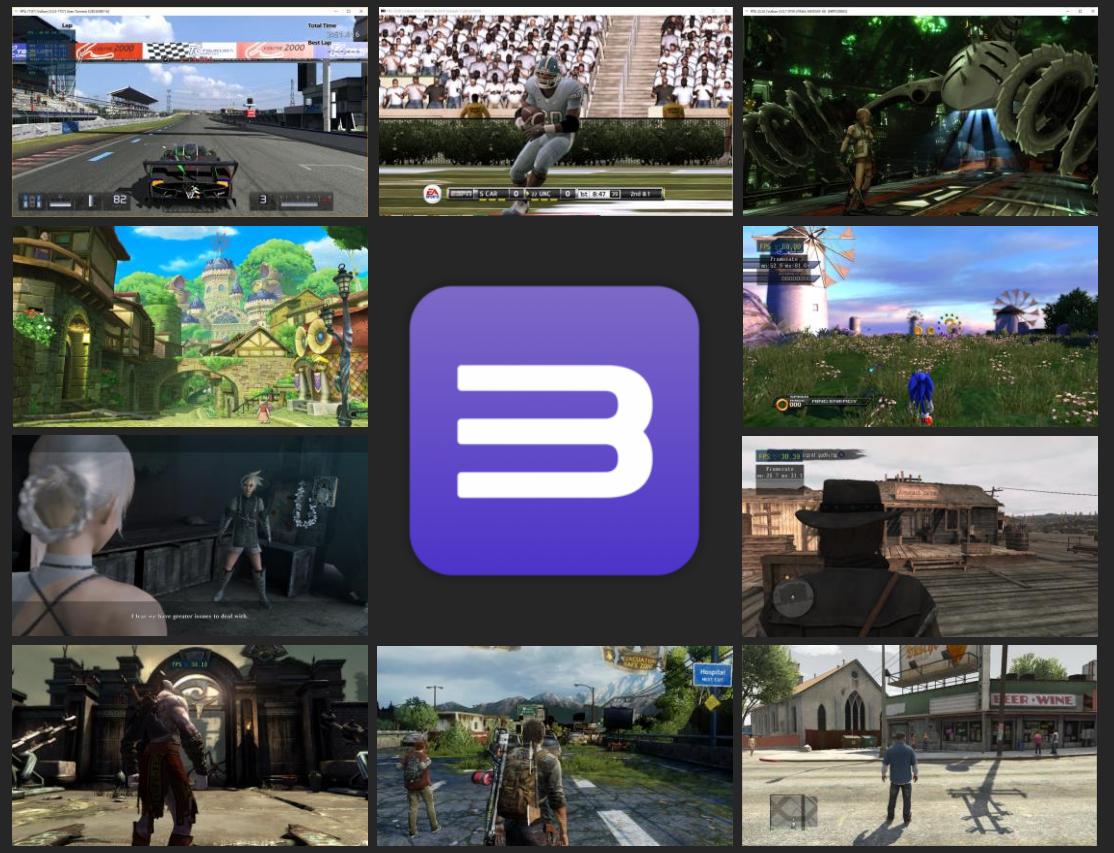


~2021

RPCS3 v0.0.20 Alpha

$ uptime

- RPCS3 became 10 years old!
- This would not be possible without the relentless efforts of the current maintainers, developers, admins, contributors and supporters.
- **Disclaimer**: For the last years I have not been active in RPCS3/PS3 scene.
  - Many thanks to the RPCS3 team for their support.



~2021

RPCS3 v0.0.20 Alpha

$ ls

Disclaimer: Hard to fit in <1 hour

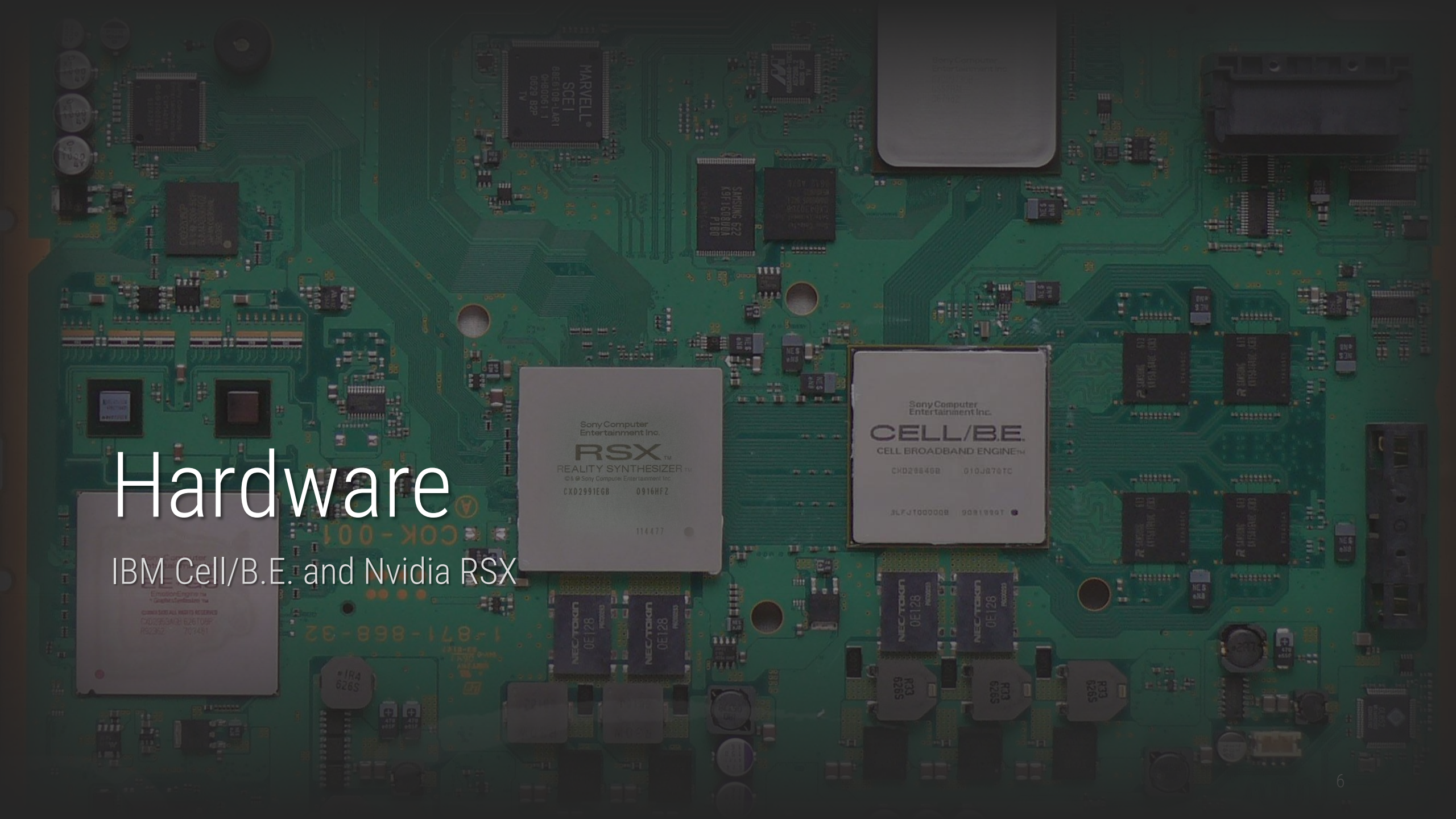
Agenda

- PS3 hardware:
  - IBM Cell/B.E. (CPU)
  - Nvidia RSX (GPU)
- PS3 software:
  - CellOS Lv-1 (Hypervisor)
  - CellOS Lv-2 (Kernel) and userland
- PS3 emulators, mostly RPCS3:
  - Design strategies, evolution over time
  - Tooling, testing, and clever tricks

Disclaimer: Hard to fit everyone's background

Motivation

- Demystifying PS3 software/hardware for emulator developers:
  - ×20 transistors/bytes ≠ ×20 complexity
  - Drivers are thinner than they seem!
- Mapping new/unknown concepts into old/existing knowledge
- Spreading brilliant-yet-hidden ideas
- Embracing *impossible* projects

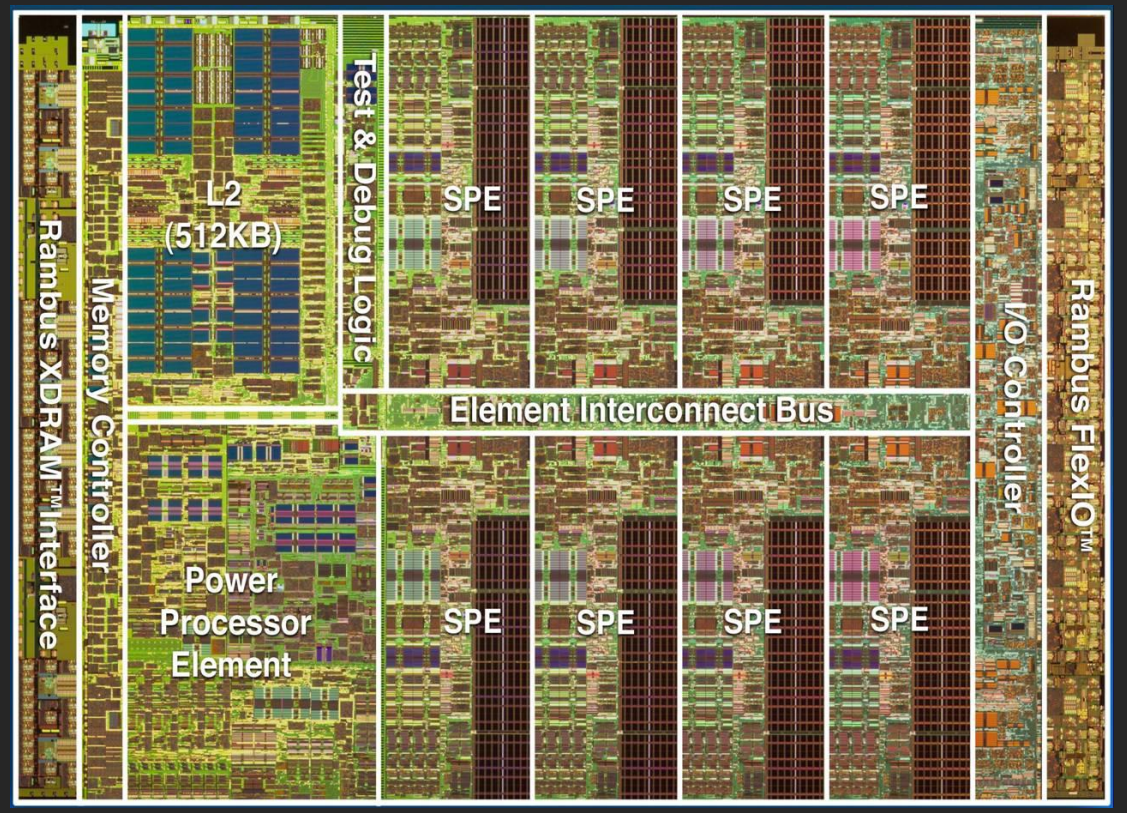


Hardware

IBM Cell/B.E. and Nvidia RSX

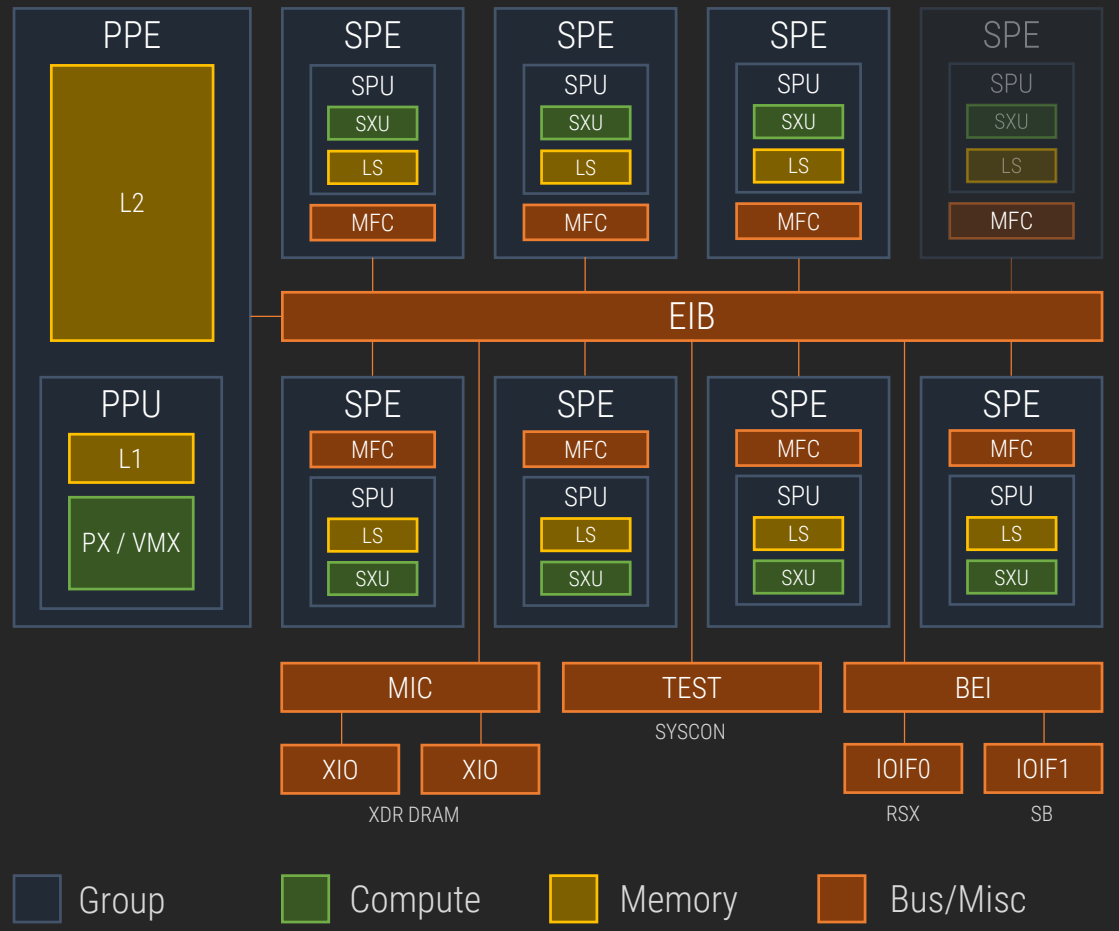
IBM Cell/B.E.

- Design goals
  - ISA-heterogeneous CPU with:
    - PPE: Power Processor Element (*leader*)
    - SPE: Synergistic Processing Element (*assistant*)
  - Prioritizes bandwidth over latency
- Developed by Sony, Toshiba, and IBM (STI)
  - Beginning in March 2001
    - 4-year process
    - 400 million USD budget
    - 400 people
  - Ambitious initial design: 4 PPEs / 32 SPEs achieving theoretical 1 teraFLOPS
    - Scientific computing
    - Software rendering
- Public docs/specifications!



IBM Cell/B.E.

- 9-core heterogeneous 3.2GHz CPU
  - 1 Power Processor Element (PPE)
  - 8 Synergistic Processing Elements (SPEs)
    - 1 disabled to improve chip yield
    - 1 reserved to system
- 3 interfaces to external hardware:
  - Memory Interface Controller (MIC) for 256 MiB Rambus XDR DRAM
  - Broadband Engine Interface (BEI) with 2 Rambus FlexIO interfaces:
    - IOIF0 for Nvidia RSX (GPU)
    - IOIF1 for Toshiba SCC (Southbridge)
  - TEST/DBG for Syscon
- 12-node Element Interconnect Bus (EIB)



IBM Cell/B.E. > PPE

- 64-bit PowerPC (Big-Endian)
- 32 KiB L1 icache
- 32 KiB L1 dcache
- 512 KiB L2 cache
- Full IEEE-754 compliant
- Altivec/VMX SIMD support
- 2 hardware threads

IBM Cell/B.E. > PPE

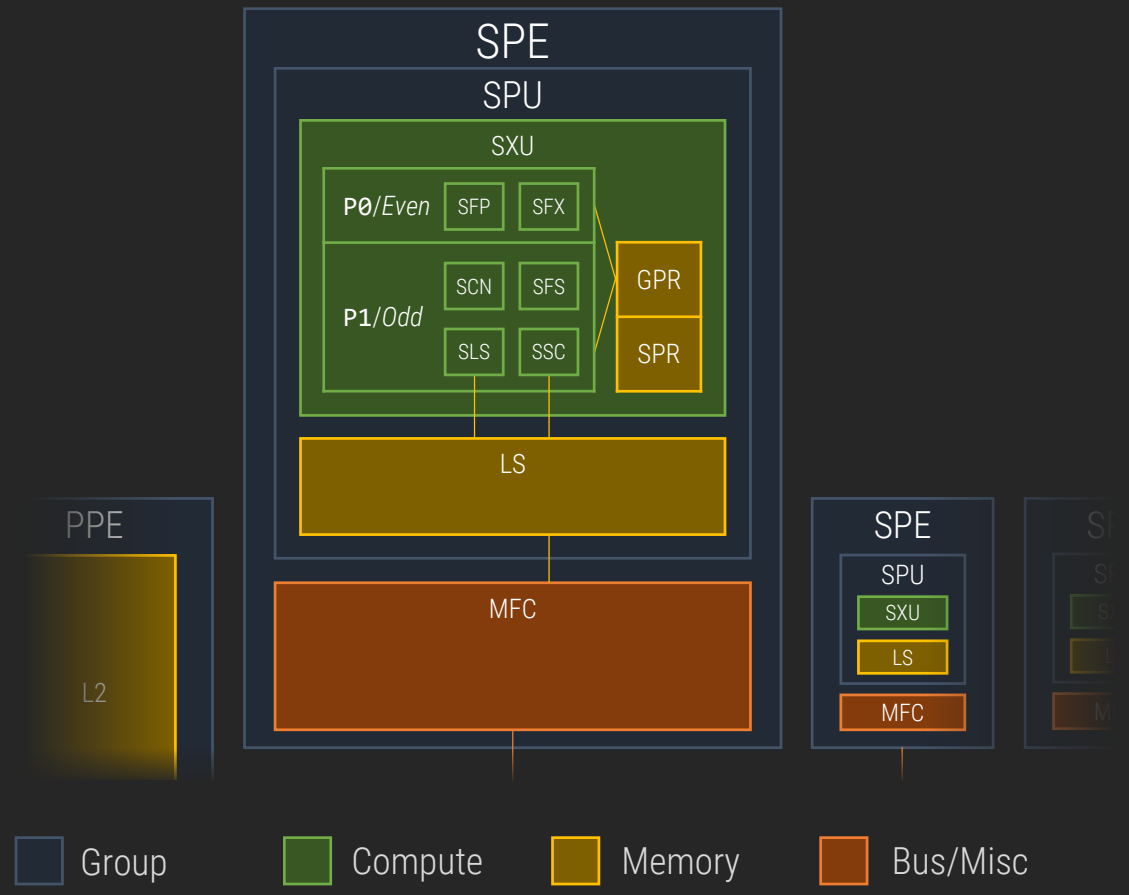
- 64-bit PowerPC (Big-Endian)
  - Register space

GPR (r0-r31)	XER	CR (cr0-cr7)
FPR (f0-f31)	FPSCR	PC, LR, CTR
VR (v0-v31)	VSCR	...

  - Register aliases: `SP=r1`, `TOC=r2`
  - Fixed-size 4-byte instructions
    - Very simple encoding!
- Similar to *Xenon* cores (Xbox 360)
- Slightly modified ABI for a 32-bit *Effective Address* space

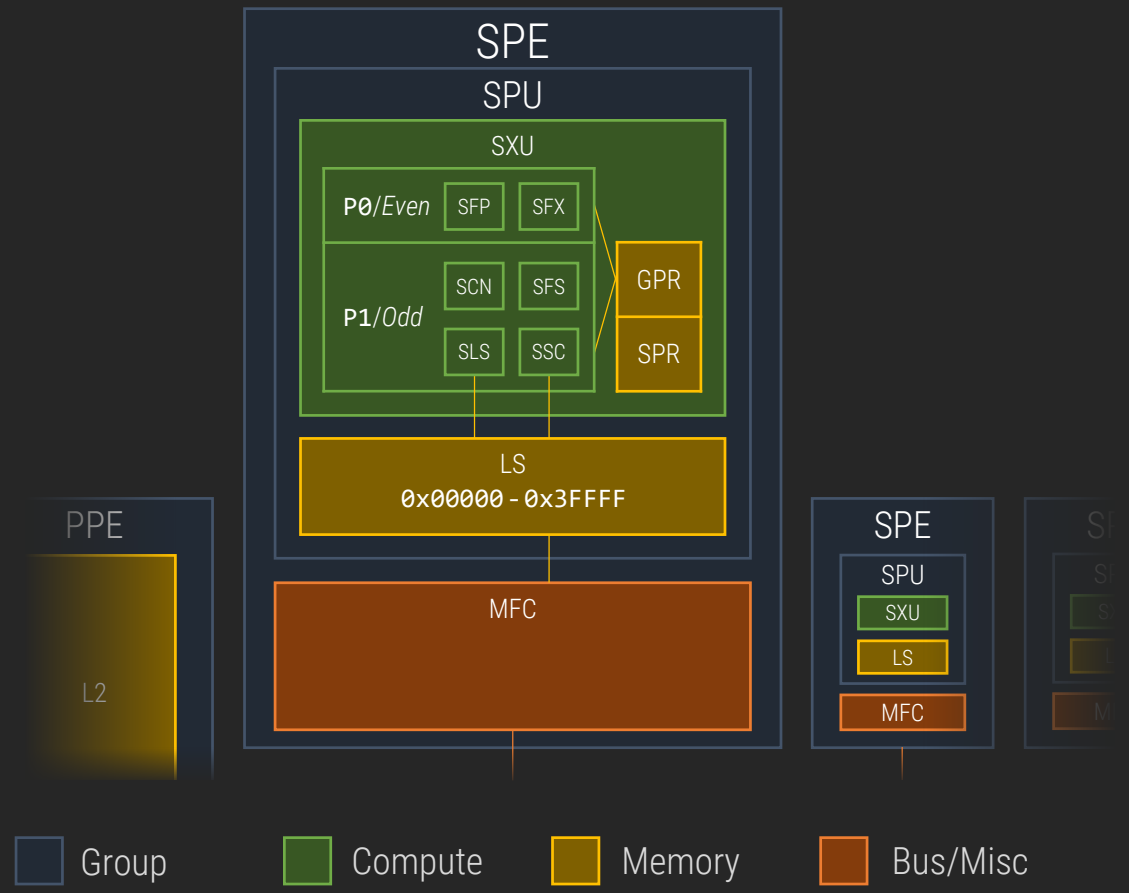
IBM Cell/B.E. > SPE

- RISC/BE custom architecture
  - Encoding similar to PowerPC
- 256 KiB Local Storage (LS)
  - Effectively an addressable L1 cache
  - Customizable via **`LSLR`**
- Register space:
  - Named SPRs: **`PC`**, **`FPSCR`**, **`SRR0`**, …
  - 128 × 128-bit SIMD GPRs (2 KiB!)
  - 128 × 128-bit SPRs (optional)
- 3.2 GHz and 2 pipelines:
  P0/*Even* and P1/*Odd*



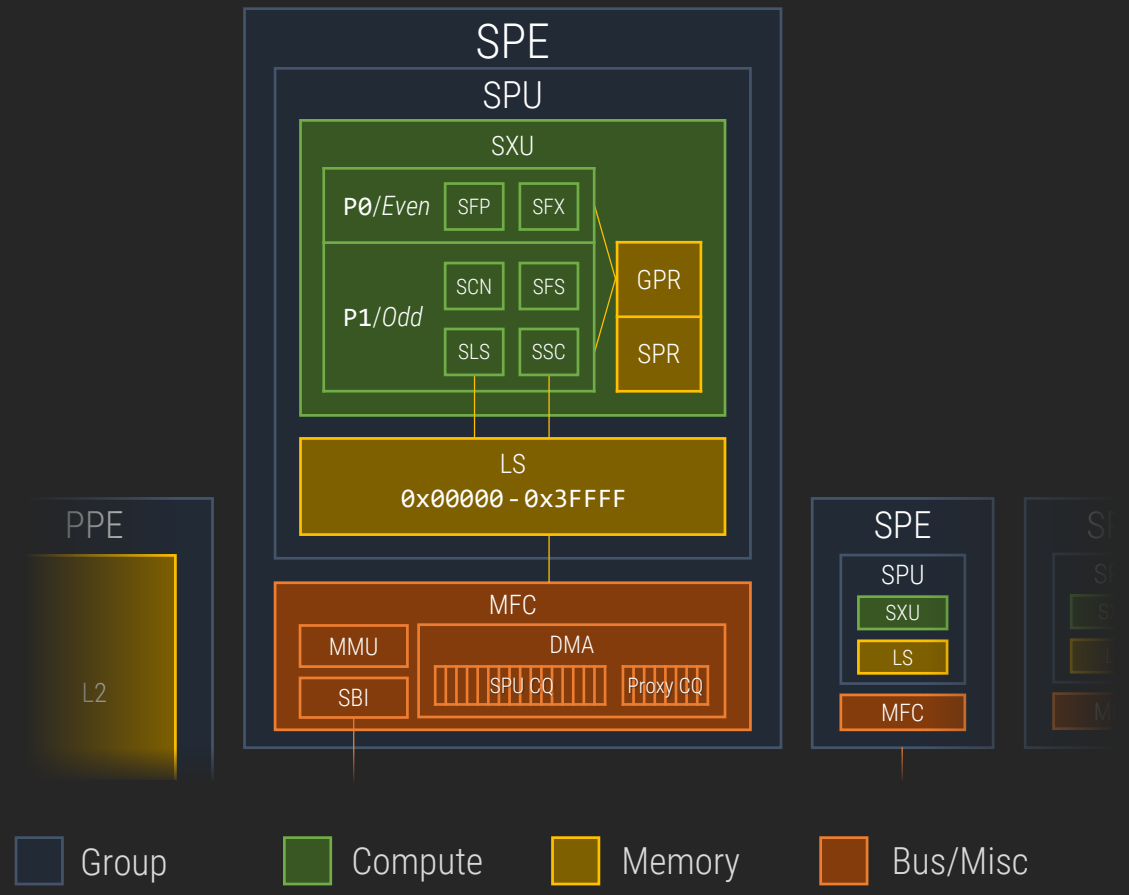
IBM Cell/B.E. > SPE

- **Fixed-Point Unit** (FXU = SFX+SFS)
  - Integer ALU operations on:
    - `S08`, `S16`, `S32` (Signed)
    - `U08`, `U16`, `U32` (Unsigned)
  - Multiplication only with 16-bit operands
- **Floating-Point Unit** (FPU = SFP)
  - **Single**/`F32`: Not IEEE-754 compliant:
    - Only supports round-towards-zero
    - NaN/Inf/Denorms not supported
  - **Double**/`F64`: *Mostly* IEEE-754 compliant
    - Only a subset of operations is supported
  - `FPSCR`: Status/Control for individual GPR slices in SP/DP modes.



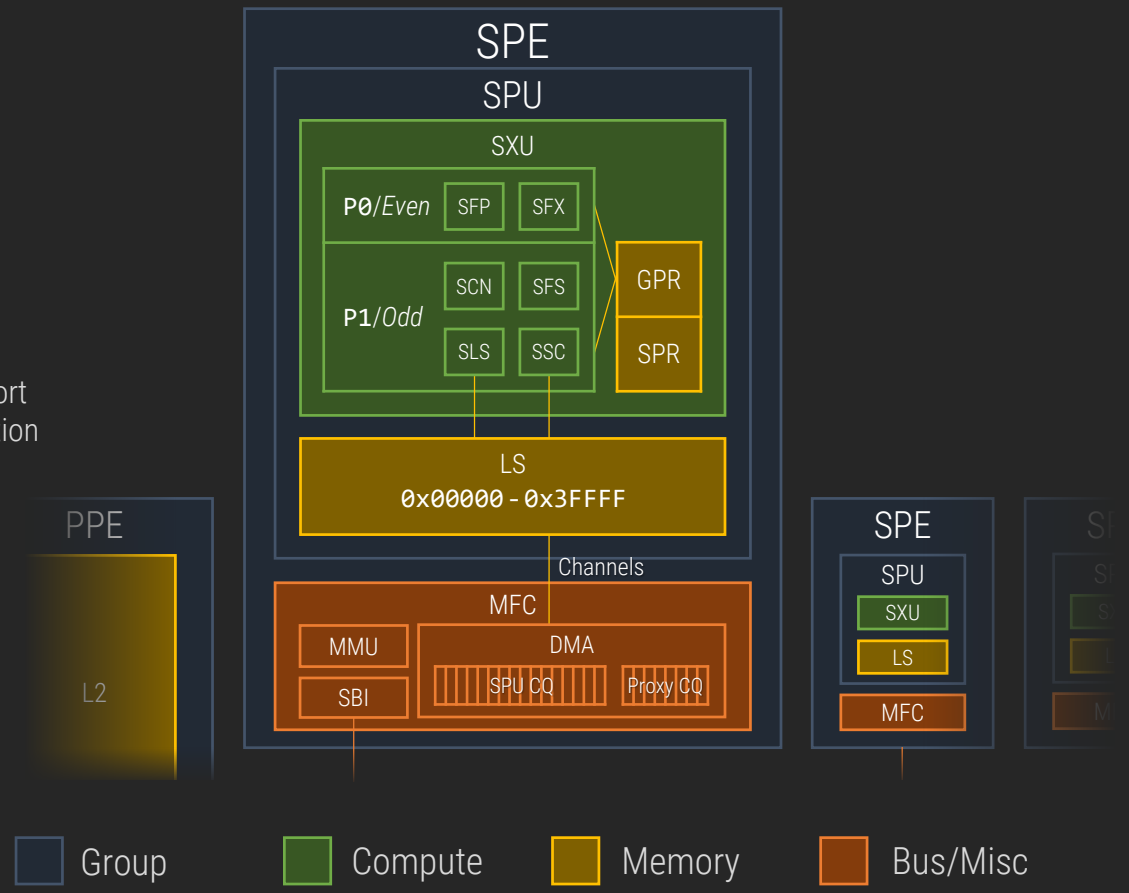
IBM Cell/B.E. > SPE

- **Memory Flow Controller** (MFC)
  for LS-LS and LS-EIB transfers
  - Can alias LS into PPE address space
- DMA Engine
  - SPU/Proxy Command Queues:
    `get*`, `put*`, `snd*`, `barrier`, `eieio`, `sync`, `sdc*`
  - Atomic operations: `{get,put}*ll*`
- Memory Management Unit (MMU)
  - PPE-compatible/controllable via MMIO
  - TLB with fine-grained control
  - System maps PPE EAs 1:1
- Synergistic Bus Interface (SBI)



IBM Cell/B.E. > SPE

- **SPU Mailboxes**
  - Communication/interrupts across SPUs.
- **SPU Channels** $\in \{0, 127\}$ to interface with mailboxes and the MFC:
  - Via instructions
    `rdch` Read channel (into GPR)
    `wrch` Write channel (from GPR)
    } Channels support only one operation
  - Think of as the `in`/`out` of x86 I/O ports
    - Except that some channels can block
  - Channel count via `rdchcnt` indicates the number of entries for that channel
    - Non-blocking channels set `rdchcnt` = 1.
    - When `rdchcnt` = 0 the channel blocks.



IBM Cell/B.E. > SPE

https://youtu.be/ehwFOM4CBKA

https://youtu.be/zKqZKXwop5E

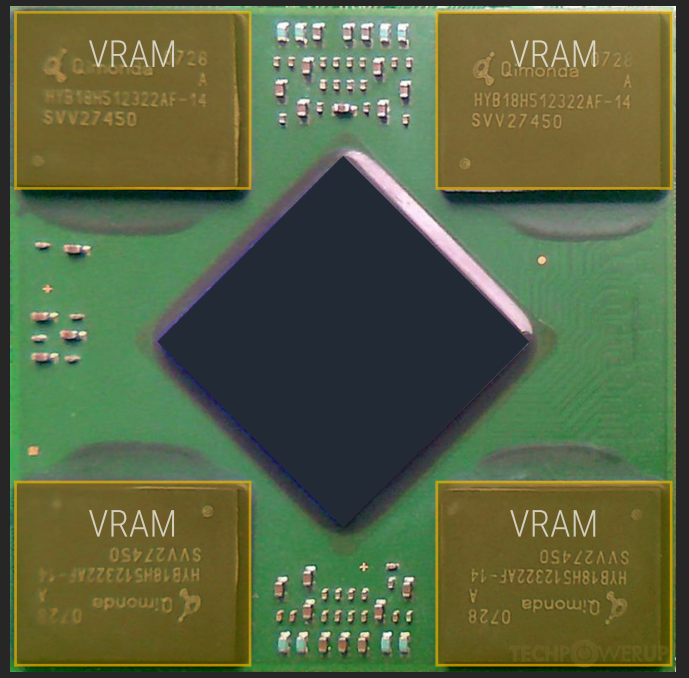
Nvidia RSX

- *Cell/B.E.* appeared insufficient for real-time graphics
  - Scalability problems?
  - Latency problems?
  - Unhappy developers?

  } *Just speculation*
- Sony/Nvidia introduce the RSX GPU
  - Reportedly based on Nvidia 7800GTX.
  - Nvidia groups it as G70/G71 (NV47). Firmware reports it as "*NV4D*".
  - Part of the *Curie* family.
  - "*Direct3D 9*" family.

Nvidia RSX

- Original plans
  - 550 MHz GPU
    - Crippled to 500 MHz for vertex shaders
    - Non-unified shader architecture
  - 256 MB GDDR3 SDRAM at 700 MHz
    - Crippled to 650 MHz
- Theoretical figures and benchmarks provide different figures
- This is of little importance to us…

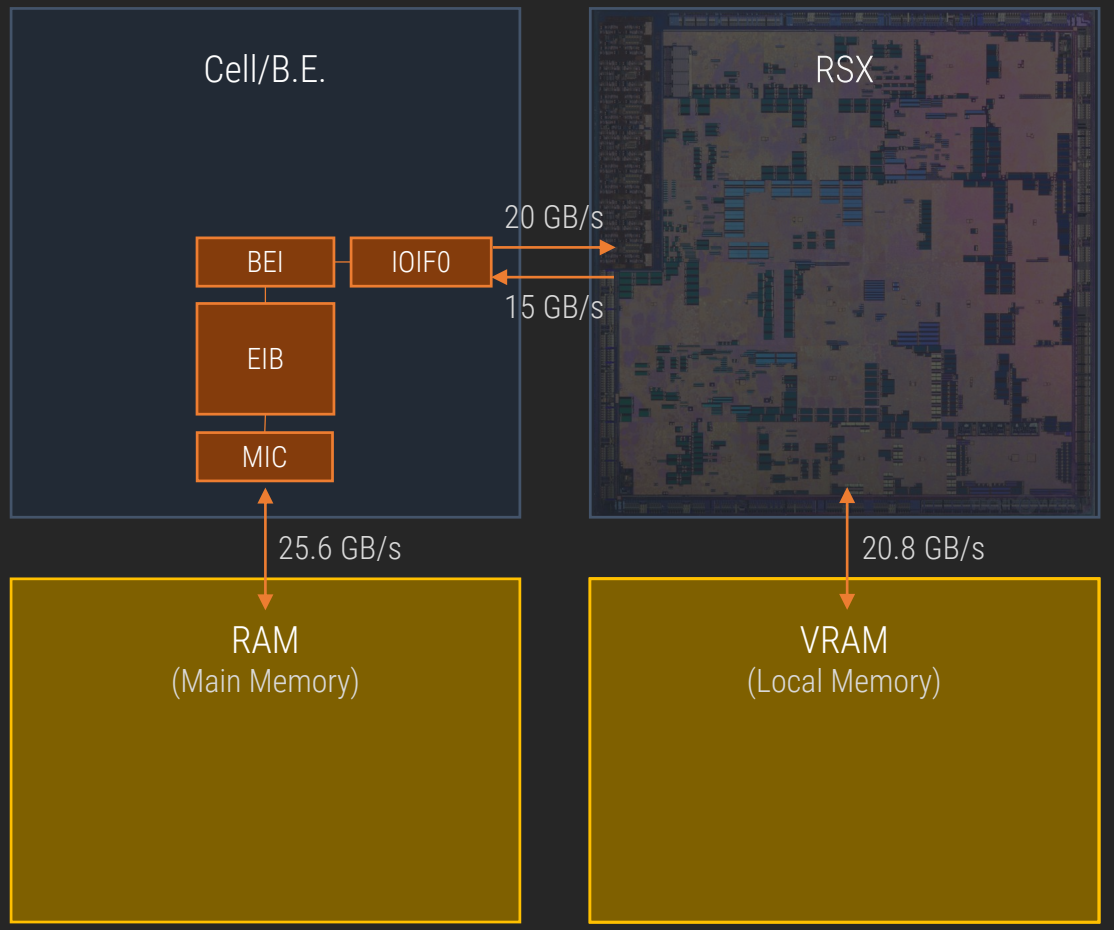


Nvidia RSX

- …But it influences how developers will use the hardware
  - And consequently what emulators will spend most time doing

Processor	256MB XDR	256MB GDDR3
Cell Read	16.8GB/s	16MB/s (15.6MB/s @ 650 MHz)
Cell Write	24.9GB/s	4GB/s
RSX Read	15.5GB/s	22.4GB/s (20.8GB/s @ 650 MHz)
RSX Write	10.6GB/s	22.4GB/s (20.8GB/s @ 650 MHz)

Cell/RSX bandwidth measurements at PSDevWiki

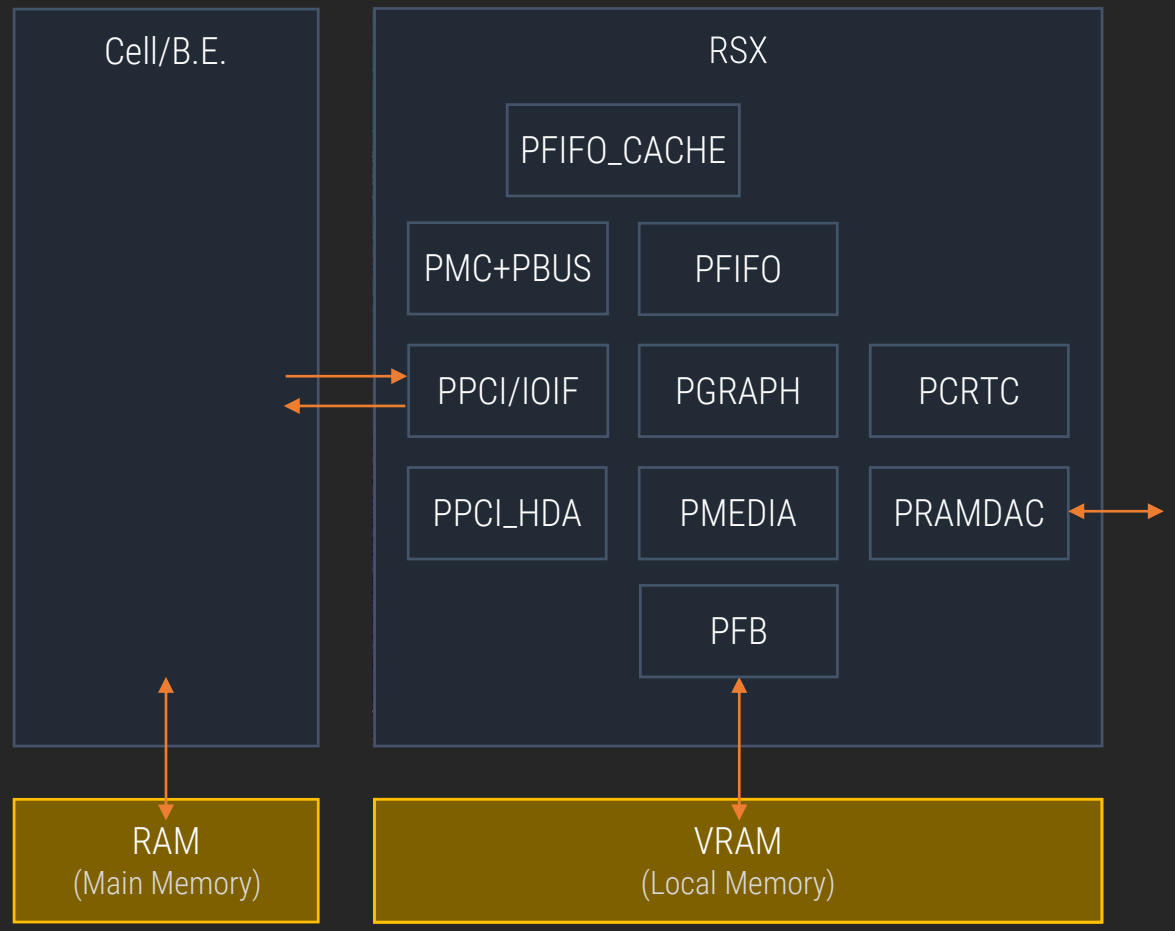


Nvidia RSX

- Made of several engines

		~AMD
PGRAPH	Graphics engine	~GFX/GCA
PFIFO	Command processor	~CP
PCRTC	Display controller	~DCE/DCN
PMEDIA	Video encoder/decoder?	~VCE/UVD
PFB	Memory controller	~GMC
PBUS	Bus interface	~BIF
...	...	...

- Documented at Nouveau/Envytools.
  - Thanks to Marcelina Kościelnicka.



Nvidia RSX

- Made of several engines

		~AMD
PGRAPH	Graphics engine	~GFX/GCA
PFIFO	Command processor	~CP
PCRTC	Display controller	~DCE/DCN
PMEDIA	Video encoder/decoder?	~VCE/UVD
PFB	Memory controller	~GMC
PBUS	Bus interface	~BIF
...	...	...

- Documented at Nouveau/Envytools.
  - Thanks to Marcelina Kościelnicka.

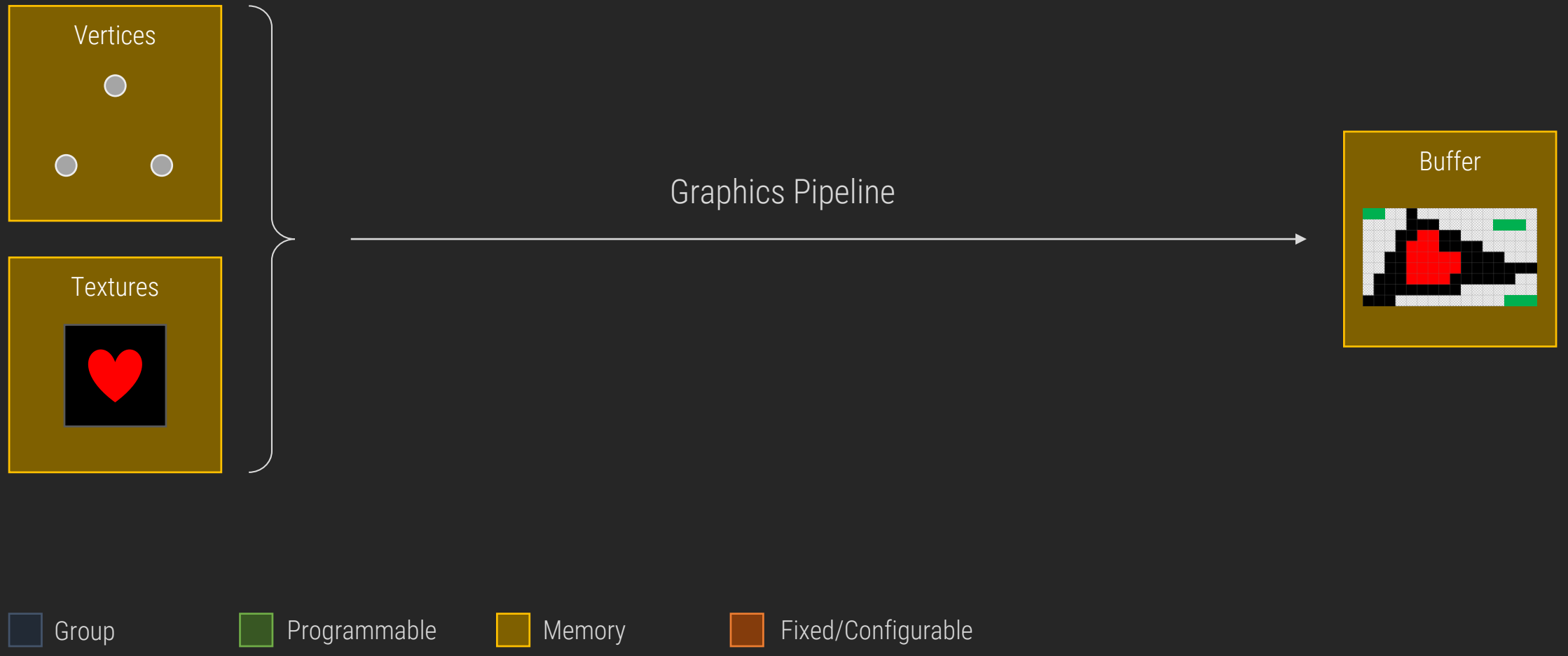
- Configurable over MMIO registers
  - BAR mapped via PCI/IOIF
  - Some engines mapped directly even into userland

```
// LV2 Syscall 675 (0x2A3)
uint64_t sys_rsx_device_map(
  uint32_t* mmio_addr,
  uint32_t* vram_addr,
  uint64_t device_id);
```

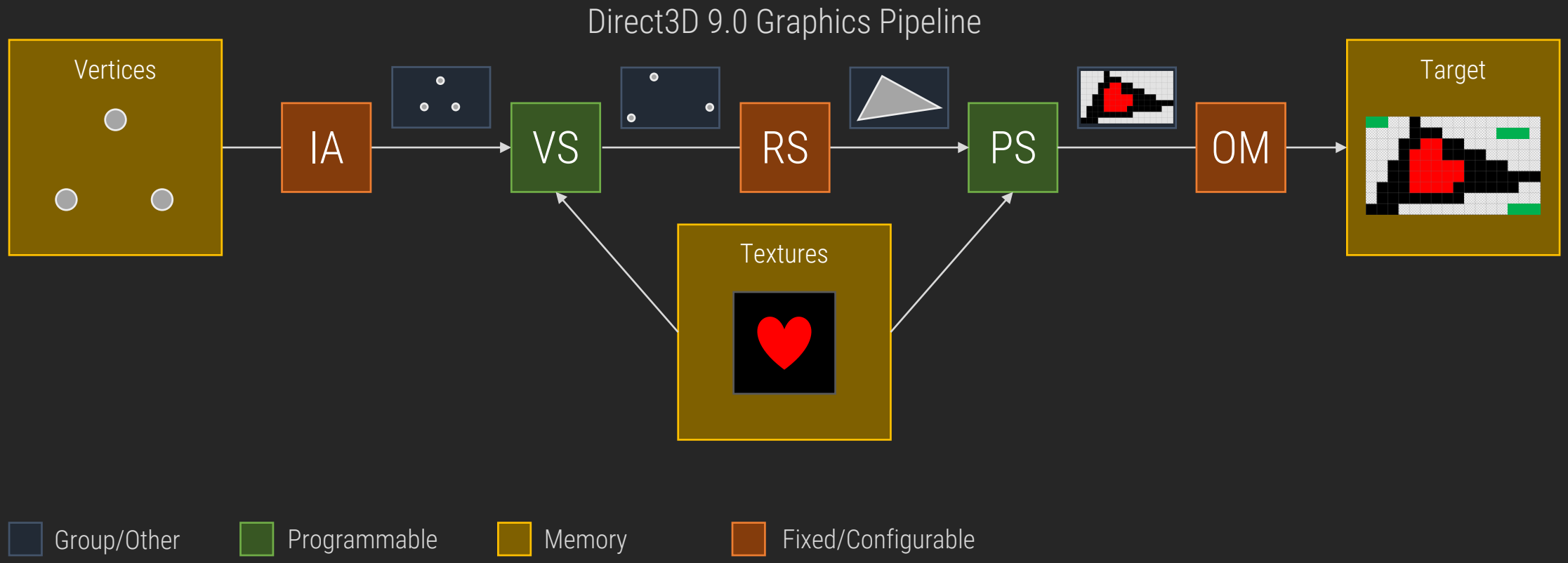
    - Which allows interesting exploit ideas, e.g. "*GPU context escapes*"

- Focus on user-PGRAPH/PFIFO

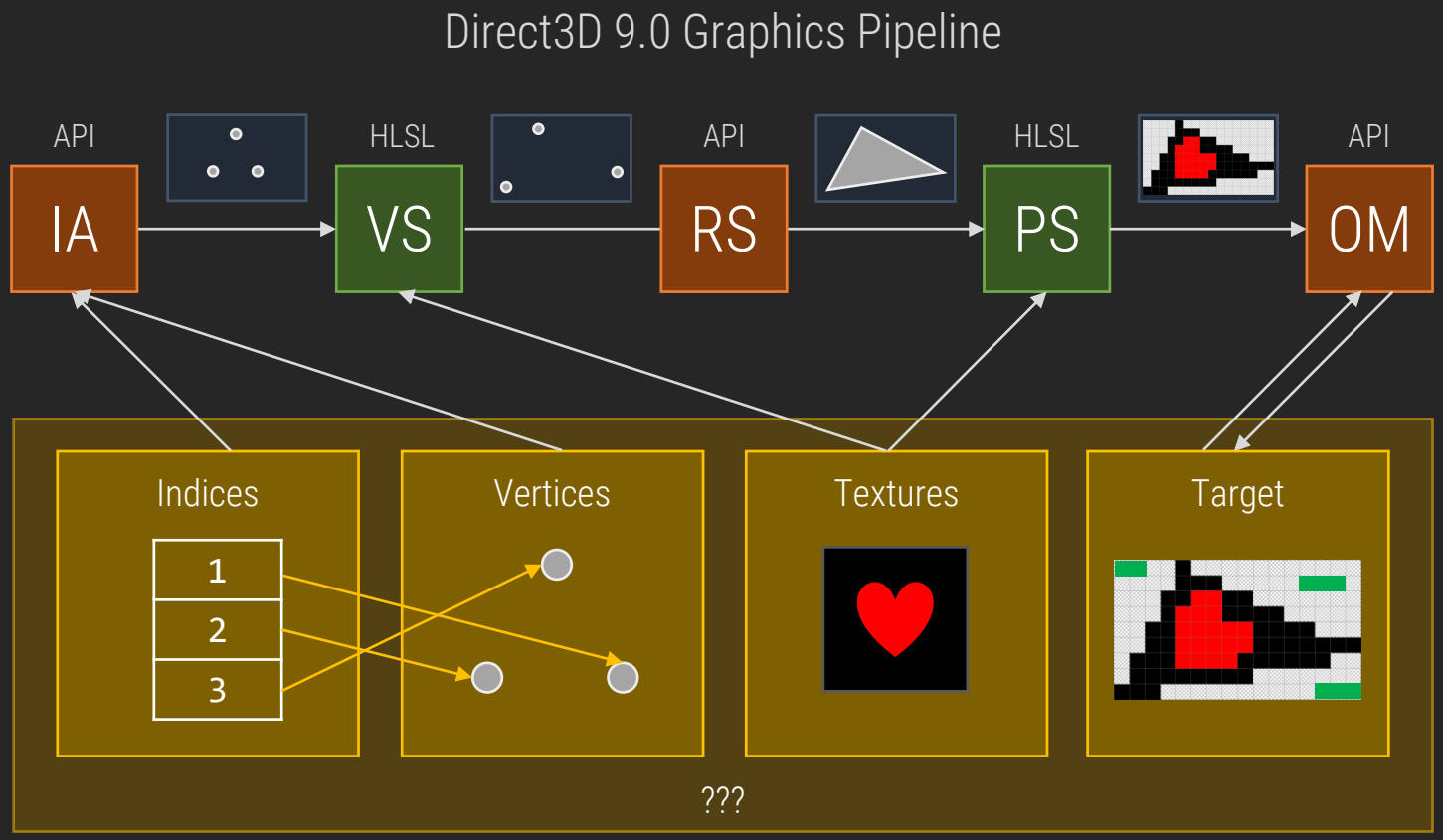
Nvidia RSX > PGRAPH



Nvidia RSX > PGRAPH

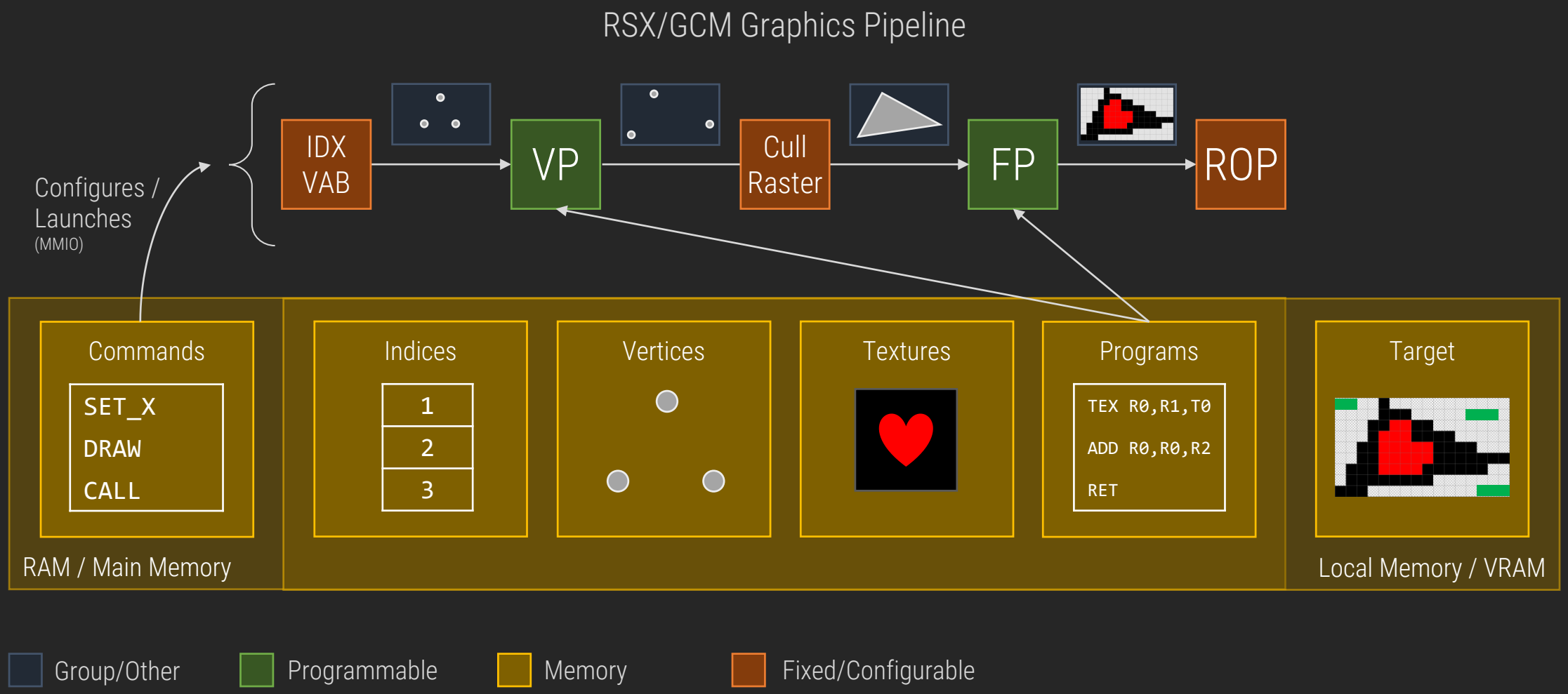


Nvidia RSX > PGRAPH



Group/Other Programmable Memory Fixed/Configurable

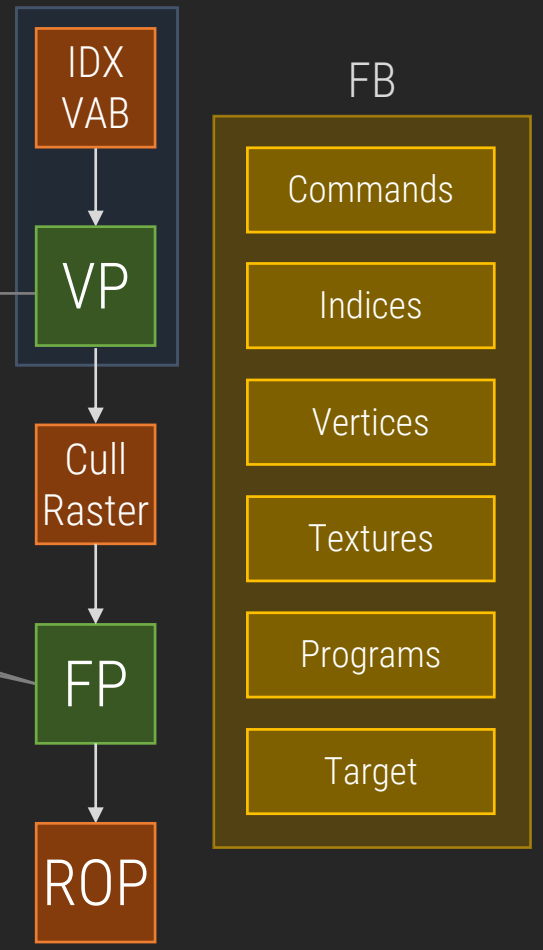
Nvidia RSX > PGRAPH



Nvidia RSX > PGRAPH

- Graphics engine in RSX
- Non-unified shader architecture
  1. Geometry block
     - *Vertex Processing Engine (VPE)*
  2. Raster block
  3. Fragment/Texture block
     - *Shader Computation Top (SCT)*
     - *Shader Computation Bottom (SCB)*
  4. Raster Operation block

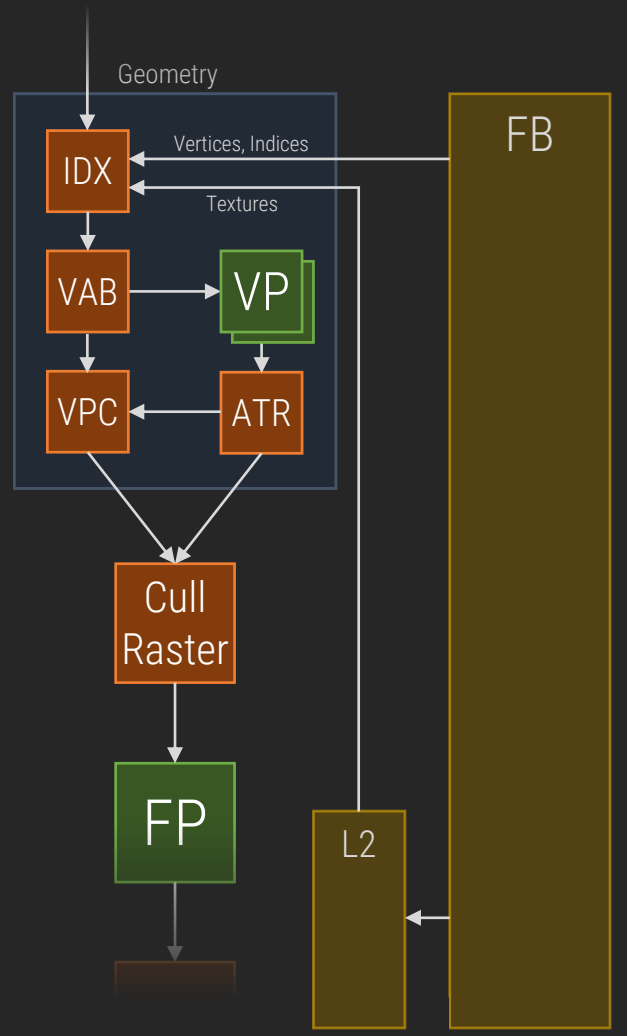
  Discard fragments early
- We will ignore most units.
  - Focus on PGRAPH units which have an *observable effect* in applications.



Nvidia RSX > PGRAPH

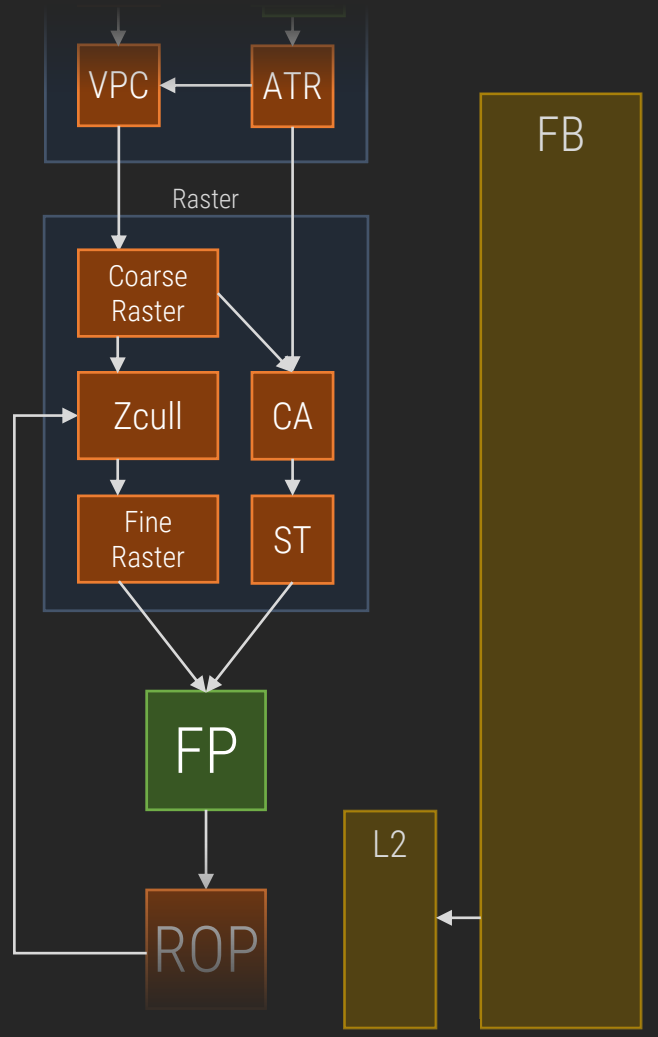
- **Vertex Processing Engine (VPE)**
- Register space
  - Inputs `v0-v15` ~Position, Normal, …
  - Outputs `o0-o15` ~Position, Color, Clips, …
  - Data `r0-r31` ~Locals
  - Constants `c0-c467` ~Globals/Uniforms
  - Samplers `tex0-tex4` ~Texture samplers

  } Hardcoded mappings
  - Stack/calls, Conditions/branching, Boolean constants, …
- Storage for 512 × 128-bit instructions with built-in swizzling/shuffling, masks, conditionals, and ABS/NEG operations
  - Floating-point arithmetic and math `MAX/MIN`, `ADD/SUB/MUL/DIV`, `LOG/LG2/EXP/EX2`, `SIN/COS`
  - Comparisons: `SLE/SLT/SNE/SGT/SGE/SEQ`
  - Texture fetch: `TEX`, Stack access: `PSH/POP`, `NOP`, `MOV`



Nvidia RSX > PGRAPH

- **Rasterizer**
  - Turns vectors into fragments
  - Vector attribute interpolation for each fragment
  - *Zcull*: Checks depth/stencil buffers (update by ROPs) to discard fragments that will not be visible.
- Supports OpenGL 1.5 primitives
  - `POINTS`, `LINES`, `LINE_STRIP`, `LINE_LOOP`, `TRIANGLES`, `TRIANGLE_FAN`, `TRIANGLE_STRIP`, `QUADS`, `QUAD_STRIP`, `POLYGON`
- 2D Raster block available, but 3D-2D context switch is expensive.

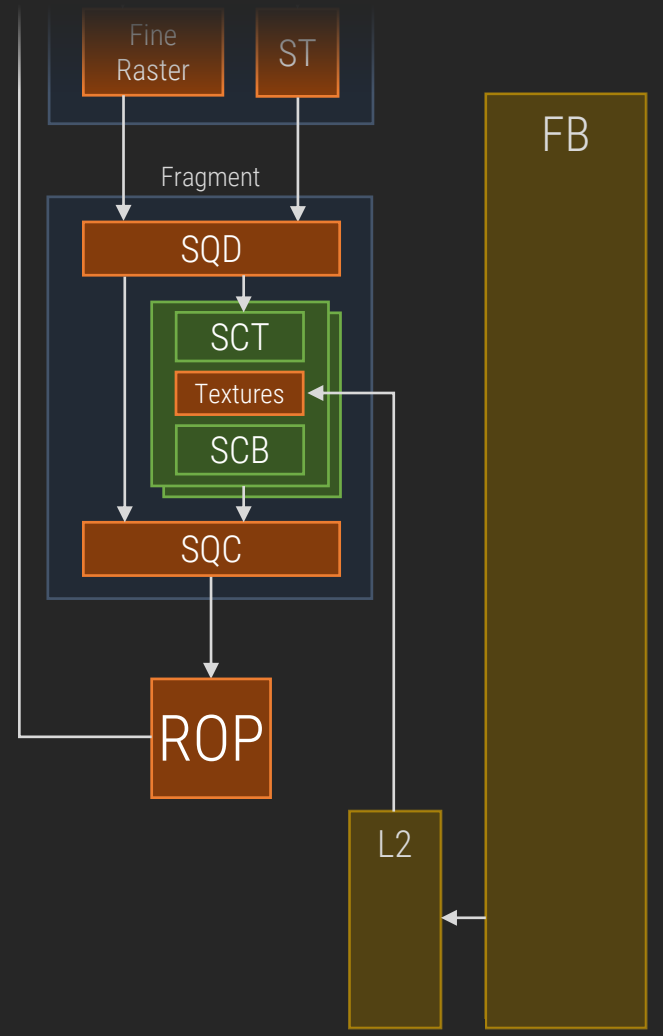


Nvidia RSX > PGRAPH

- **Fragment Shader (SCT+SCB)**
- Register space
  - Inputs `f0-f14` ~Position, Color, …
  - Output `r_i,h_2i` (i∈0…4) ~4×Color, Depth

  } Hardcoded mappings

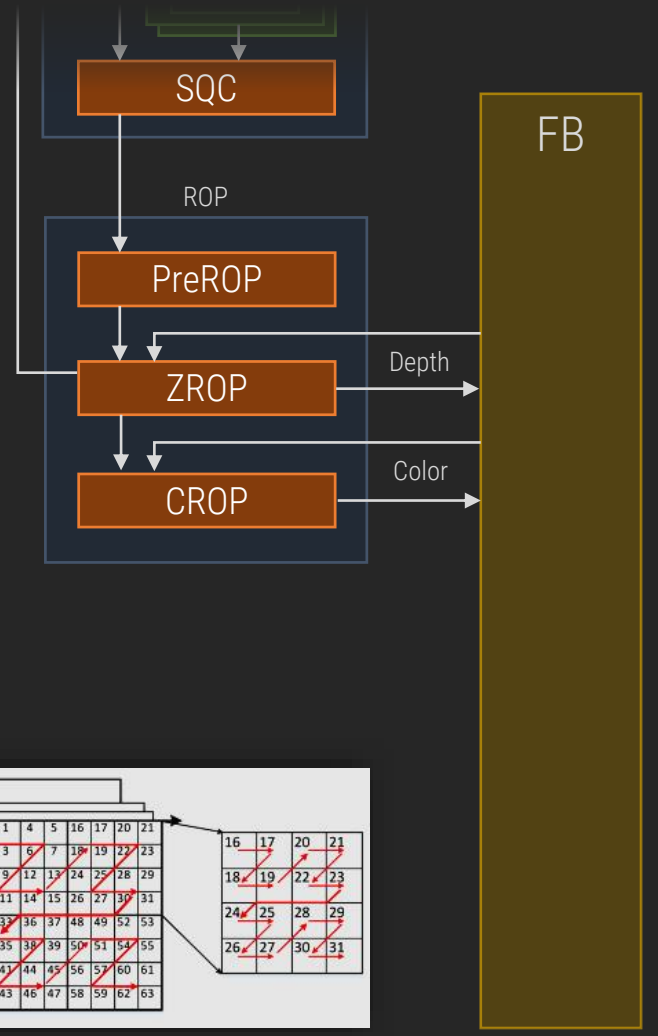
  - Data (Overlapped) `r0-r47` (FP32) / `h0-h47` (FP16) ~Locals
  - Samplers `tex0-tex15` ~Texture samplers
  - Constants are inlined after instructions!
- Mixed-endian 128-bit instructions in VRAM with same VPE suffixes plus scaling, saturation. Same opcodes except:
  - Added screen space derivatives: **DDX**/**DDY**
  - Added pack: **PK{2,4,16,B,G}** and unpack: **UP{2,4,16,B,G}**
  - Added texture fetches with bias/derivatives/LOD: **TEX**, **TXP**, **TXB**, **TXD**, **TXL**
  - No stack accesses, though subroutines are supported
  - SCB/SCT synchronization via **FENCT**/**FENCB**



Nvidia RSX > PGRAPH

- **Raster Operation (ROP)**
- Updates the Z/Depth-buffers and Zcull RAM to discard subsequent occluded fragments.
- Finally, it updates the color buffers (≤4 MRTs), if allowed by depth tests.

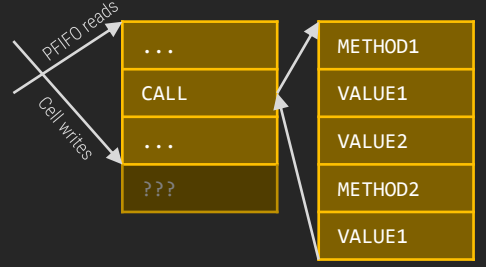
- Most texture attributes map to standard ones, e.g. texel formats, wrap modes, etc.
  - Except for cache-optimized swizzled textures →



Nvidia RSX > PFIFO

- Command submission engine
  - `PFIFO_CACHE` MMIO registers mapped by LV2/GCM into userland

```
+0x40 DMA_PUT
+0x44 DMA_GET
+0x48 REF_CNT
```

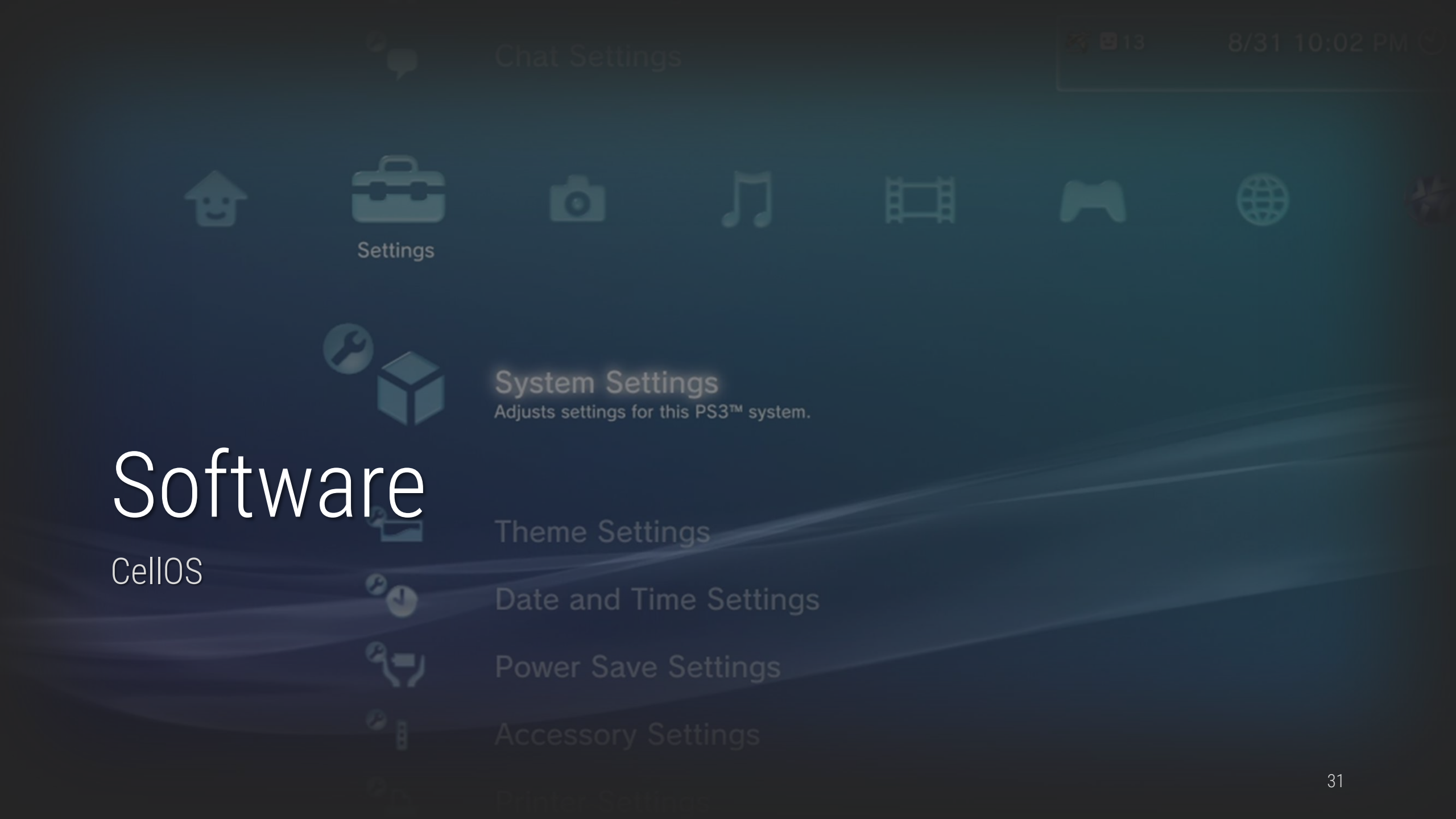


- **PFIFO commands**
  - Branching via JUMP/CALL/RET
  - Methods bound to engines/driver via 16-bit offsets and 32-bit values
  - Similar to AMD PM4.

- **PFIFO methods**
  - `NV406E` *NV40_CHANNEL_DMA* Synchronization/semaphores
  - `NV4097` *NV40_CURIE_PRIMITIVE* PGRAPH context configuration/draws (3D)
  - `NV3089` NV30_SCALED_IMAGE_FROM_MEMORY Image blitting (2D)

  ...Many others

- Method offsets software-defined but essentially constant in PS3
  - Emulators turn this into a big static buffer or switch-case statement

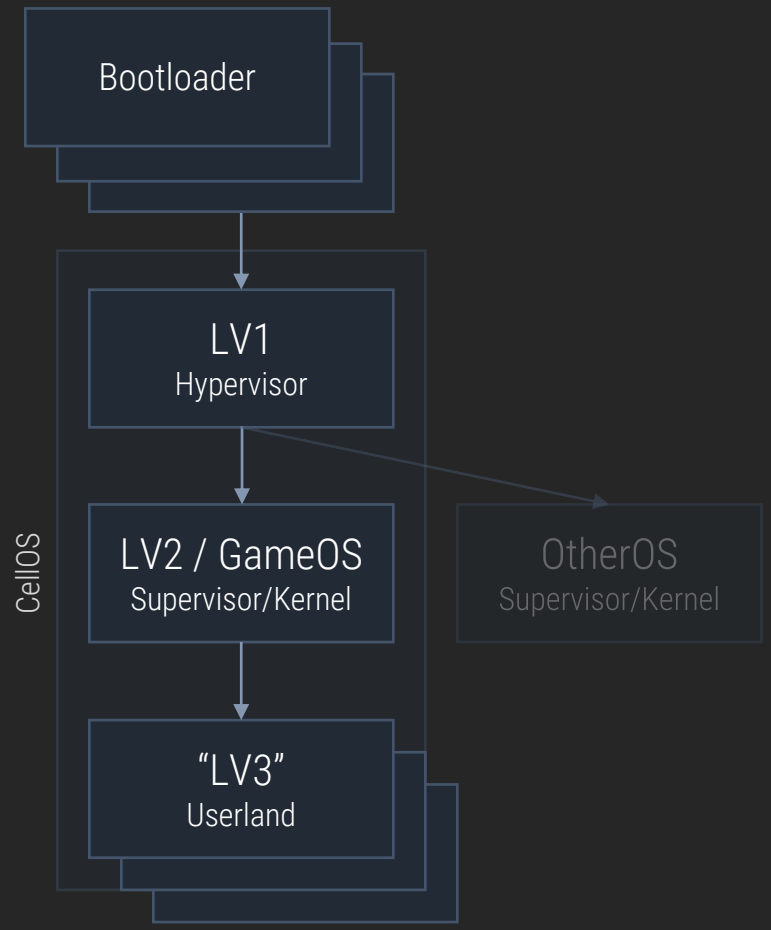


Software

CellOS

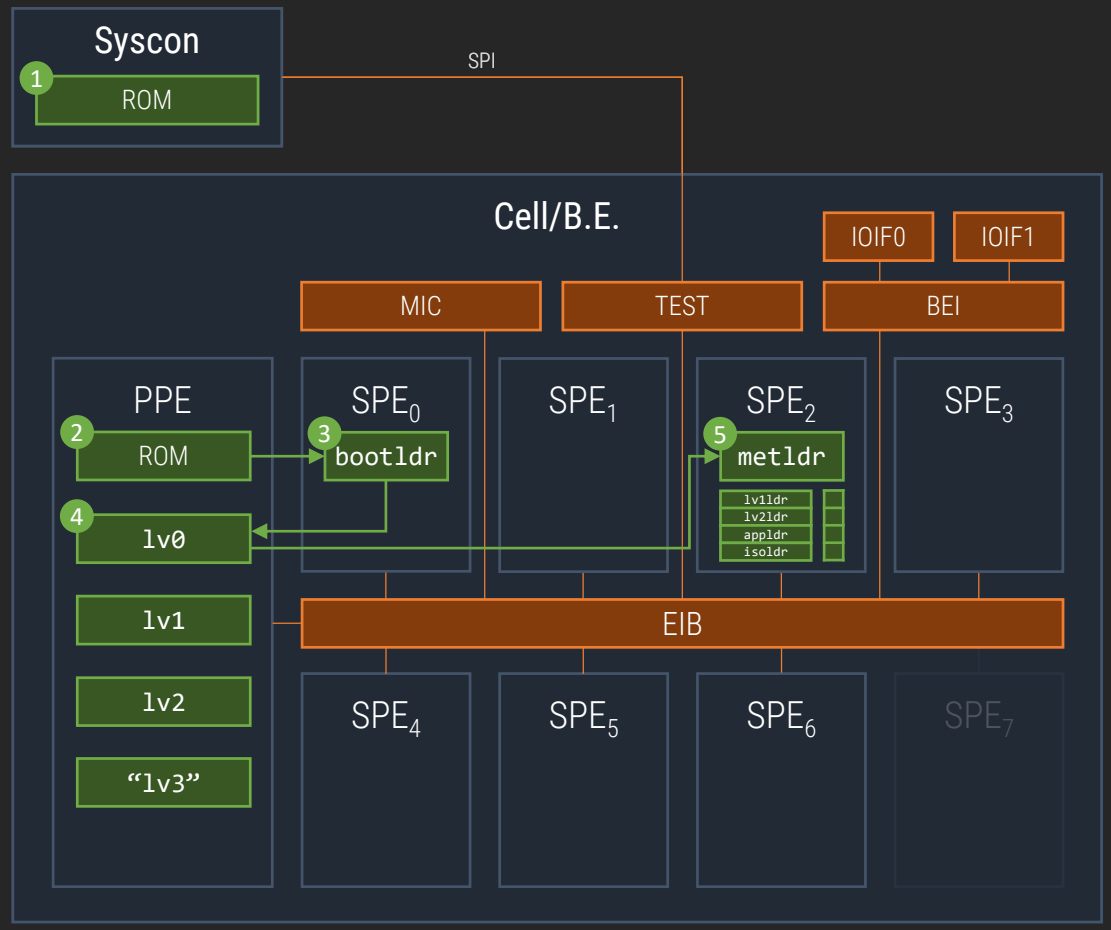
Overview

- Complex multi-stage boot process involving different devices
- Check Fail0verflow's 27C3 *Console Hacking (2010)* talk! https://media.ccc.de/v/27c3-4087-en-console_hacking_2010
- Our focus is:
  - Mainly CellOS/GameOS
  - Understanding *what-runs-where* and *who-knows-who*



Bootloader ≤3.55

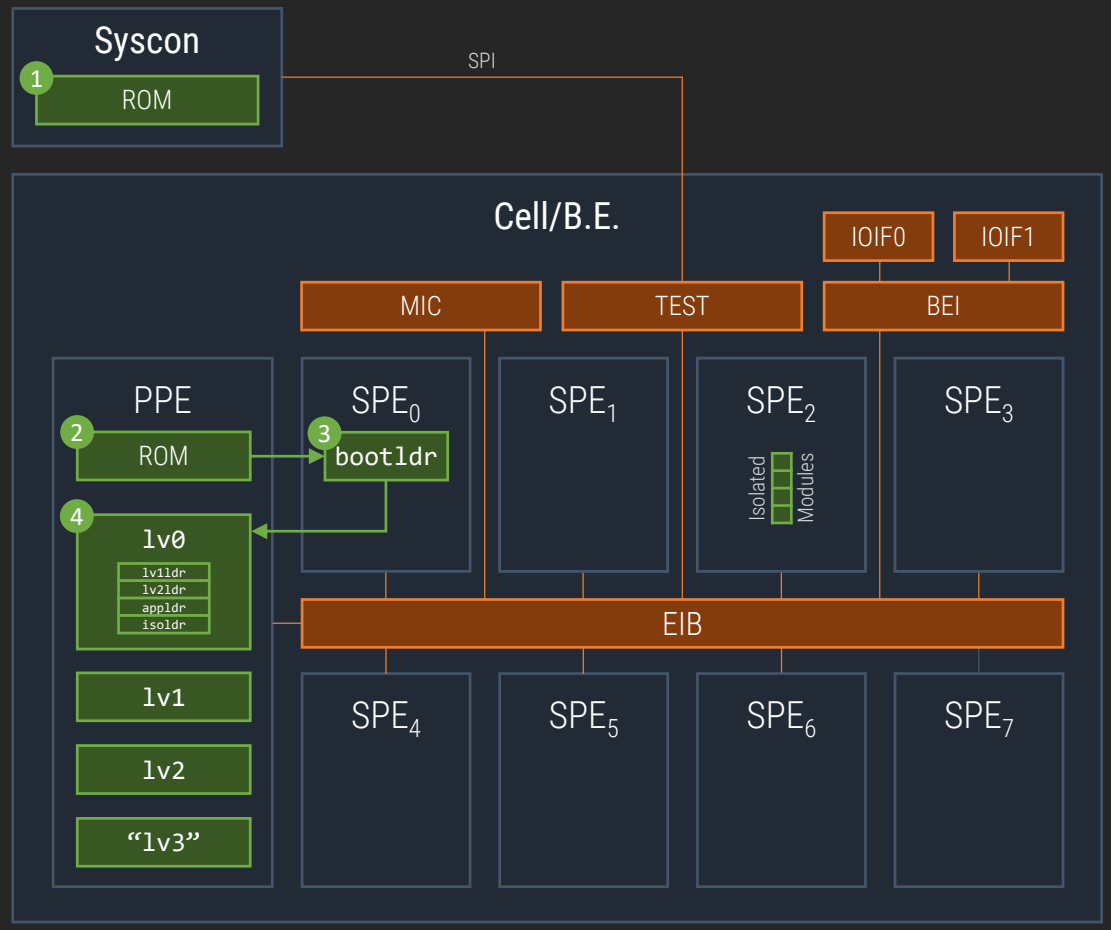
1. *Syscon* powers on, runs its internal ROM:
   - Initializes Cell/B.E by submitting the *Configuration Ring* via SPI/TEST
   - Deactivates 8th SPU
2. PPU boots from Cell's hidden ROM:
   - Loads **`bootldr`** from Flash into SPU_0
3. SPU_0 /**`bootldr`** initializes memory and I/O, and loads **`lv0`** into PPU
4. PPU/**`lv0`** loads **`metldr`** into SPU_2
5. SPU_2 /**`metldr`** loads:
   1. **`lv1ldr`** Loads CellOS Lv-1 (hypervisor)
   2. **`lv2ldr`** Loads CellOS Lv-2 (kernel)
   3. **`appldr`** Loads VSH/apps/games
   4. **`isoldr`** Loads isolated SPU modules



Bootloader ≥3.60

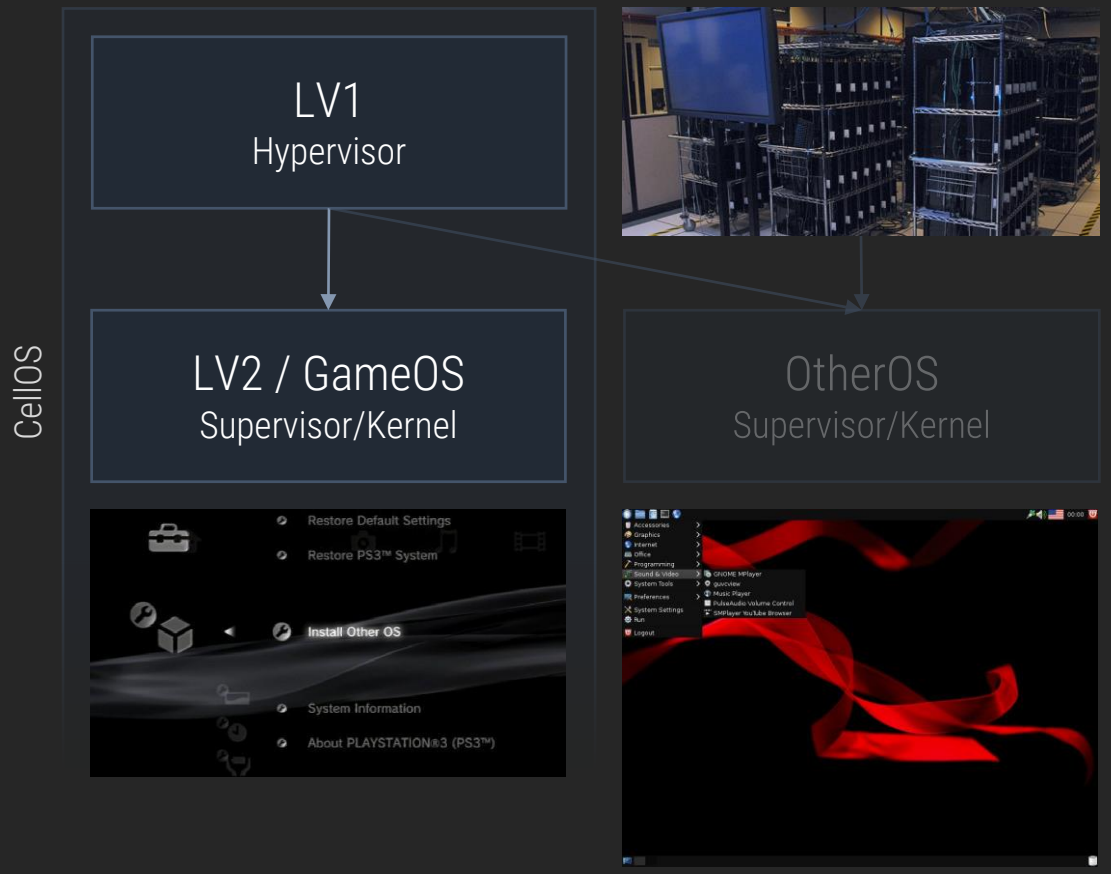
1. *Syscon* powers on, runs its internal ROM:
   - Initializes Cell/B.E by submitting the *Configuration Ring* via SPI/TEST
   - Deactivates 8th SPU
2. PPU boots from Cell's hidden ROM:
   - Loads **`bootldr`** from Flash into SPU_0
3. SPU_0 /**`bootldr`** initializes memory and I/O, and loads **`lv0`** into PPU
4. PPU/**`lv0`** loads

    1. **`lv1ldr`** Loads CellOS Lv-1 (hypervisor)
    2. **`lv2ldr`** Loads CellOS Lv-2 (kernel)
    3. **`appldr`** Loads VSH/apps/games
    4. **`isoldr`** Loads isolated SPU modules



CellOS Lv-1 (Hypervisor)

- Supercomputing/marketing ambitions resulted in *OtherOS*:
  - Yellow Dog Linux
  - Red Ribbon Linux
- Isolation/security requirements
  - Unrestricted hardware access
    - Could endanger store monopoly
    - Could enable software piracy
- Sony introduces *CellOS Lv-1* to allocate hardware resources

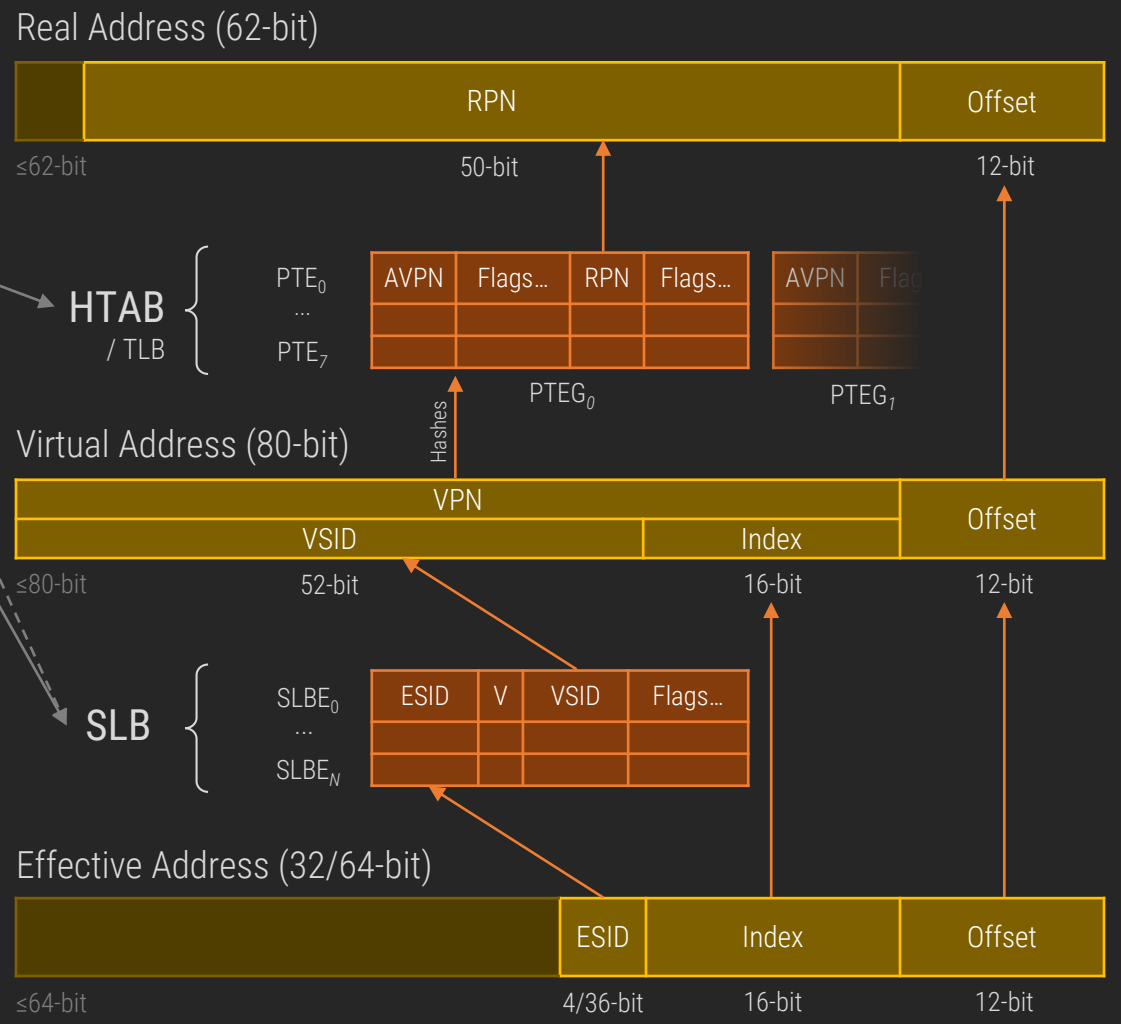


CellOS Lv-1 (Hypervisor)

- Privilege levels (PowerPC)

Privilege 1	`HV=1, PR=0`	CellOS Lv-1
Privilege 2	`HV=0, PR=0`	CellOS Lv-2
Userland	`HV=0, PR=1`	CellOS "Lv-3"

- 2-stage address translation
  - Linear search on SLB
  - Hash-table lookup on HTAB
    - Allocated/managed by LV1



CellOS Lv-1 (Hypervisor)

- ~150 hypercalls (`r11` ∈ [0,255])

`lv1*`	Memory, interrupt mapping/allocation
`lv1_*pci*`	PCI access
`lv1_*spe*`	SPE allocation/management
`lv1_*lpm*`	Logical Performance Monitor access
`lv1_*uart*`	Virtual UART (PS3 AV) access
`lv1_*device*`	Open devices and manage DMA/MMIO regions
`lv1_storage*`	Hard Disk/Flash access
`lv1_net*`	NIC driver interface
`lv1_gpu*`	Nvidia RSX driver interface (limited in *OtherOS*)
`lv1*`	Misc: Version, Pause/Panic, Tracing,

- Standard PowerPC ABI Inputs/outputs on `r3-r10`
- Further resources
  - Defined/used in Linux (upstream) `arch/powerpc/include/asm/lv1call.h`
  - RSX/IOIF drivers reversed at `{3141card,AlexAltea}/lv1-reversing`
  - PSDevWiki's HV Syscall Reference `psdevwiki.com/ps3/HV_Syscall_Reference`

CellOS Lv-2 (Kernel)

- Custom monolithic kernel with cherry-picked code from FreeBSD/NetBSD
- **Kernel object**-based interfaces
  1. `int sys_*_create(int* id, …);`
  2. `int sys_*_methods(int id, …);`
  3. `int sys_*_destroy(int id, …);`
- Kernel objects have
  - 8-bit type identifiers
    `TIMER`, `MUTEX`, `FD`, `CRYPTO`, …
  - 32-bit instance identifiers
- Error codes based on *errno.h*

```
CELL_OK      = 0             ...
CELL_EAGAIN  = 0x80010001    CELL_ENOTSDATA  = 0x8001003B
CELL_ENOSYS  = 0x80010002    CELL_ESDKVER    = 0x8001003C
CELL_ENOMEM  = 0x80010003    CELL_ENOLICDISC = 0x8001003D
CELL_ESRCH   = 0x80010004    CELL_ENOLICENT  = 0x8001003E
```

So far so good… — wat

- Many device drivers have dedicated syscalls, not IOCTLs
- Some are limited to root/devkits

CellOS Lv-2 (Kernel)

- ~750 syscalls (`r11` ∈ [0,1023])

`sys_process*`	Process management
`sys_ppu_thread*`	PPU Thread management
`sys_spu_thread*`	SPU Thread management
`sys_raw_spu*`	RawSPU Thread management
`sys_isolated_spu*`	IsoSPU Thread management
`sys_rsx`	RSX video interface
`sys_rsxaudio`	RSX audio interface
`sys_bt`	Bluetooth interface
`sys_usb{d,btaudio}`	USB interface
`sys_hid`	Human interface devices
`sys_net*`	Sockets interface
`sys_fs*`	Filesystem interface (POSIX+extra)
`sys_ss*`	Process socket service
`sys_storage*`	Hard Disk/Flash access
`sys_event*`	Event flag and event queue/ports
`sys_*mutex*`	Mutex and lightweight mutex creation
`sys_{memory,vm}*`	Virtual memory management
`sys_prx*`	Dynamic library loading
`sys_{dbg,deci3}*`	Debugging and system introspection
`sys_crypto*`	Crypto engine (PPU)

CellOS “Lv-3” (Userland)

- PowerPC *Problem State* processes
  - 32-bit effective address space, but mostly standard PPC ABI
  - May allocate SPU and RSX contexts
- System boots **Virtual Shell (VSH)**
  - Launches *XrossMediaBar* (XMB)
  - System configuration, apps/games installation (PKGs) and loading:
    - Identifiers and basic file structure:

```
ABCD12345/ICON0.PNG
ABCD12345/PARAM.SFO
ABCD12345/USRDIR/{EBOOT.BIN, …}
```

    - Signature enforcement; no root/RWX
  - Own RSX context for game overlays

- **SELF/SPRX modules**
  ELF wrapper/container with
  - Custom header with metadata for versioning, encryption, verification
  - Tracking imports/exports through 32-bit function identifiers
  - Custom ELF flags/types

```
enum {
    /* Program types */
    // Cell OS Lv-2 (OS) specific
    PT_PROC_PARAM   = 0x60000001,
    PT_PRX_PARAM    = 0x60000002,
    // Cell B.E. (CPU) specific
    PT_SCE_PPURELA  = 0x700000A4,
    PT_SCE_SEGSYM   = 0x700000A8.
    // ...
};
```

```
enum {
    /* Program flags */
    PF_SPU_X = 0x00100000,
    PF_SPU_W = 0x00200000,
    PF_SPU_R = 0x00400000,
    PF_RSX_X = 0x01000000,
    PF_RSX_W = 0x02000000,
    PF_RSX_R = 0x04000000,
    // ...
};
```

CellOS “Lv-3” (Userland)

- **Virtual filesystem**
  - `/dev_hdd0` Hard drive: Persistent files (apps, saves) UFS
  - `/dev_hdd1` Hard drive: Volatile files (cache) FAT32
  - `/dev_flash*` Flash storage FAT16/12
  - `/dev_usb*` USB mass-storage devices FAT32/FAT
  - `/dev_{ms,cs,sd}` MS/CS/SD card devices FAT32/FAT
  - `/dev_bdvd` Blu-ray device UDF/DISCFS
  - `/host_root` Remote development host computer *DUMMYFS*/HOSTFS
  - `/app_home` Application directory *DUMMYFS*
  - ...

CellOS “Lv-3” (Userland)

Dynamic Libraries (.sprx)

- System libraries files stored in `/dev_flash/sys/external` `/dev_flash/sys/internal`
  - Interfaces and functionality
    - LV2 kernel and system utilities
    - Image/audio/video codecs (PPU/SPU!)
    - Networking utilities
  - ~150 modules and ~4000 functions!
- Other formats: SDATA, MSELF
  - Used by game developers

Static Libraries (.a/.h)

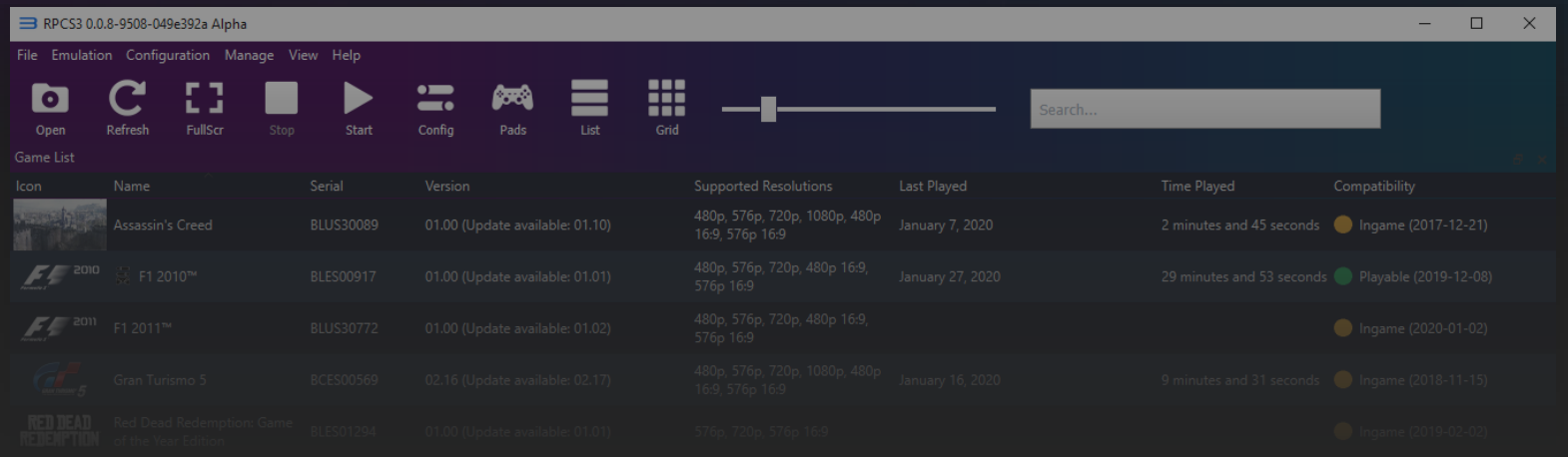
- Provided by official PS3 SDK, linked into the final application
  - Although {in,NDA-}accessible we can find them via symbols/diffs
    - GCM command submission
    - C standard library
  - Can be high-level emulated, if you try hard-enough

Open-source
PlayStation 3 Emulator

RPCS3 is an experimental open-source Sony PlayStation 3 emulator and debugger written in C++ for Windows and Linux. RPCS3 began development in May of 2011 by its founders, DH and Hykem.

Emulators

RPCS3



Approach

Low-level → High-level

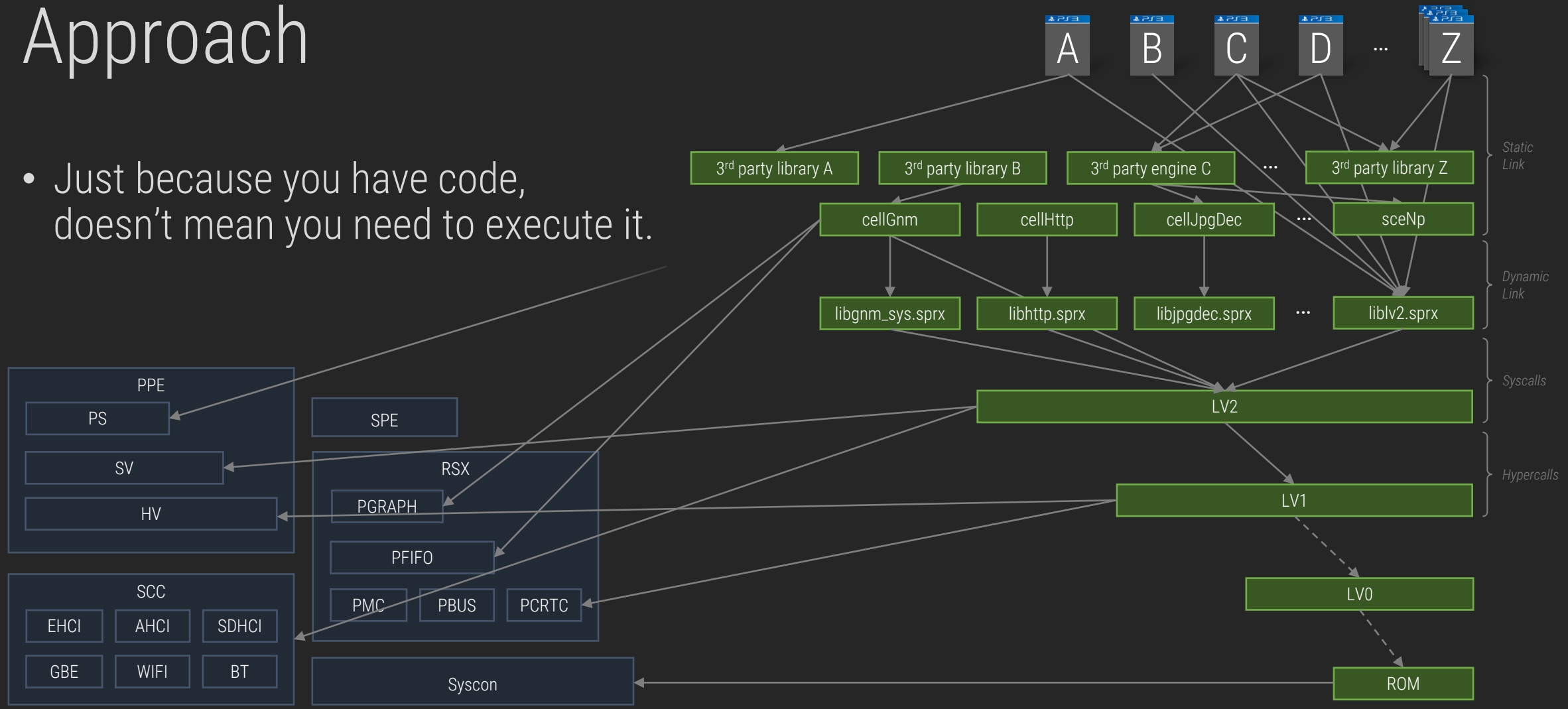
- *Full-system* or *user-mode* emulation?
  1. **What do you want to preserve?**
     e.g. Applications and games?
  2. Is it / Are they…
     - Easy to obtain?
     - Fast to execute?
     - Riddled with versions/variants?
     - Does not isolate underlying dependencies?
       - **Run it** → For each dependency…
     - Easy to understand?
     - Easy to implement?
     - Exposing a stable interface?
       - **Implement it**

- All PlayStation 3 emulators do user-mode emulation.

  This resulted in the misnomers:
  - "*LLE*": Executing system libraries
  - "*HLE*": Hooking system libraries
- With the exception of IBM's own *Full-System Simulator*.

Approach

- Just because you have code, doesn't mean you need to execute it.

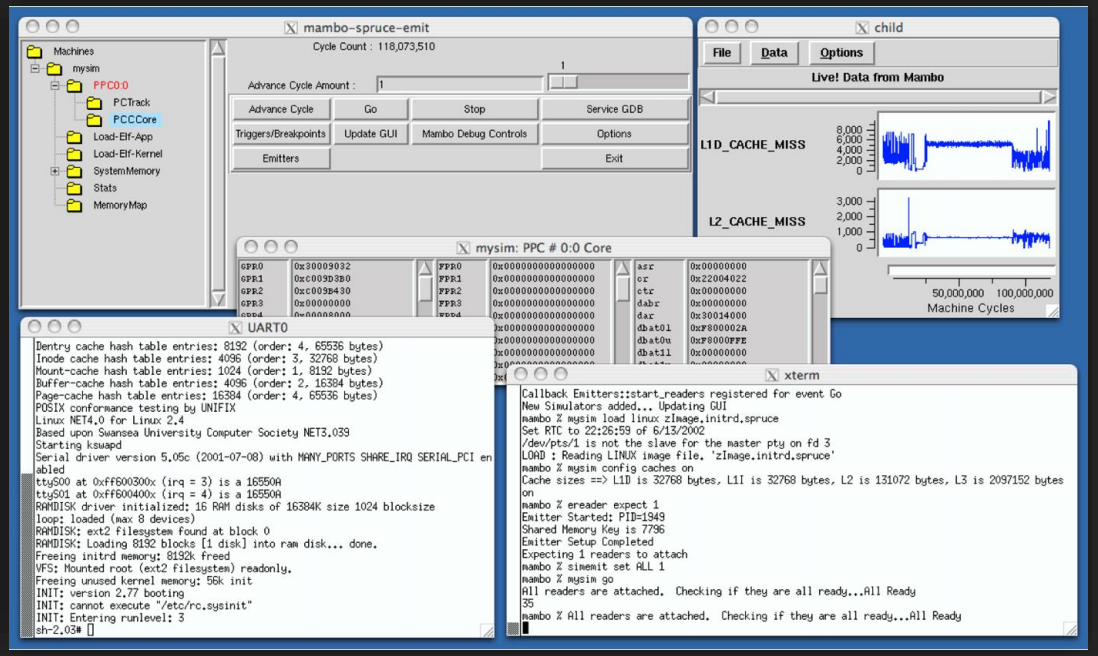


Full-System Emulation

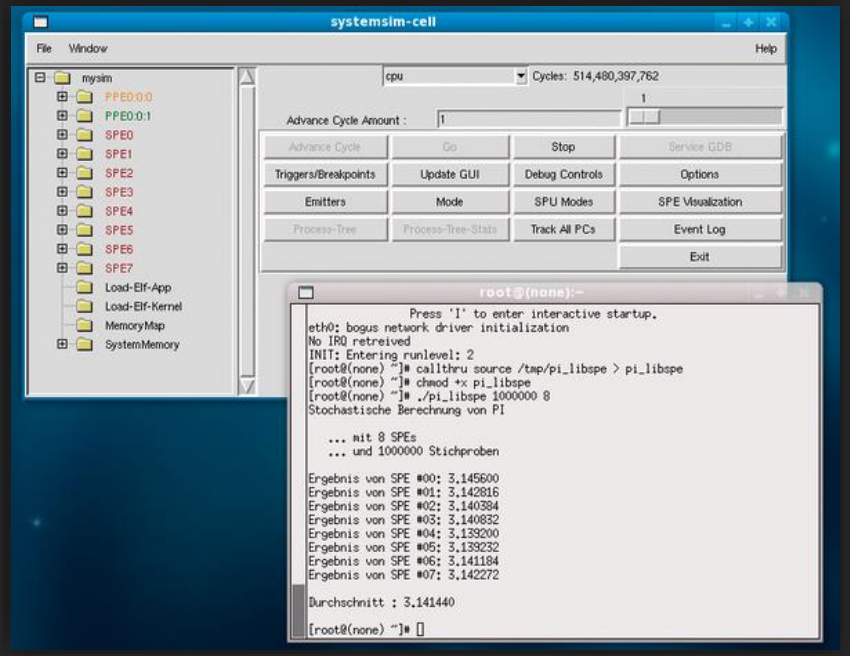
- Implement *every* observable behavior in *every* device where the machine software runs or accesses.
  - IBM Cell/B.E.: Privileged PPE/SPE modes, Nvidia RSX: All engines, Syscon, Toshiba SCC: EHCI, GBE, AHCI, SDHCI, …
- **Impractical for developers**
  - Reverse-engineering undocumented hardware that is less understood than its drivers/libraries.
  - Guest libraries/syscalls map easier to host counterparts than devices, e.g. forwarding sockets vs GBE I/O.
- **Impractical for end-users**
  - Performance overhead, e.g. multiple address space translations via software: EA → VA → PA → Host.
  - Hard-to-access firmware blobs might be required by the emulator.

Full-System Emulation

- Only for the IBM Cell/B.E. and it is closed-source.



Mambo: Full-System Simulator for the PowerPC architecture



SystemSim: IBM Full-System Simulator

User-Mode Emulation

- Considers the *execution surface* of user applications to reimplement their *direct* software/hardware dependencies.
- Frequent myths
  - *"Low accuracy"*
    Inaccuracies can always be fixed, they are just more subtle.
  - *"x10 overhead"*
    Focused on Cell/B.E. emulation, but:
    - *Address translation can be avoided*
    - *Optimizing recompilers can achieve 1:1 binary translation*

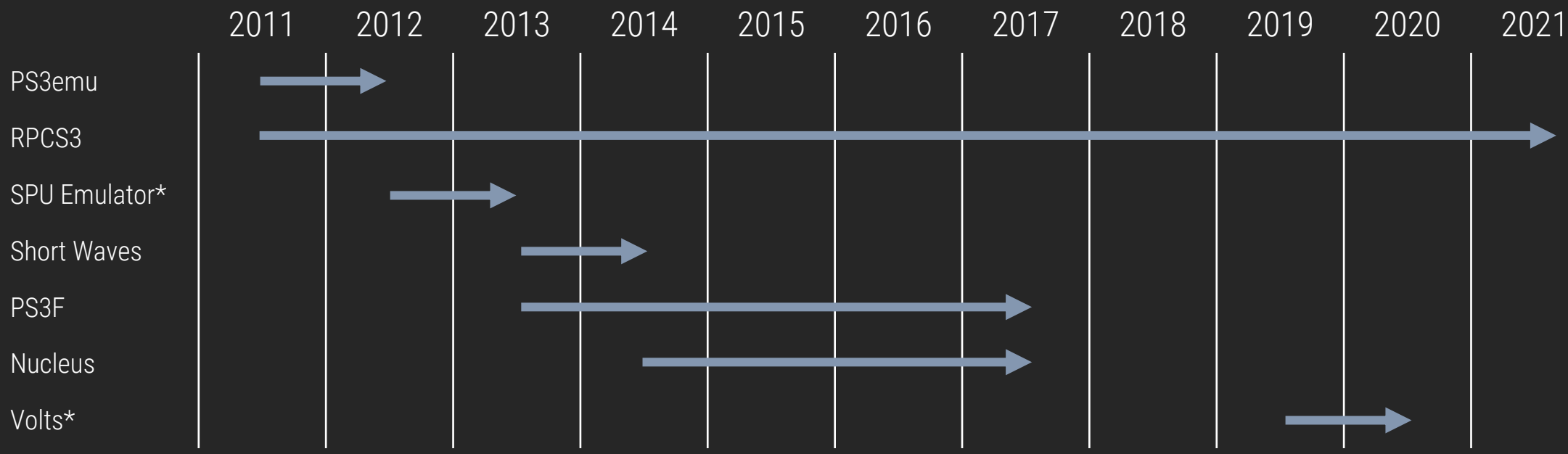
- *Shopping list*
  - PPE binary translator
  - SPE binary translator

  } Simple 32-bit EAS!

  - RSX emulation
    - PFIFO command processor
    - PGRAPH single-context translation
    - VP/FP translators
    - *(Alternatively, reimplement GCM)*
  - LV2 high-level emulation
    - *(Alternatively, reimplement PRXs)*
  - Cryptography…

User-Mode Emulation

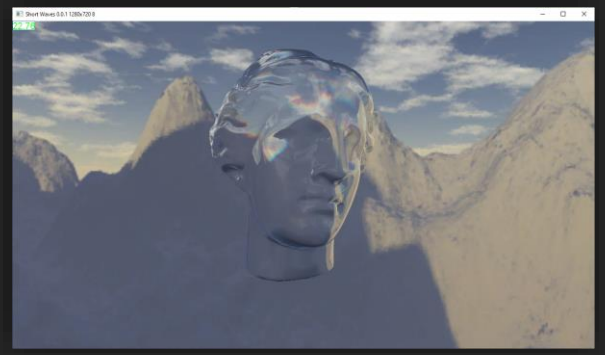
- All publicly-known PS3 emulators have chosen this approach.



Emulators

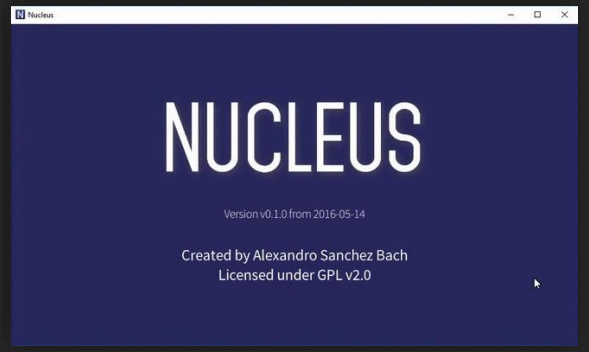
Short Waves *(InoriRus)*

- Binary translators:
  - PPU/SPU interpreters
  - PPU JIT recompiler (via C++)
- “HLE”: Reimplements SPRXs.
- Direct3D9 backend.
  - Emits HLSL code.



Nucleus *(AlexAltea)*

- Binary translators:
  - PPU/SPU interpreters
  - PPU/SPU JIT recompilers (via IR)
- “LLE”: Executes SPRXs.
- Direct3D12/Vulkan backends.
  - Emits IR to be converted to HLSL or SPIR-V (~1:1).



Emulators > Short Waves

recomp_0.cpp

```cpp
#include "recomp.h"

RECOMP_HEADER(2);

RECOMP(00010218_00010224)
{
    DECL_GPR_READ(0);
    DECL_GPR_READ(1);

    GPR_U(0) = GPR_U(0) | uint64_t(0
    { READ_MEM_8((1 ? GPR_S(1) : 0)
    GPR_U(0) = RESULT; }
    switch(256)
    {
        case 0b0000100000:
            XER = GPR_U(0);
            break;
        case 0b0100000000:
            LR = GPR_U(0);
            break;
        case 0b0100100000:
            CTR = GPR_U(0);
            break;
        default:
            EXIT("Invalid spr %d\n",
        break;
    }
    GPR_S(1) = (1 ? GPR_S(1) : 0) +

    INST_NUM(4);


    END:;
```

recomp_1.cpp

```cpp
#include "recomp.h"

RECOMP_HEADER(2);

RECOMP(00050e88_00050f1c)
{
    DECL_GPR_READ(0);
    DECL_GPR_READ(1);
    DECL_GPR_READ(2);
    DECL_GPR_READ(3);
    DECL_GPR_READ(9);
    DECL_GPR_READ(11);
    DECL_GPR_READ(31);

    { READ_MEM_8((1 ? GPR_S(1) : 0)
    GPR_U(2) = RESULT; }
    GPR_U(0) = GPR_U(3) | GPR_U(3);
    if (0)
    {
        UPDATE_CR0(GPR_S(0));
    }
    { uint32_t n = 31;
    uint64_t rs = (GPR_U(0) << 32) |
    uint64_t r = ROTL64(rs, 64-n);
    uint64_t m = MASK64(n+32, 63);
    int s = GET_BIT64(GPR_U(0), 32);
    uint64_t s64 = 0;
    if (s) s64 = ~s64;
    GPR_U(9) = (r & m) | (s64 & (~m)
    if (0)
    {
        UPDATE_CR0(GPR_S(9));
    }
```

recomp_tbl.cpp

```cpp
#include "recomp.h"

RECOMP_HEADER();
RECOMP(00010218_00010224);
RECOMP(00010230_00010248);
RECOMP(00010250_0001025c);
RECOMP(00010270_0001027c);
RECOMP(00010284_00010290);
RECOMP(00010298_000102a8);
RECOMP(000102d0_000102dc);
RECOMP(000102e4_000102f0);
RECOMP(00010354_000103a4);
RECOMP(000103ac_0001041c);
RECOMP(00010424_00010434);
RECOMP(0001044c_0001045c);
```

build.bat

```
gcc -c -O3 -o recomp_11.o recomp_11.cpp
gcc -c -O3 -o recomp_12.o recomp_12.cpp
gcc -c -O3 -o recomp_13.o recomp_13.cpp
gcc -c -O3 -o recomp_14.o recomp_14.cpp
gcc -c -O3 -o recomp_15.o recomp_15.cpp
gcc -c -O3 -o recomp_16.o recomp_16.cpp
gcc -c -O3 -o recomp_17.o recomp_17.cpp
gcc -c -O3 -o recomp_18.o recomp_18.cpp
gcc -c -O3 -o recomp_19.o recomp_19.cpp
gcc -c -O3 -o recomp_20.o recomp_20.cpp
gcc -c -O3 -o recomp_21.o recomp_21.cpp
gcc -c -O3 -o recomp_tbl.o recomp_tbl.c
gcc -shared -O3 -o recomp.dll recomp_tb
```

Emulators > Nucleus

- FOSS emulator developed between 2014 and 2017.
  - Written in C++ and cross-platform: Windows, Linux, macOS, Android and… UWP 🤦
  - PS3-oriented, but multi-guest support and custom…
    - CPU translation-oriented IR (originally LLVM), optimization passes and compiler backends
    - Graphics API with a shader IR (subset of SPIR-V) and Direct3D12/Vulkan backends
    - User Interface API: Idiomatic syntactic sugar to interactive nodes/widgets (HTML/CSS inspired)
  - Some regrets: YAGNI and Premature Optimization effect

⚠ Not to be confused with:

Nucleus: Compiler-Agnostic Function Detection in Binaries

- Released on 2017 by Dennis Andriesse et al.
- Which I contributed to, and led to some hilarious confusion

Emulators > Nucleus

Standard LLVM IR Builder

- Operates on *llvm::Value**.
- Errors caught at **guest** compile-time.

```
void Recompiler::nandx(Instruction code) {
    llvm::Value* rs = getGPR(code.rs);
    llvm::Value* rb = getGPR(code.rb);
    llvm::Value* ra;

    ra = builder.CreateAnd(rs, rb);
    ra = builder.CreateNeg(ra);
    setGPR(code.ra, ra);

    // Update CR0
}
```

Custom type-checked LLVM IR Builder

- Operates on *Value<T,N=1>*

  T ∈ `I8`, `I16`, `I32`, `I64`, `F32`, `F64` and *N > 1* for vectors.
- Errors caught at **host** compile-time.

```
void Recompiler::nandx(Instruction code) {
    Value<I64> rs = getGPR(code.rs);
    Value<I64> rb = getGPR(code.rb);
    Value<I64> ra;

    ra = builder.CreateAnd(rs, rb);
    ra = builder.CreateNeg(ra);
    setGPR(code.ra, ra);

    // Update CR0
}
```

Emulators > Nucleus

Standard LLVM IR Builder

- Operates on *llvm::Value**.
- Errors caught at **guest** compile-time.

Custom type-checked LLVM IR Builder

- Operates on *Value<T,N=1>*

  T ∈ `I8`, `I16`, `I32`, `I64`, `F32`, `F64` and *N > 1* for vectors.
- Errors caught at **host** compile-time.

```
void Recompiler::nandx(Instruction code) {
    auto rs = getGPR(code.rs);
    auto rb = getGPR(code.rb);

    auto t0 = builder.CreateAnd(rs, rb);
    auto ra = builder.CreateNeg(t0);
    setGPR(code.ra, ra);

    // Update CR0
}
```

Emulators > Nucleus

Custom type-checked LLVM IR Builder

- Operates on *Value<T,N=1>*

  T ∈ `I8`, `I16`, `I32`, `I64`, `F32`, `F64` and *N > 1* for vectors.

- Errors caught at **host** compile-time.

```
void Recompiler::nandx(Instruction code) {
    auto rs = getGPR(code.rs);
    auto rb = getGPR(code.rb);

    auto t0 = builder.CreateAnd(rs, rb);
    auto ra = builder.CreateNeg(t0);
    setGPR(code.ra, ra);

    // Update CR0
}
```

```
template <typename T=Any, int N=1>
class Value {
    llvm::Value* value;

public:
    Value(llvm::Value* v = nullptr) : value(v) {}

    operator llvm::Value*() { return value; }

    // ...
};
```

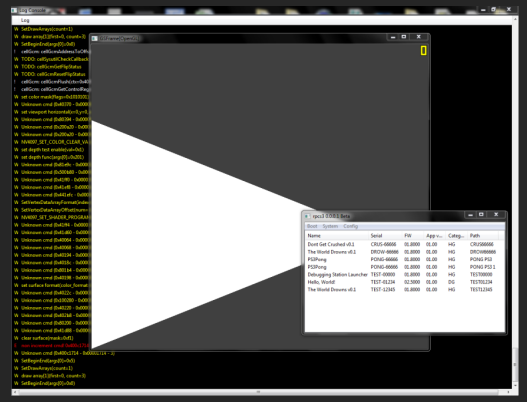
```
template <typename T, int N>
Value<T,N> CreateAnd(Value<T,N> lhs, Value<T,N> rhs) {
    static_assert(is_integer<T>::value,
        "Instruction requires integer scalar or vector");
    return builder.CreateAnd(lhs.value, rhs.value);
}
```

Emulators > RPCS3

- Focus for the rest of this talk.
- Project started by **DH** (PS2 emudev) and **Hykem** (PSP emudev/hacker), and the support of **BlackDaemon** since 2010.
- Following the noble emudev tradition of intractable names (e.g. EPSXE, PCSX2, JPCSP)
  RPCS3: "~~Russian~~ Real PC PlayStation 3" emulator
- Initial commit on May 23rd, 2011
- Originally a PS3 ELF (dis)assembler using wxWidgets and written in C++.

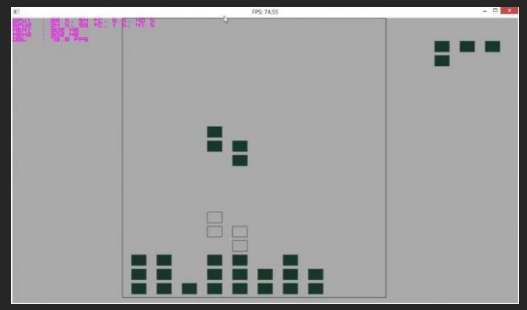
RPCS3

- Earlier years: Proof of concept running simple homebrew:
  - Fake-signed and unencrypted ELFs
- Accomplished with:
  - Userland PPU interpreter
    - Experimental SPU interpreter
  - Simple LV2/PRX HLE covering: Threading/synchronization, timers, filesystem, graphics, gamepad.
  - RSX emulation with OpenGL/GLSL.
- Original disassembler evolved into a debugger.



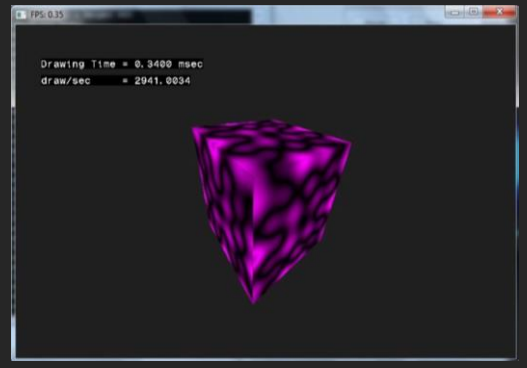
~2011

RPCS3 v0.0.0.1 Beta



~2012

RPCS3 v0.0.0.2 Beta

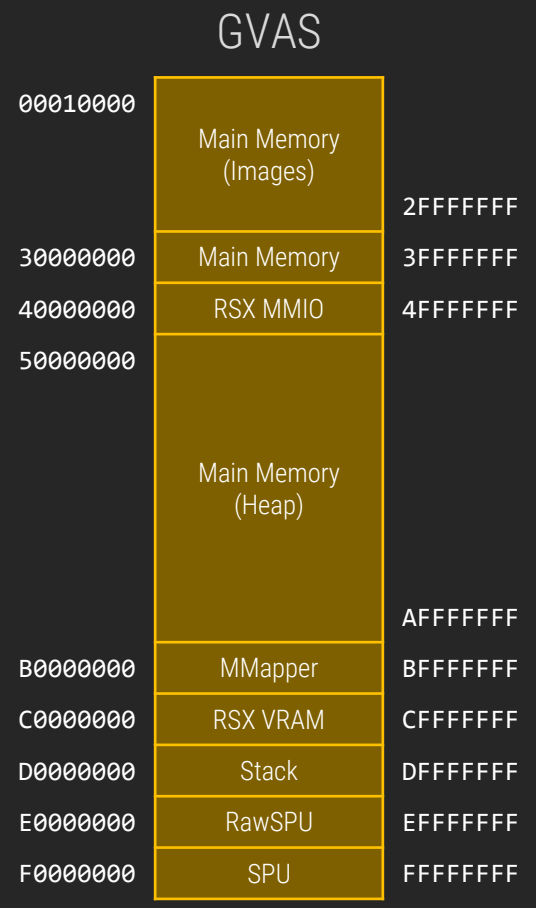


~2013

RPCS3 v0.0.0.3 Beta

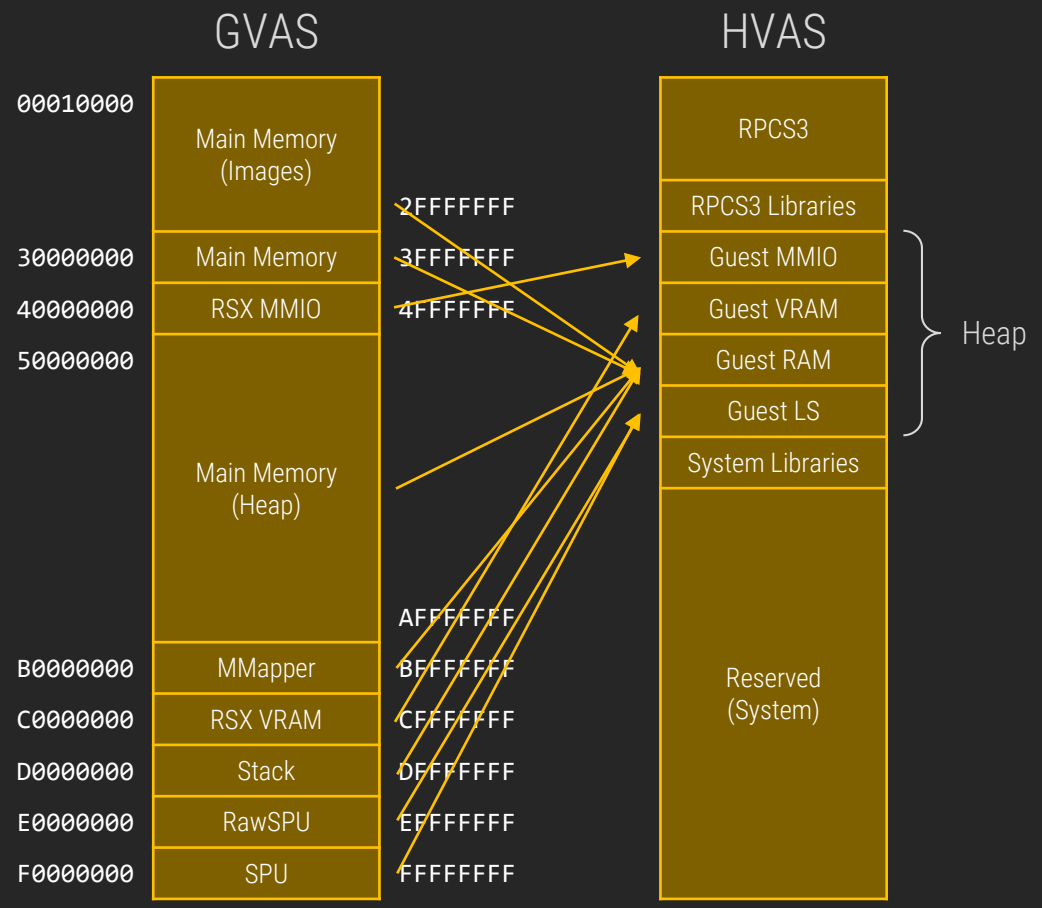
RPCS3 > Memory

- Originally, RPCS3 supported both 32-bit and 64-bit hosts.
- The guest virtual address space is 32-bit with fixed memory maps.
  - Games/libs hardcode addresses!



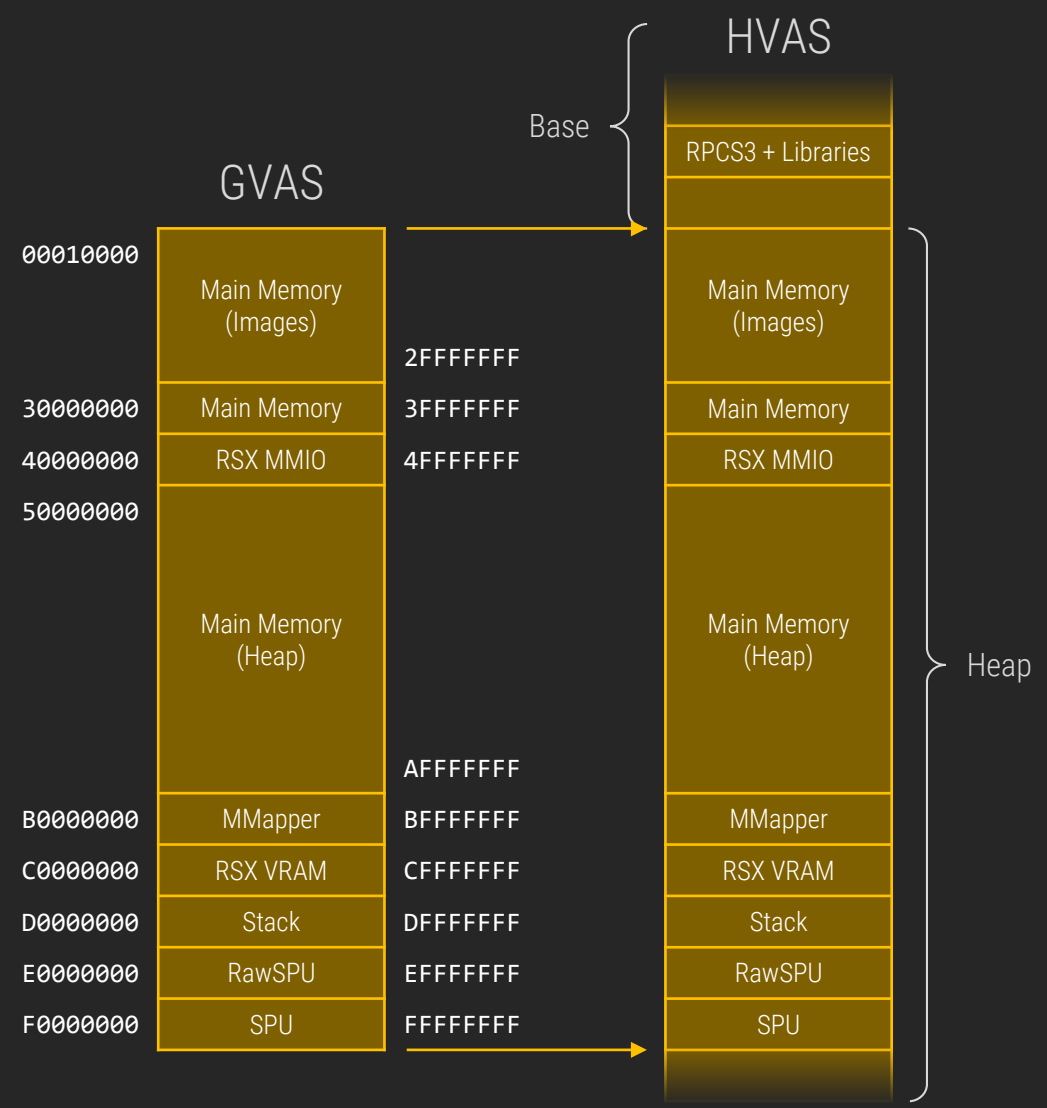
RPCS3 > Memory

- Originally, RPCS3 supported both 32-bit and 64-bit hosts.
- The guest virtual address space is 32-bit with fixed memory maps.
  - Games/libs hardcode addresses!
- First solution:
  Traverse array of memory blocks
  - Slight linear search penalty on every memory access in guest code.
  - Only pre-allocate hardcoded blocks.



RPCS3 > Memory

- Dropping 32-bit hosts support simplified memory translation.
- **Reserving the entire 4 GB GVAS** in the host process enables:
  - Constant time guest access with negligible overhead.
    - Single addition of a base address!
  - Hardcoded blocks still pre-allocated.



RPCS3 > Memory

- **Implicit endian conversion**
  - `be_t<T>` Big-Endian
  - `le_t<T>` Little-Endian
- Alias to `T` or `se_t<T>` to select *"native"* or *"swapped"* endian.

```
struct args {
    be_t<u32> lhs; // 00 00 00 37 = 0x37   (BE)
    be_t<u32> rhs; // 00 00 13 00 = 0x1300 (BE)
};

u32 hle_add(args* a) {
    return a->lhs
         + a->rhs; // 37 13 00 00 = 0x1337 (LE)
}
```

Disclaimer: Gross oversimplification

```
template <typename T, /* ... */>
class se_t {
    T data;

    static constexpr T swap(T value) {
        // ... return __builtin_bswapN(value);
    }

public:
    // Implicit conversion to T
    constexpr operator T() const noexcept {
        return swap(data);
    }

    // Implicit conversion from T
    se_t<T>(T rhs = 0) : data(swap(rhs)) {}
    se_t<T>& operator=(T rhs) {
        data = swap(rhs);
        return *this;
    }

    // Operators...
    template <typename R>
    se_t<T>& operator+=(const se_t<R>& rhs) {
        return *this = T(*this) + R(rhs);
    }
};
```

RPCS3 > Memory

- **Implicit memory translation**
  - `vm::ptr<T>` to represent `T*` in the guest virtual address space.
    - Both address and pointed primitive types auto-casted from/to `be_t`.

```
struct args {
    be_t<u32> lhs;
    be_t<u32> rhs;
    vm::ptr<be_t<u32>> r;
};

void hle_add(vm::ptr<args> a) {
    a->r = a->lhs + a->rhs;
}
```

```
/* Instead of...
void hle_add(u32 a_ptr) {
    auto a = reinterpret_cast<args*>(
        ptr(a_ptr));
    auto r = reinterpret_cast<u32*>(
        ptr(__builtin_bswap32(a->r));
    *r = __builtin_bswap32(
        __builtin_bswap32(a->lhs) +
        __builtin_bswap32(a->rhs);
} */
```

Disclaimer: Gross oversimplification

```
inline void* base(u32 addr) {
    return g_base_addr + addr;
}

template <typename T>
class ptr {
    be_t<u32> addr;

public:
    ptr(u32 addr = 0) : addr(addr) {}

    // Implicit conversion to T*
    T* operator->() const {
        return static_cast<T*>(vm::base(addr));
    }

    // Pointer arithmetic...
    ptr<T> operator+(u32 n) const {
        return addr + n * sizeof(T));
    }

    // ...
};
```

RPCS3 > CPU

- Fundamental part of emulators.
  - Mistakes can be hard to debug.
  - Robust CPU emulation improves development speed elsewhere.
- Cell/B.E. emulation improvements during 2013~2014 led to the first "*playable*" commercial games.
  - PPU/SPU interpreters since 2011.
  - PPU/SPU recompilers since 2014.
    - Only x86 hosts; ARM in-progress.
    - Started by *Nekotekina*/*Gopalsr83*.

Arkedo Series: 02 - SWAP!

~2014 Q1

Disgaea 3: Absence of Justice

~2014 Q2

RPCS3 v0.0.0.5 Beta

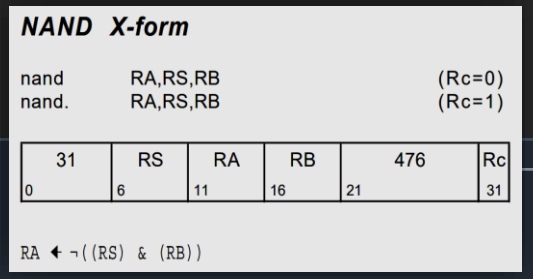
RPCS3 > CPU

- Most work for emulator developers involve:
  - Reading ISA specifications and implementing instructions of the following types: ALU: Integer, Floating-Point, Vector; Control Flow; Memory; Control/Status.
  - Testing/fixing instruction accuracy.
  - Maintaining/improving performance.
- Our focus:
  - Cover RPCS3 PPU/SPU translators and their evolution in chronological order.
  - Provide binary translation examples: 1 ALU/CF instruction (seen one, seen all).
  - Explain edge cases and workarounds.
  - CPU-related peripherals, e.g. SPU MFC.

RPCS3 > CPU > PPU

- **PPU Interpreter** (~2011)
  - PPU state/registers in-memory.
  - Simple fetch-decode-execute loop.
- Custom bitfields decoder: Extracts opcode and arguments, dispatch to instruction methods.
- Translator/disassembler dispatch via **dynamic polymorphism** and so-called “instruction binders”
  - Needless indirection/complexity.

Disclaimer: Gross oversimplification




```
struct PPUThread {
    u64 lr, pc;
    u64 gpr[32];
    f64 fpr[32];
    union { u8 b[16]; u16 h[8]; u32 w[4]; /* ... */ } vr;
    // ...
};

class PPUDecoder { /* void Decode(u32 code); ... */ };

class PPUInterpreter : public PPUDecoder {
    PPUThread& ppu;

    virtual void B(s32 ll, u32 aa, u32 lk) {
        ppu.SetPC(aa ? ll : cpu.pc + ll); // Deferred
        if (lk) cpu.lr = cpu.pc + 4;
    }
    virtual void NAND(u32 ra, u32 rs, u32 rb, bool rc) {
        ppu.gpr[ra] = ~(ppu.gpr[rs] & ppu.gpr[rb]);
        if (rc) ppu.UpdateCR0<s64>(ppu.gpr[ra]);
    }
    // ...
};
```

RPCS3 > CPU > PPU

- **PPU Interpreter** (~2015)
- Refactor by *Nekotekina*.
- Switched to instruction dispatch via **“*static*” polymorphism**.
  - Some indirection is unavoidable due to opcode dispatch tables
- Optimizations specific to x86
  - Accelerated VMX instructions with SSE/AVX intrinsics.
    - Auto-vectorization sometimes fails

Disclaimer: Gross oversimplification

```
struct PPUThread { /* ... */ };

template <typename T>
class PPUDecoder { /* ... { [0x300] = &T::VADDSBS; } */ };

class PPUInterpreter : public PPUDecoder<PPUInterpreter> {
    void VADDSBS(u32 vd, u32 va, u32 vb) {
        ppu.vr[vd].m128i = _mm_adds_epi8(
            ppu.vr[va].m128i, ppu.vr[vb].m128i);
        /* Instead of ...
        for(u32 i = 0; i < 16; i++) {
            s16 result = (s16)ppu.vr[va].b[i] +
                         (s16)ppu.vr[vb].b[i];
            if (result > 0x7f)
                ppu.vr[vd].b[i] = 0x7f;
            else if (result < -0x80)
                ppu.vr[vd].b[i] = -0x80;
            else
                ppu.vr[vd].b[i] = result;
        } */
    }
    // ...
};
```

RPCS3 > CPU > PPU

- **PPU JIT Recompiler** (~2014)
- Originally written by *Gopalsr83*.
- Simple LLVM-based translator
  - Basic blocks as translation units
  - Every branch returns to the emulator where the next block is looked
    - When `LR` is accessed, emit `CALL`/`RET`
  - Register access on `PPUThread` via `CreateAligned{Load,Store}`
    - Some maybe optimized-away by LLVM, e.g. on Dead Store Elimination pass

Disclaimer: Gross oversimplification

```
struct PPUThread { /* ... */ };

template <typename T>
class PPUDecoder { /* ... */ };

class PPURecompiler : public PPUDecoder<PPURecompiler> {
    void NAND(u32 ra, u32 rs, u32 rb, bool rc) {
        auto result = ir.CreateNeg(
            ir.CreateAnd(GetGpr(rs), GetGpr(rb)));
        SetGpr(ra, result);
        if (rc) UpdateCr0(result);
    }
    void ISYNC() {
        ir.CreateCall(/* ... */ Intrinsic::x86_sse2_mfence));
    }
    void VADDSBS(u32 vd, u32 va, u32 vb) {
        ir.CreateCall(/* ... */ Intrinsic::x86_sse2_padds_b),
            GetVrAsIntVec(va, 8), GetVrAsIntVec(vb, 8));
        // ...
    }
    // ...
};
```

RPCS3 > CPU > PPU

- **PPU AOT Recompiler** (~2016)
- Originally written by *Nekotekina*.
- Still LLVM IR, same ALU code, but
  - Entire modules as translation units
    - Feasible thanks to RWX/JIT ban
    - First run is latency-free!
  - Removed x86-specific intrinsics
  - Moved to a type-checked builder with overloaded operators
- PPU analyzer!

Disclaimer: Gross oversimplification

```
/* PPUThread, PPUDecoder<T>, ... */

class PPUTranslator : public PPUDecoder<PPUTranslator> {

    void NAND(u32 ra, u32 rs, u32 rb, bool rc) { /* ... */ }

    void ISYNC() {
        ir.CreateFence(AtomicOrdering::Acquire);
    }

    void VADDSBS(u32 vd, u32 va, u32 vb) {
        const auto [a, b] = get_vrs<s8[16]>(op.va, op.vb);
        const auto r = add_sat(a, b); // Intrinsic::sadd_sat
        set_vr(op.vd, r);
        set_sat(r ^ (a + b));
    }

    // ...
};
```

RPCS3 > CPU > SPU

Similar evolution

- ~2012 **SPU Interpreter**
- ~2014 **SPU JIT Recompiler** (ASMJIT)
  - Direct SPU-to-x86 translation; no IR
    - Low latency, important for JITs
    - Built-in register allocator
  - AVX-512 support (+20% perf)
- ~2018 **SPU JIT Recompiler** (LLVM)
  - Function granularity
  - Higher performance over ASMJIT

- LLVM recompiler is preferred
  - Avoids needsless register load/stores by tracking accesses per-block
  - Pre-analyzes the function to generate a complete CFG

- We'll not cover translation again!

RPCS3 > CPU > SPU

SPU Channels are I/O-like:

- Fixed numbers in `rdch`/`wrch`
  - Insert call to native handler function
  - Sometimes, even inline LLVM-IR!
- Some channels behave memory-like, with atomic accesses
  - Emit `load+store` or `atomicrmw`
  - Targets typically are `U32` or…
    - **`spu_channel`**: `atomic_t<u64>` wrapper with *push*/pop** methods for access.
    - **`spu_channel_4`**: Same holding a queue of 4 values, used for SPU mailboxes.

MFC Emulation

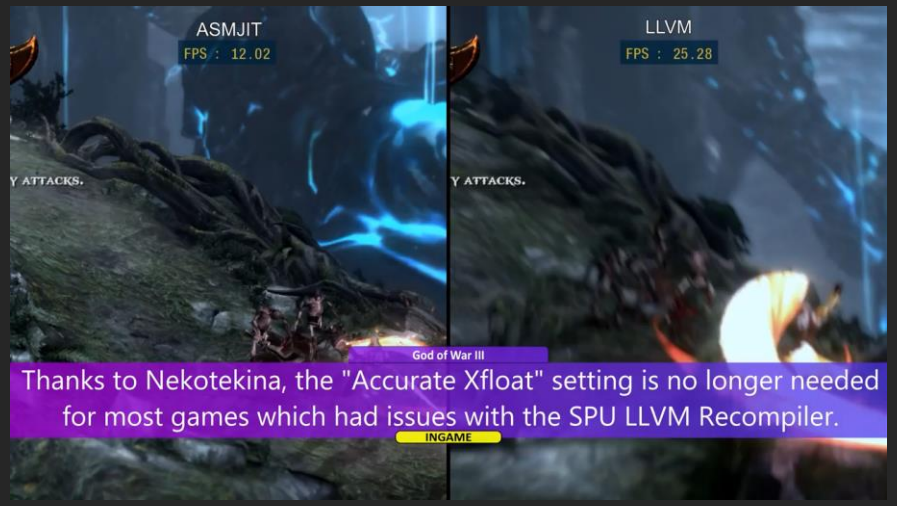
- Same EA space ⇒ no soft-MMU
- Requests initiated via `wrch MFC_Cmd,$reg`
  - Inline memcpy-like LLVM IR with optional fences.
    - Interpreter fallback for debugging.
  - Pseudocode example for `PUT`

```
src = gep(lsptr,  zext<u64>(MFC_EAL));
dst = gep(memptr, zext<u64>(MFC_LSA));
// Swap if GET, check barriers/frences, …
switch (MFC_Size) {/* t = type<u8*>(); */}
store(load(bitcast(src,t),bitcast(dst,t)));
```

RPCS3 > CPU > SPU

- Optional-solutions to edge cases
  - ***Accurate Xfloat***
    Emulating non-IEEE F32 behavior: ±*smax*-clamp and round-to-zero.
    1. Extend sign/exponent/mantissa bits from `U32[4]` to `F64[4]`
    2. Do the operation (on AVX2/YMM!)
    3. Bitcast/truncate back to `U32[4]`, clamping and rounding as needed
  - ***Approximate Xfloat***
    - Math: Apply clamps, set denorms to 0. *<smin*
    - Comparisons: Bitcast to integer.

Accurate vs approximate xfloat



RPCS3 - God of War 3 Major Performance Improvement!
(via the Official RPCS3 YouTube channel)

RPCS3 > GPU

- **RSX frontend**
  - Heavily changing ever since 2012
  - Manages guest GPU
    - PFIFO thread
    - PGRAPH state
    - DMA transfers
    - Common helpers for backends, e.g. converting non-standard formats

Dispatches

Trivia: References in the codebase to *GS* originate in DH's early work in PS2 emulation and its so-called *Graphics Synthesizer*. ☺

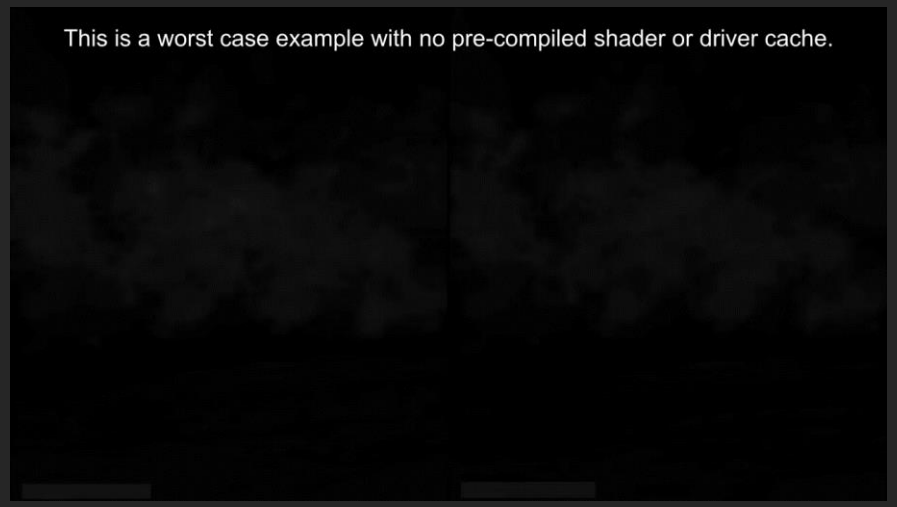
- **Vulkan backend** for RSX (~2016)
  - Created/maintained by *kd-11*.
  - Within weeks of the Vulkan release!
  - OpenGL/D3D12 backends exist(ed)
  - Manages host GPU → Out-of-scope!
    - Translates draw calls
    - Reports (e.g. Zcull occlusion queries)
    - Supports RSX frontend with compute shaders (e.g. deswizzling)
    - Exposes the render surface to UI

RPCS3 > GPU

- **RSX shader translation**:
  Emits GLSL, compiled to SPIR-V
  - Host GPU driver takes few milliseconds of CPU time to compile shaders.
    - Games have stuttering in new scenes.
- Fortunately, completing draws is not critical to applications...
- *Asynchronous shader translation*
  Compile in other threads, and issue draw calls only for complete pipelines.
  - New scenes may be incomplete
  - *Shader Interpreter* (aka Ubershaders)
    - Tricky accuracy-vs-performance, but better than nothing!

- Helpful resources
  - `NV_vertex_program*`
  - `NV_fragment_program*`

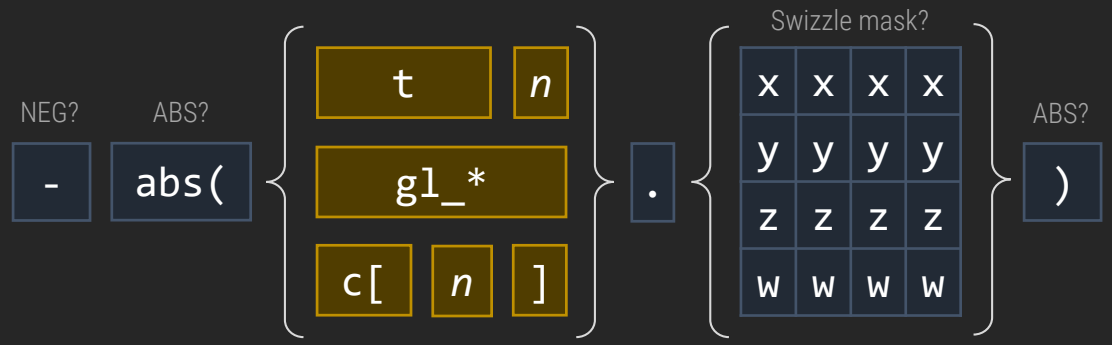
Synchronous vs asynchronous shader translation



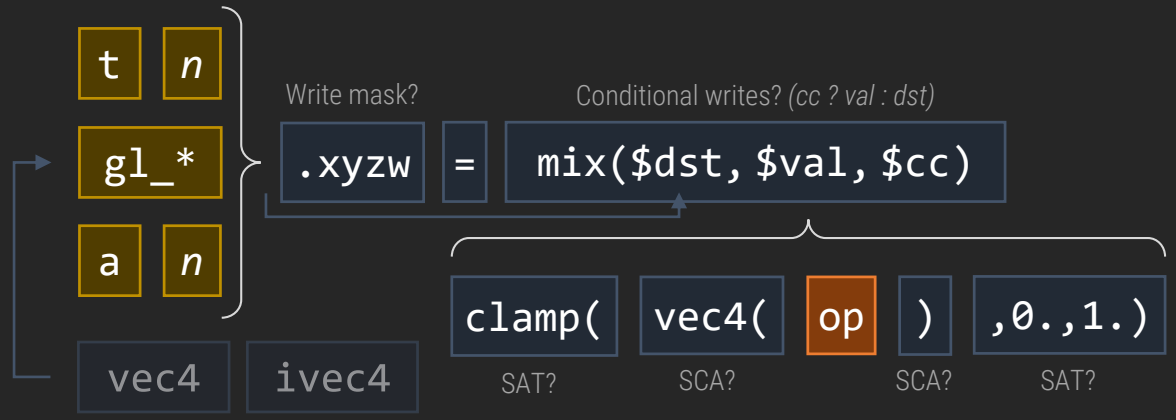
RPCS3 - Eliminating Stutter with Asynchronous Shader Implementation!
(via the Official RPCS3 YouTube channel)

RPCS3 > GPU > VP

- Vector sources (**$0**, **$1**, **$2**)



- Vector destination

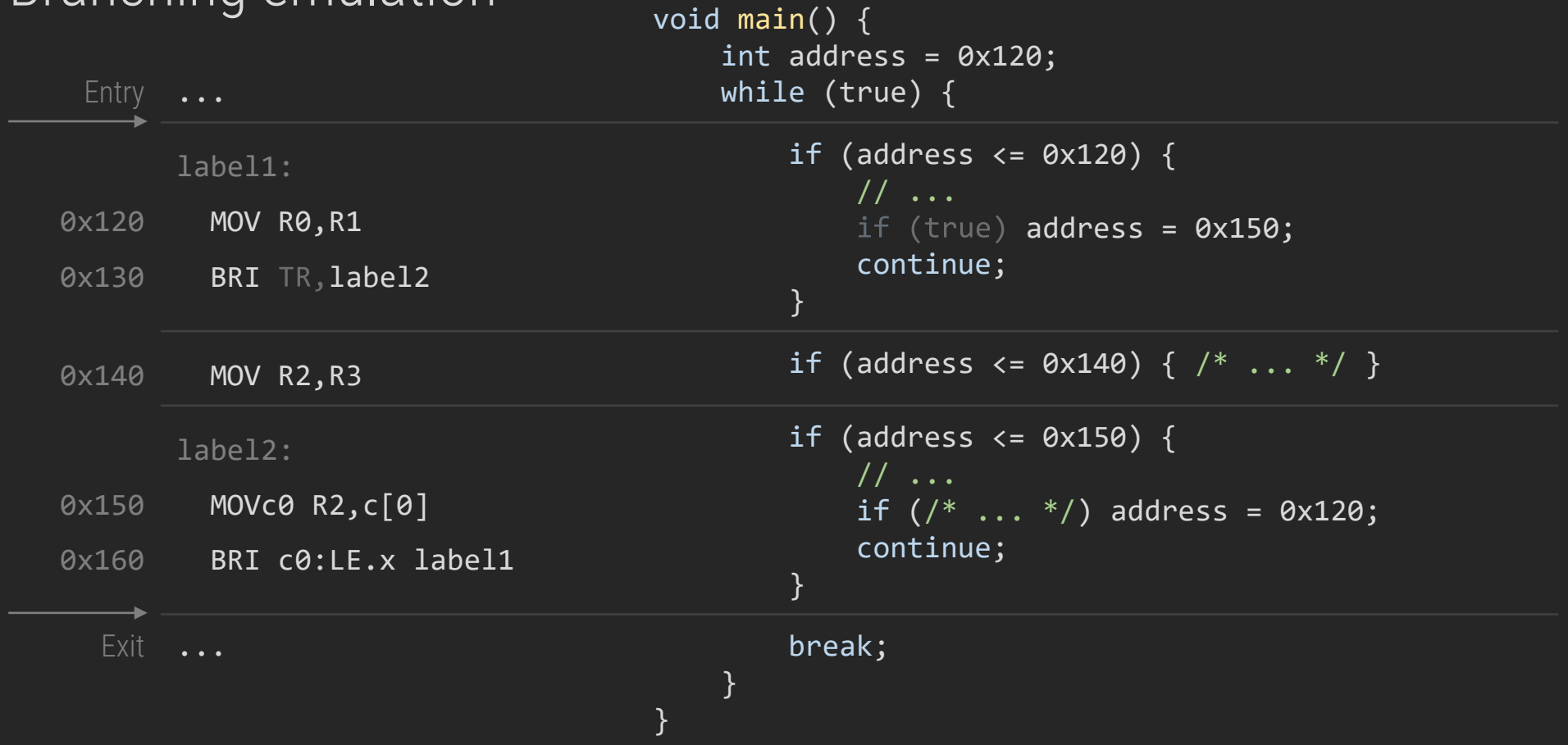


- Opcodes
  - MOV `"$0"`
  - MUL `"$0 * $1"`
  - ADD `"$0 + $2"`
  - FLR `"floor($0)"`
  - MIN `"min($0, $1)"`
  - MAD `"fma($0, $1, $2)"`
  - RCP `"(1.0 / $s)"`
  - RSQ `"1./sqrt(max($s, 0.0000000001))"`
  - RCC `"clamp(1.0 / $s, 5.42101e-20, 1.884467e19)"`
  - LIT `"lit_legacy($s)"`
  - ...

```
vec4 lit_legacy(const in vec4 val) {
    vec4 cval = val;
    cval.x = max(val.x, 0.);
    cval.y = max(val.y, 0.);
    vec4 r; r.x = 1.; r.w = 1.; r.y = cval.x;
    r.z = cval.x > 0. ?
        exp(cval.w * log(max(cval.y, 0.0000000001))) : 0.;
    return r;
}
```

RPCS3 > GPU > VP

- Branching emulation



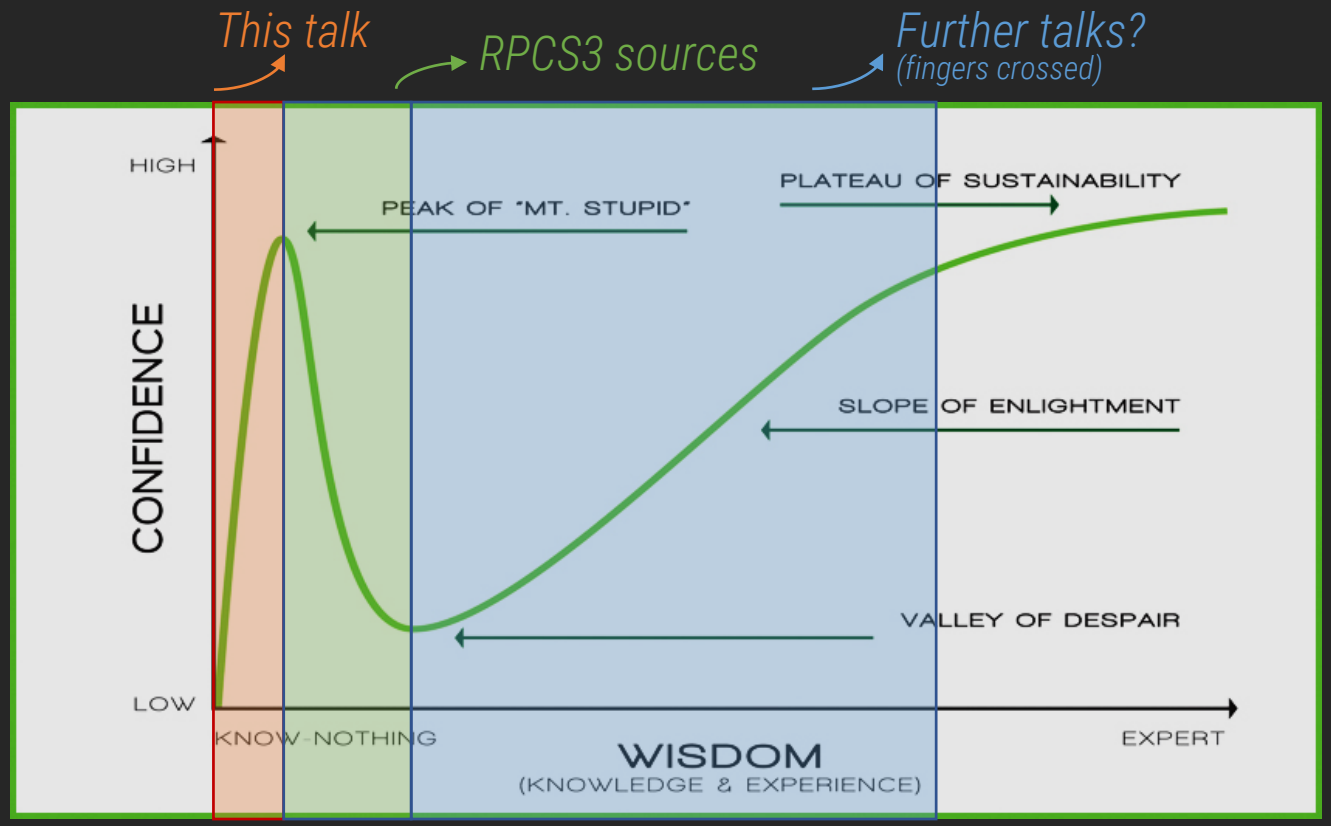
RPCS3 > GPU > FP

- Same with fragment programs
  - Texture samplers turn into `sampler{1D,2D,3D,Cube}`
  - `KIL` to `discard` if drivers allow
- Half-precision registers via
  `GL_EXT_shader_explicit_arithmetic_types_float16`
  `VK_KHR_shader_float16_int8`
  - Types: `float16_t`, `f16vec{2,3,4}`
  - When unavailable (or broken drivers) emulate with F32 and clamping.

- Pack/Unpack operations
  - `PK2` `"vec4(uintBitsToFloat(packHalf2x16($0.xy)))"`
  - `PK4` `"vec4(uintBitsToFloat(packSnorm4x8($0)))“`
  - `UP2` `"unpackHalf2x16(floatBitsToUint($0.x)).xyxy"`
  - `UP4` `"unpackSnorm4x8(floatBitsToUint($0.x))"`
  - `...` (some with gamma correction: `PKG`/`UPG`)
- Many hardware tests by *kd-11*.
  - Checking for IEEE-compliance.
    e.g. `MUL`/`DIV`/`EX2`/`LG2` are, but…
  - `RSQ` ignores the input sign
  - `DIVSQ` *always* computes $0/\sqrt{x}$ as 0
  - `SAT` flag flushes NaN to 0

RPCS3 > GPU

- This is just an overview! Beware the Dunning-Kruger effect.



RPCS3 > HLE

- RPCS3 reimplements both
  - *Syscalls*: Hook PPU `sc` instruction
  - *Libcalls*: Hook `addr:rtoc` import
- Hook implementation
  1. Manager registers native functions identified by an index into an array.
  2. When translating, emit IR branch to an ASMJIT-generated trampoline function that switches between guest and host contexts.

- Motivation
  - Performance gains:
    e.g. `cellVdec` via *ffmpeg*.
  - Emulation without firmware when reimplementing all modules (WIP).
  - Cheating around bugs to parallelize development of components.
- Some actual disadvantages:
  - "*Higher development efforts*"
    Would you reimplement a proprietary video decoder?

RPCS3 > HLE

- **Implicit ABI conversion**
  - Parameter/return values on registers/stack:
    ```
    ARG_GENERAL  // in: r3-r10, out: r3
    ARG_FLOAT    // in: f1-f13, out: f1-f4
    ARG_VECTOR   // in: v2-v13, out: v2
    ```
  - Parameter register allocation depends on argument position and previous argument types.
  - Convert argument/return values from/to PPU registers.

```
// int hle_add(u32 lhs, u32 rhs, u32* out);
int hle_add(PPUThread& ppu,
      u32 lhs, u32 rhs, vm::ptr<u32> out) {
    out = lhs + rhs;
    return CELL_OK;
}
```

```
/* Instead of...
void hle_add(PPUThread& ppu) {
    auto lhs = static_cast<u32>(ppu.gpr[3]);
    auto rhs = static_cast<u32>(ppu.gpr[4]);
    auto out = reinterpret_cast<u32*>(
        ptr(static_cast<u32>(ppu.gpr[5])));
    *out = lhs + rhs;
    ppu.gpr[3] = CELL_OK;
} */
```

Disclaimer: Gross oversimplification

```
template <typename T, arg_class type>
void bind_result(PPUThread& ppu, const T& value) {
    if constexpr (type == ARG_GENERAL) // std::is_integral_v<T>…
        ppu.gpr[3] = value;
    if constexpr (type == ARG_FLOAT) // std::is_floating_point_v<T>…
        ppu.fpr[1] = value;
    // ...
}
template <typename T, arg_class type, int ng, int nf, int nv>
T bind_arg(PPUThread& ppu) {
    if constexpr (type == ARG_GENERAL)
        return ppu.gpr[ng + 3];
    if constexpr (type == ARG_FLOAT)
        return ppu.gpr[nf + 1];
    // ...
}
template <typename TR, typename... TAs>
void do_call(PPUThread& ppu, TR(*func)(TAs...)) {
    // Turn TAs into arg_class tracking ng/nf/nv at compile-time.
    // Then extract and forward to func using bind_arg.
    bind_result(ppu, call<T...>(ppu, func, arg_info_pack_t<>{}));
}
```

Even better: See Tony Wasserka at C++Now 2018:
"Generative Programming in Action: Emulating the 3DS"

Testing

“*Unit*” tests *(Nucleus)*

- Tests run in **host** machine.
- Steep development curve.
  - Create assemblers, stabilize API.
  - Emulator-specific.
- More reliable, self-contained.

```
void PPCTestRunner::addic() {
    // Add Immediate Carrying
    TEST_INSTRUCTION(test_addic, R1, SIMM, R2, CA, {
        state.r[1] = R1;
        run({ a.addic(r2, r1, SIMM); });
        expect(state.r[2] == R2);
        expect(!state.cr.field[0].lt);
        expect(!state.cr.field[0].gt);
        expect(!state.cr.field[0].eq);
        expect(!state.cr.field[0].so);
        expect(!state.xer.so && !state.xer.ov);
        expect(state.xer.ca == U08(CA));
    });
    test_addic(0x0000000010000FFFFULL, 0x0001,
               0x0000000010010000ULL, false);
    test_addic(0x00000000FFFFFFFFULL, 0x0001,
               0x0000000100000000ULL, false);
    test_addic(0x00000000FFFF0001ULL, 0xFFFF,
               0x00000000FFFF0000ULL, true);
    test_addic(0xFFFFFFFFFFFFFFFFULL, 0x0001,
               0x0000000000000000ULL, true);
}
```

Run All | Run... | Playlist : All Tests

Category [PowerPC Tests] (377)	
bcx	182 ms
frsqrtex	8 ms
vexptefp	8 ms
vmaddfp	1 sec
vmaxfp	149 ms
vmaxsh	162 ms
vmaxuh	121 ms
vminfp	139 ms
addcx	24 ms
addex	24 ms
addi	18 ms
addic	11 ms
addic_	14 ms
addis	18 ms
addmex	< 1 ms
addx	13 ms
addzex	36 ms
andcx	14 ms

Summary

Last Test Run Failed

8 Tests Failed

274 Tests Passed

“*Integration/Fuzz*” tests *(RPCS3, Nucleus)*

- Tests run in **guest** machine.
- Gentle development curve
  - Create with homebrew SDKs.
  - Emulator-agnostic, cf. `ps3autotests`.
- Less reliable, many dependencies.

```
// Interesting numbers
uint64_t testInts64[] = {
  0x0000000000000000LL, 0x0000000000000001LL, // 0 1
  0x0000000000000002LL, 0xFFFFFFFFFFFFFFFFLL, // 2 3
  0xFFFFFFFFFFFFFFFELL, 0x0003333300330033LL, // 4 5
  0x000000FFFFF00000LL, 0x1000000000000000LL, // 6 7
  /* ... */
};

#define ITERATE1 /* ... For each r1 in testInts64 */
#define PRINT2   /* ... Print inputs and outputs */

ITERATE1(__asm__("addic   %0,%1,%2" :
                 "=r"(r0) : "r"(r1), "i"(0));
         PRINT2("addic  ", i, 0, r0));
ITERATE1(__asm__("addic   %0,%1,%2" :
                 "=r"(r0) : "r"(r1), "i"(+1));
         PRINT2("addic  ", i, +1, r0));
ITERATE1(__asm__("addic   %0,%1,%2" :
                 "=r"(r0) : "r"(r1), "i"(-1));
         PRINT2("addic  ", i, -1, r0));
```

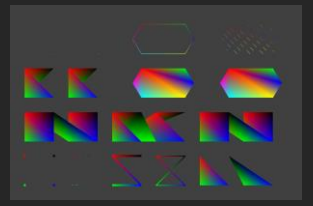
```
[...]
addic  ([04],[00]) -> FFFFFFFFFFFFFFFE
[0000000000000000 : 00000000]
addic  ([05],[00]) -> 0003333300330033
[0000000000000000 : 00000000]
addic  ([06],[00]) -> 000000FFFFF00000
[0000000000000000 : 00000000]
addic  ([07],[00]) -> 1000000000000000
[0000000000000000 : 00000000]
addic  ([08],[00]) -> 1FFFFFFFFFFFFFFF
[0000000000000000 : 00000000]
addic  ([09],[00]) -> 4238572200000000
[0000000000000000 : 00000000]
addic  ([10],[00]) -> 7000000000000000
[0000000000000000 : 00000000]
addic  ([11],[00]) -> 0000000072233411
[0000000000000000 : 00000000]
[...]
```

Testing

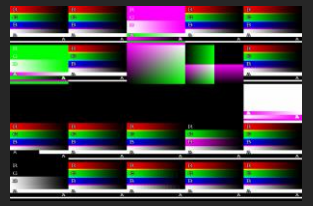
“*Integration/Fuzz*” tests *(RPCS3, Nucleus)*

- Tests run in **guest** machine.
- Gentle development curve
  - Create with homebrew SDKs.
  - Emulator-agnostic, cf. `ps3autotests`.
- Less reliable, many dependencies.
- But might be the most cost-effective way of discovering bugs and regressions.
- Tests exist for PPU/SPU, RSX and LV2.
- Executed first on a real PlayStation 3 console, output files are ground truth for the emulator.

RSX Primitives

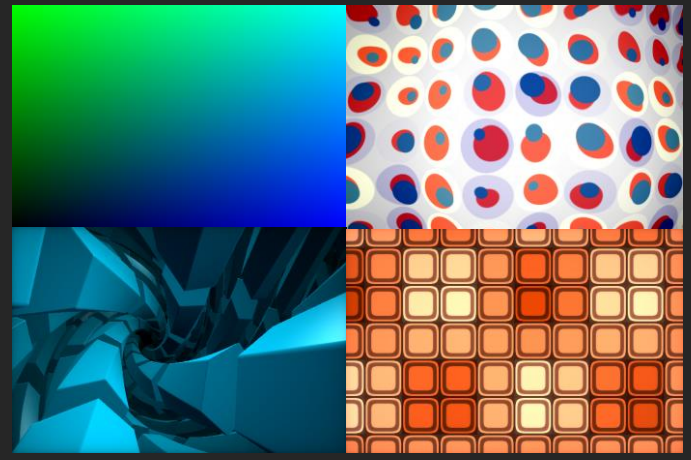


RSX Textures



RSX FPs

Thanks to Shadertoy!



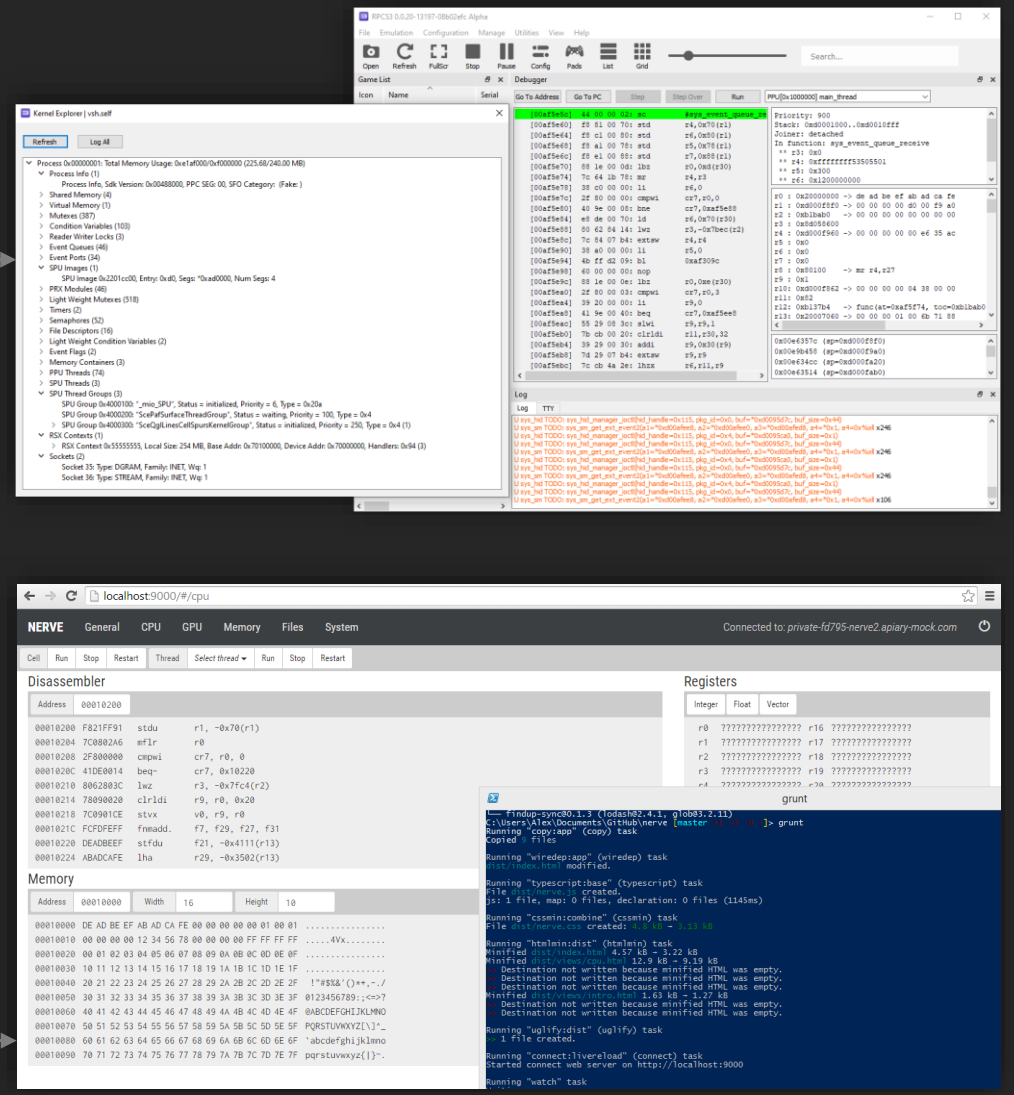
PPU Instructions

```
// Interesting numbers
uint64_t testInts64[] = {
  0x0000000000000000LL, 0x0000000000000001LL, // 0 1
  0x0000000000000002LL, 0xFFFFFFFFFFFFFFFFLL, // 2 3
  0xFFFFFFFFFFFFFFFELL, 0x0003333300330033LL, // 4 5
  0x000000FFFFF00000LL, 0x1000000000000000LL, // 6 7
  /* ... */
};

#define ITERATE1 /* ... For each r1 in testInts64 */
#define PRINT2   /* ... Print inputs and outputs */

ITERATE1(__asm__("addic   %0,%1,%2" :
                 "=r"(r0) : "r"(r1), "i"(0));
         PRINT2("addic  ", i, 0, r0));
ITERATE1(__asm__("addic   %0,%1,%2" :
                 "=r"(r0) : "r"(r1), "i"(+1));
         PRINT2("addic  ", i, +1, r0));
ITERATE1(__asm__("addic   %0,%1,%2" :
                 "=r"(r0) : "r"(r1), "i"(-1));
         PRINT2("addic  ", i, -1, r0));
```

```
[...]
addic  ([04],[00]) -> FFFFFFFFFFFFFFFE
[0000000000000000 : 00000000]
addic  ([05],[00]) -> 0003333300330033
[0000000000000000 : 00000000]
addic  ([06],[00]) -> 000000FFFFF00000
[0000000000000000 : 00000000]
addic  ([07],[00]) -> 1000000000000000
[0000000000000000 : 00000000]
addic  ([08],[00]) -> 1FFFFFFFFFFFFFFF
[0000000000000000 : 00000000]
addic  ([09],[00]) -> 4238572200000000
[0000000000000000 : 00000000]
addic  ([10],[00]) -> 7000000000000000
[0000000000000000 : 00000000]
addic  ([11],[00]) -> 0000000072233411
[0000000000000000 : 00000000]
[...]
```

Debugging

- Both RPCS3 and Nucleus support guest CPU/GPU and HLE debugging
  - Disassembler for PPE/SPE code, RSX FP/VP shaders and commands
  - Control via pause/step/resume
  - Memory and register access
  - LV2 kernel object introspection
- Different debug interfaces:
  - **RPCS3** Built-in Qt-based tools
  - **Nucleus** Web frontend for a custom protocol (*Nerve*)
- Get proper host debugging tools!
  e.g. *VS debugger*, *RenderDoc*, *NSight*, etc.



Fin

- This concludes our tour through physical and virtual PS3 consoles.
- Pending features:
  - Full PSN-like online functionality.
  - Support for USB peripherals.
  - Firmware-less emulation via HLE.
  - Support for ARMv8+ hosts.
  - *Improve debuggers/tests.*

- Closing thoughts and wishes:
  - Aspiring emudevs, aim for the endgame:
    If you want to emulate *X*, forget about NES.
    Go for *X*. Both will be hard, and you will need every ounce of motivation.
  - Don't be afraid of starting your emulator.
    Cross-pollination grows devs and scenes.
    - In FOSS, alternatives are not competitors.
    - Don't be afraid of stopping it either.
      Escalation of commitment does more harm than good.
  - Read emulator sources; follow commits.
    - Tastes develop when exposed to variety.
    - Hidden gems are beneath every emulator.
      - Nucleus contains traces of 8~10 others.

Thank you!

Questions?

RPCS3 Founders

DH, HYKEM

RPCS3 Current Staff/Maintainers/Developers

NEKOTEKINA, KD-11, SSSHADOW, ANI, BLACKDAEMON, MEGAMOUSE, ELAD, GALCIV, DAGINATSUKO, WHATCOOKIE, HCORION, SILENT, HERRHULAHOOP, VELOCITY, JOHNHOLMESII, TGE, CLIENTHAX, YAHFZ, JUHN 0XCA7F00D, ASININE

RPCS3 Former Staff/Maintainers/Developers

TAMBRY, BIGPET, VLJ, GOPALSR83, JARVES, NUMAN, FLASH-FIRE, DANGLES, PAULS-GH, O1L, RUIPIN, SCRIBAM, RAVEN02, RAJKOSTO, FARSEER AND *YOURS TRULY* (ALEXALTEA)

PS3 Hackers/Developers

FAIL0VERFLOW (MARCAN, BUSHING, SVEN, SEGHER), GEOHOT, FLATZ, 3141CARD, XORLOSER, MATHIEULH, PHIRENZ, KAKAROTO, CFWPROHET, OCT0XOR, EUSS, NAEHRWERT, DEROAD, BL4STY, ADA, GRAF_CHOKOLO, SKFU, KDSBEST, ALDO, ROGERO, GLEVAND, ZECOXAO, RICHDEVX, EVILSPERM, AERIALX, RANCIDO, ROXANNE, TIZZYT, SCOGNITO, ANDOMA, HERMES, PS3HAX/GREG, JEVINSKIE, JUANNADIE

And everybody who remained anonymous/missed above...

EOF

Trademarks and Copyright

All trademarks and copyrighted content found on these presentation slides belong to their respective owners. The RPCS3 and Nucleus teams are in no way affiliated with Sony or PlayStation.

The "*PlayStation logo*", "*PlayStation 3 logo*", "*PlayStation 4 logo*", "*PlayStation Portable logo*", "*PlayStation Vita logo*", "*PlayStation Move logo*", "*PlayStation Network logo*", "*PlayStation Store logo*", "*PlayStation Plus logo*", "*Sony logo*", "*Sony Computer Entertainment logo*" and their aforementioned names are registered trademarks of Sony Computer Entertainment Inc.
"*Sony Entertainment Network*" is a trademark of Sony Corporation. Library programs are copyrighted by Sony Interactive Entertainment Inc.

Intellectual Properties

This presentation displays various trademarks and copyrighted content such as videogame logos, company logos, videogame names and company names. These different trademarks and copyrighted content do not belong to the author of this presentation and are properties of their respective owners. This presentation is in no way affiliated with said companies or trademarks. This presentation's only intention is to spread the word and endorse the content displayed.

Statement on Piracy and Illegal Acts

When dumping video game software, users are subject to country-specific software distribution laws. RPCS3 and Nucleus are not designed to enable illegal activity. Their developers do not promote piracy nor do they allow it under any circumstances.

The goal of RPCS3 and Nucleus is to experiment, research, and educate on the topic of PlayStation 3 emulation that can be performed on compatible devices and operating systems. All information was obtained legally by purchasing PlayStation 3 hardware and software. Additional information was obtained from various sources on the Internet that include but is not limited to system hardware and software documentation. Most of this information can be found on the PlayStation 3 Developer Wiki.