SPU wrangling

job management and debugging

jonathan garrett

jonny@insomniacgames.com

GDC 2009

introduction

- SPU system management at Insomniac
  - job-manager (PPU and GPU jobs)
  - job-manager debugging help
  - SPU debugging in general
  - debugging case studies

job-manager

- basic job-manager (small - less than 2k)
- just loads jobs onto specific SPUs
- launched at game start
  - hogs SPU – doesn’t yield
  - overwrites anything residing on SPU from init
  - most of local-store available

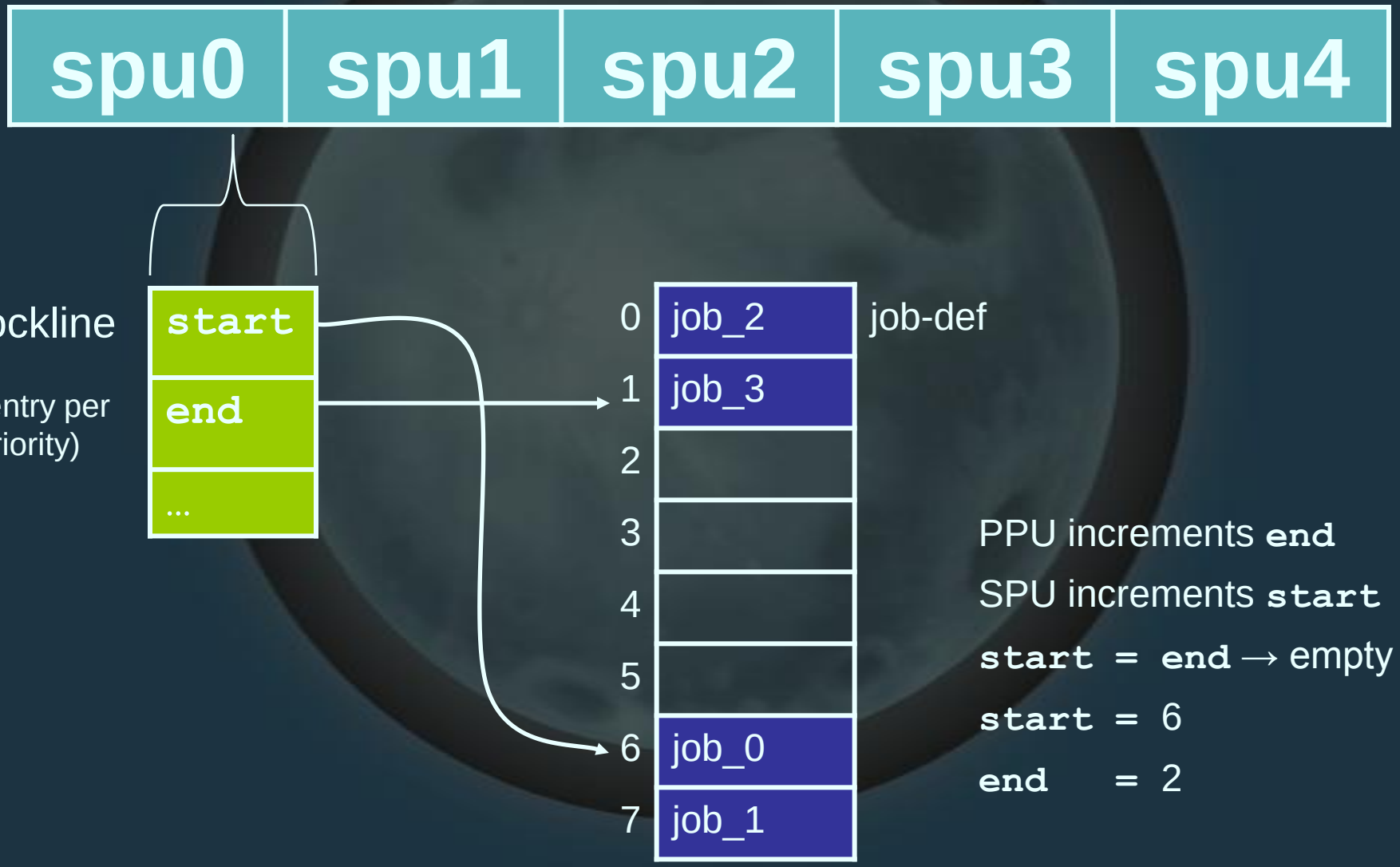
job-manager

- jobs have large granularity (whole system)
  - sub-job management up to individual system
  - load-balancing up to individual system
- we only use a couple of middleware modules
  - libs – driver code is in our system

job-manager

- jobs processed in submission-order
  - simple 3-level priority scheme
- ring-buffer per SPU
  - PPU adds a job to a specific queue
- support for GPU triggered jobs
  - PPU adds job to highest priority queue
  - GPU triggers - sync primitive to hold up
- uses lockline to avoid busy-waiting

job-manager – job-list



PPU increments **`end`**

SPU increments **`start`**

`start = end` → empty

`start =` 6

`end =` 2

job-manager job-triggering

- busy waiting:

```
while(1)
{
   dma in ring_buffer_end from PPU
   sync dma

   if (ring_buffer_start != ring_buffer_end)
   {
     break;  // we have new job
   }

   delay();  ← spin
}

// grab job at ring_buffer_start

ring_buffer_start = ((ring_buffer_start + 1) & ring_buffer_size_mask);
dma out ring_buffer_start to PPU

// process job
```

job-manager job-triggering

- busy waiting:

```
while(1)
{
   dma in ring_buffer_end from PPU
   sync dma

   if (ring_buffer_start != ring_buffer_end)
   {
     break;  // we have new job
   }

   delay();  ← spin
}

// grab job at ring_buffer_start

ring_buffer_start = ((ring_buffer_start + 1) & ring_buffer_size_mask);
dma out ring_buffer_start to PPU

// process job
```

job-manager job-triggering

- busy waiting:

```
while(1)
{
   dma in ring_buffer_end from PPU
   sync dma

   if (ring_buffer_start != ring_buffer_end)
   {
     break;  // we have new job
   }

   delay();  ← spin
}

// grab job at ring_buffer_start

ring_buffer_start = ((ring_buffer_start + 1) & ring_buffer_size_mask);
dma out ring_buffer_start to PPU

// process job
```

job-manager job-triggering

- busy waiting:

```
while(1)
{
   dma in ring_buffer_end from PPU
   sync dma

   if (ring_buffer_start != ring_buffer_end)
   {
     break;  // we have new job
   }

   delay();   ← spin
}
```

```
// grab job at ring_buffer_start

ring_buffer_start = ((ring_buffer_start + 1) & ring_buffer_size_mask);
dma out ring_buffer_start to PPU

// process job
```

job-manager job-triggering

- lockline waiting
  - dma 128-bytes (ring-buffer) from PPU and make a *reservation* on that address
  - process new job as appropriate
  - wait for reservation-lost event
    - SPU blocks on **`rdch`**
    - *sleeps* until PPU / GPU writes to reserved address
  - avoids repeated bus access

job-manager job-triggering

- lockline waiting:

```
while(1)
{
   dma_llar ring_buffer_end from PPU
   sync dma_llar

   if (ring_buffer_start != ring_buffer_end)
   {
     break;  // we have new job
   }

   wait_reservation_lost(); ← block till written (PPU / GPU write)
}

// grab job at ring_buffer_start

ring_buffer_start = ((ring_buffer_start + 1) & ring_buffer_size_mask);
dma out ring_buffer_start to PPU

// process job
```

job-manager job-triggering

- lockline waiting:

```
while(1)
{
   dma_llar ring_buffer_end from PPU
   sync dma_llar

   if (ring_buffer_start != ring_buffer_end)
   {
     break;  // we have new job
   }

   wait_reservation_lost(); ← block till written (PPU / GPU write)
}

// grab job at ring_buffer_start

ring_buffer_start = ((ring_buffer_start + 1) & ring_buffer_size_mask);
dma out ring_buffer_start to PPU

// process job
```

job-manager job-triggering

- lockline waiting:

```
while(1)
{
   dma_llar ring_buffer_end from PPU
   sync dma_llar

   if (ring_buffer_start != ring_buffer_end)
   {
     break;  // we have new job
   }

   wait_reservation_lost(); ← block till written (PPU / GPU write)
}

// grab job at ring_buffer_start

ring_buffer_start = ((ring_buffer_start + 1) & ring_buffer_size_mask);
dma out ring_buffer_start to PPU

// process job
```

job-manager – GPU interaction

- GPU renders frame-deferred
- SPU job issued in update-frame – runs during next frame
- GPU can write 16-bytes to main memory
- triggers SPU to process new job
- SPU job end triggers GPU semaphore to continue

job-manager – GPU interaction

job-manager – GPU interaction

PPU	update 0	…	…	…	…
GPU	…	render 0	…	…	…
TV	…	…	display 0	…	…
SPU	…	assist 0	…	…	…

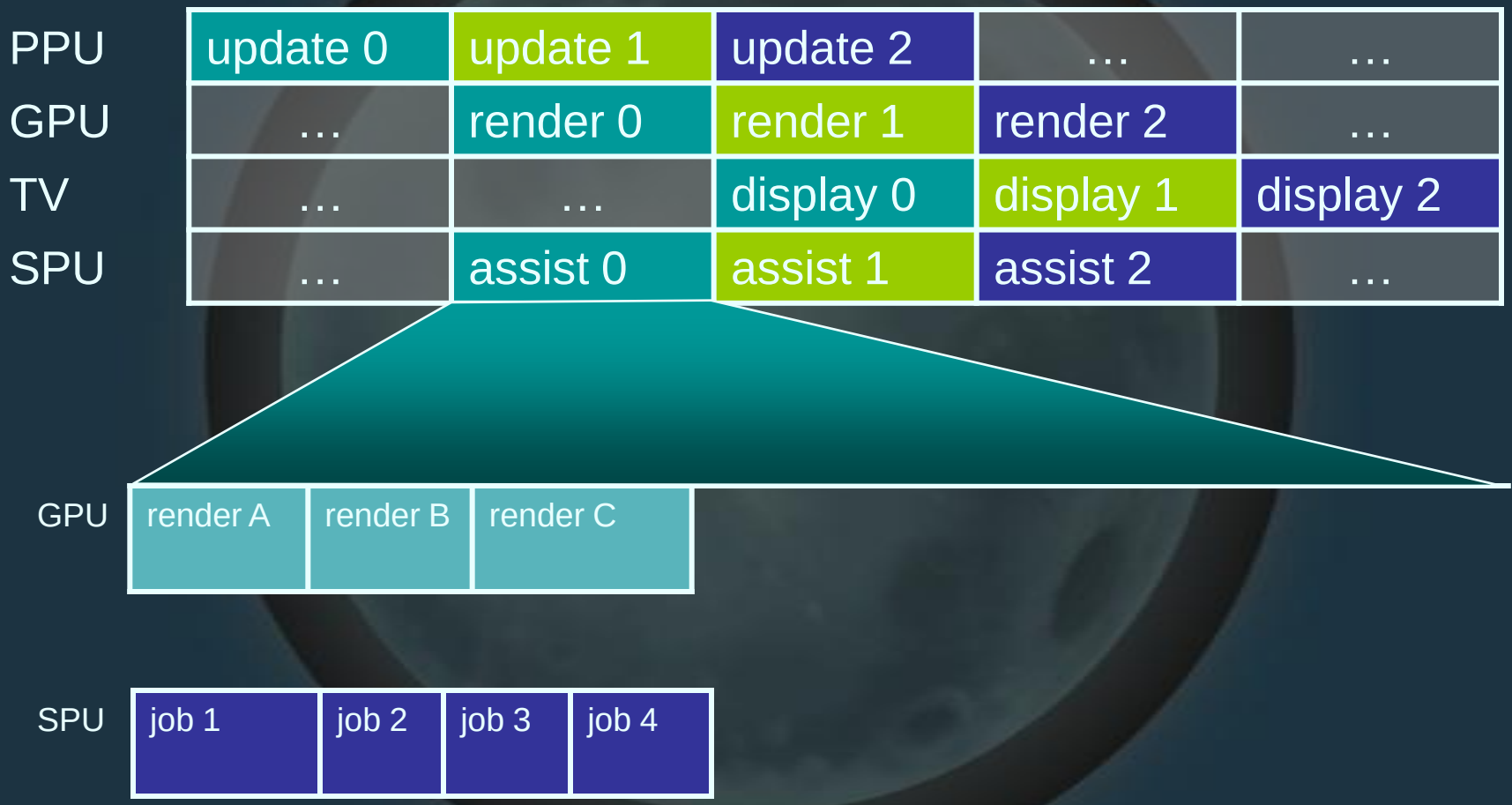
job-manager – GPU interaction

PPU	update 0	update 1	…	…	…
GPU	…	render 0	render 1	…	…
TV	…	…	display 0	display 1	…
SPU	…	assist 0	assist 1	…	…

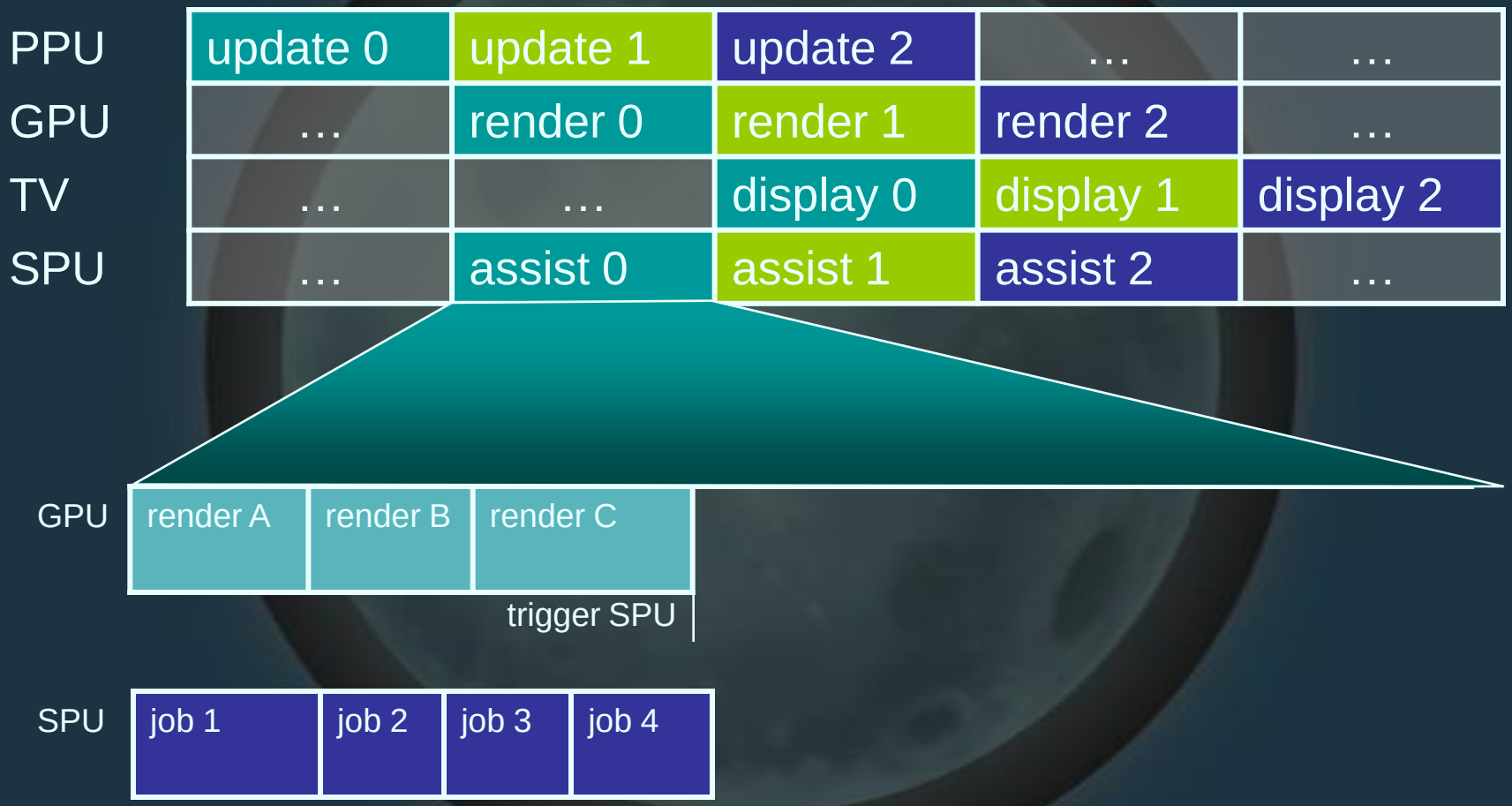
job-manager – GPU interaction

PPU	update 0	update 1	update 2	…	…
GPU	…	render 0	render 1	render 2	…
TV	…	…	display 0	display 1	display 2
SPU	…	assist 0	assist 1	assist 2	…

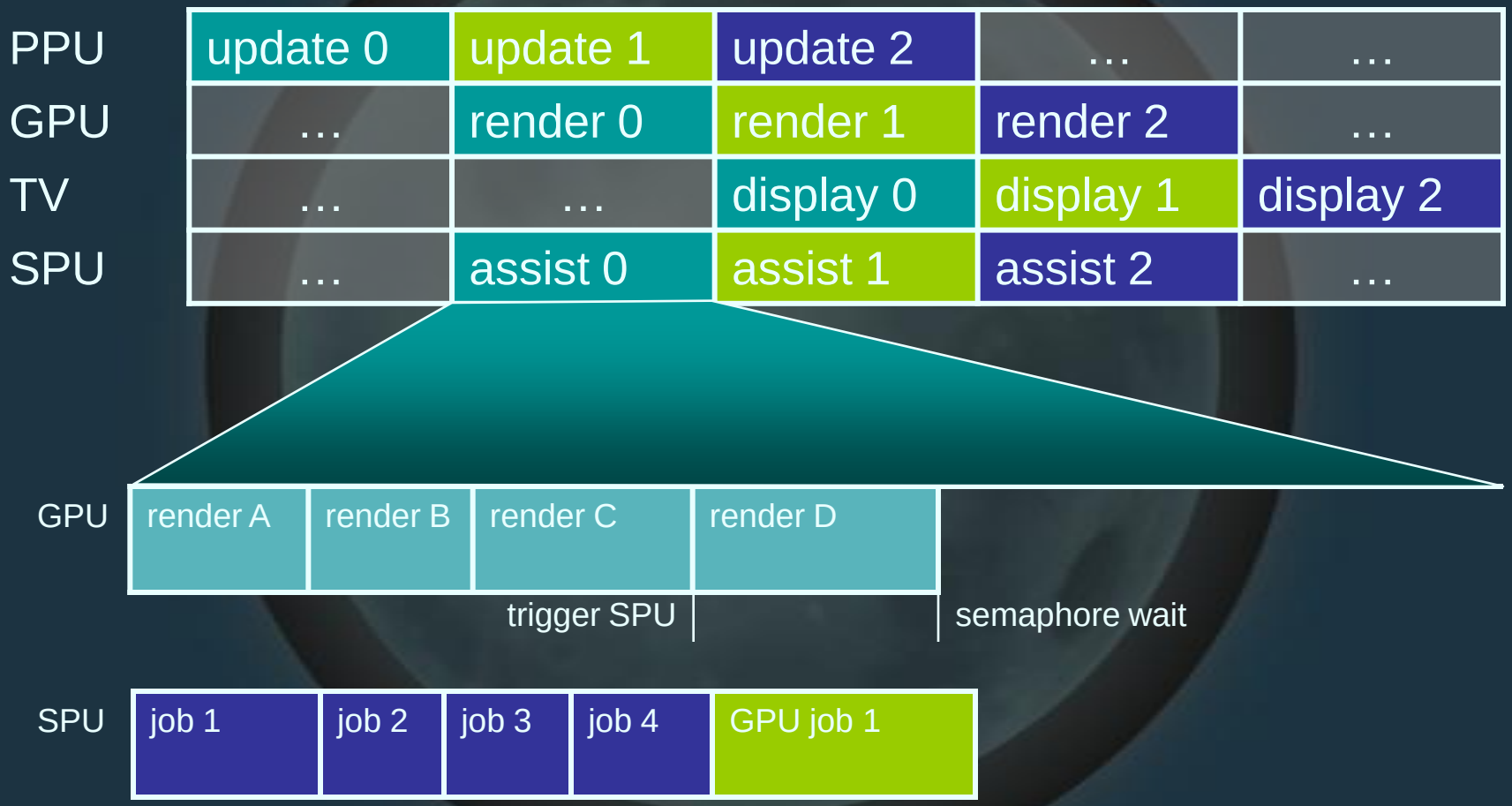
job-manager – GPU interaction



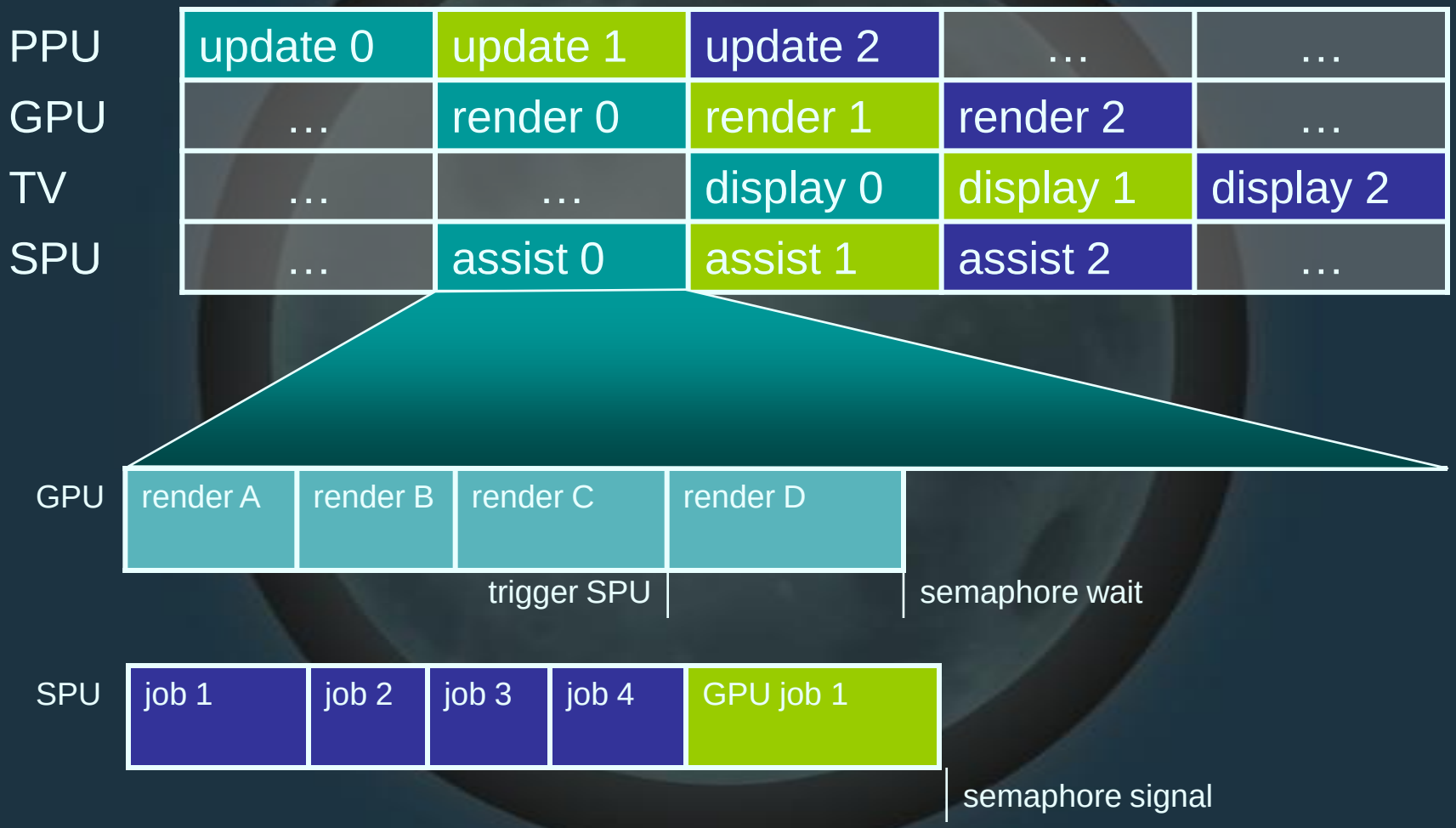
job-manager – GPU interaction



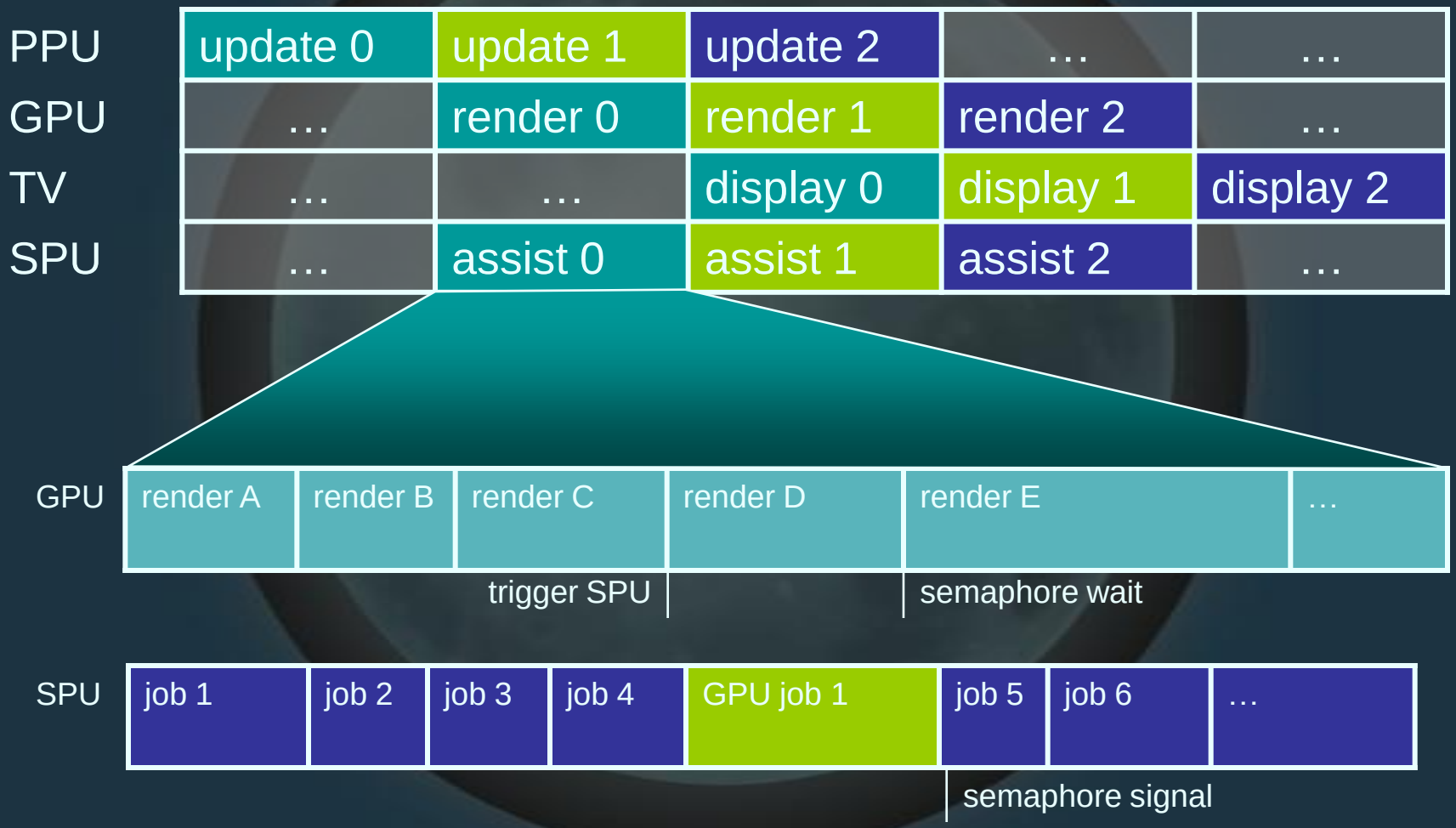
job-manager – GPU interaction



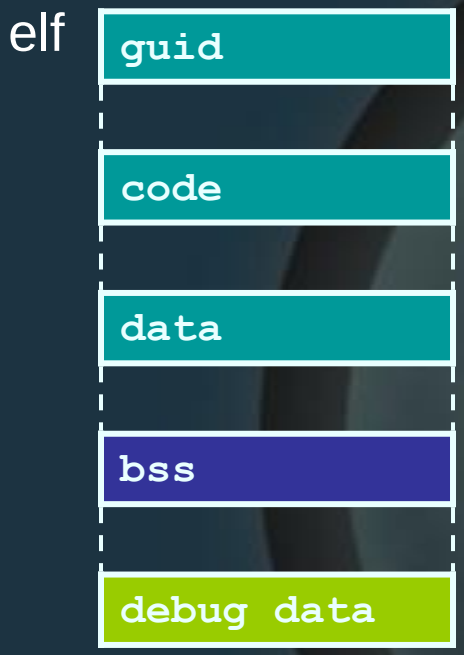
job-manager – GPU interaction



job-manager – GPU interaction

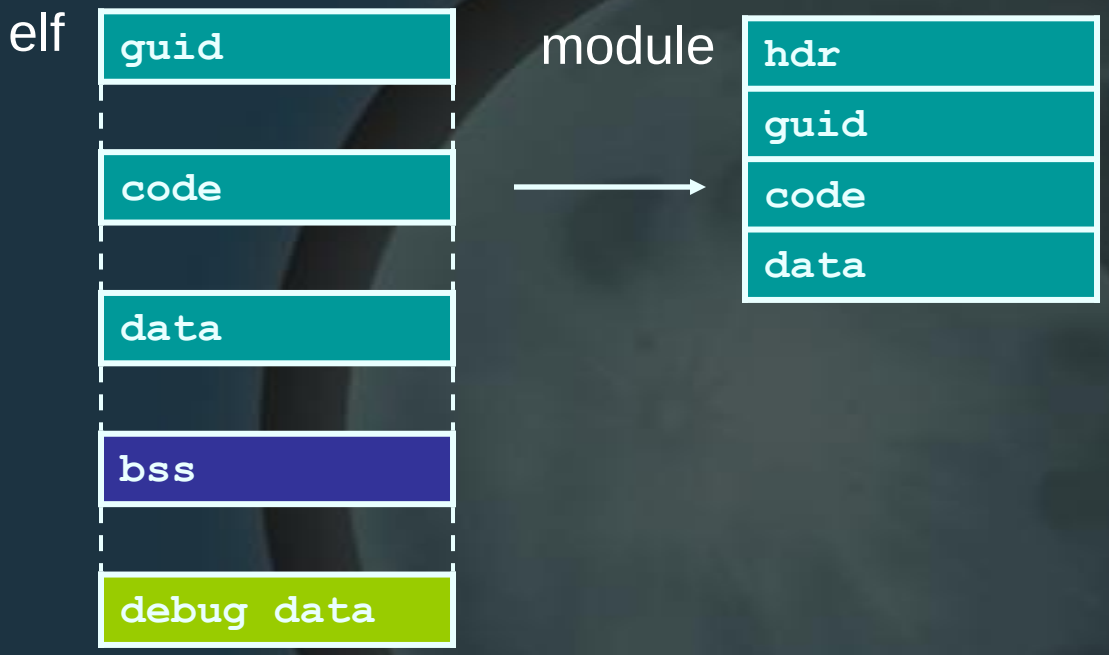


SPU elf



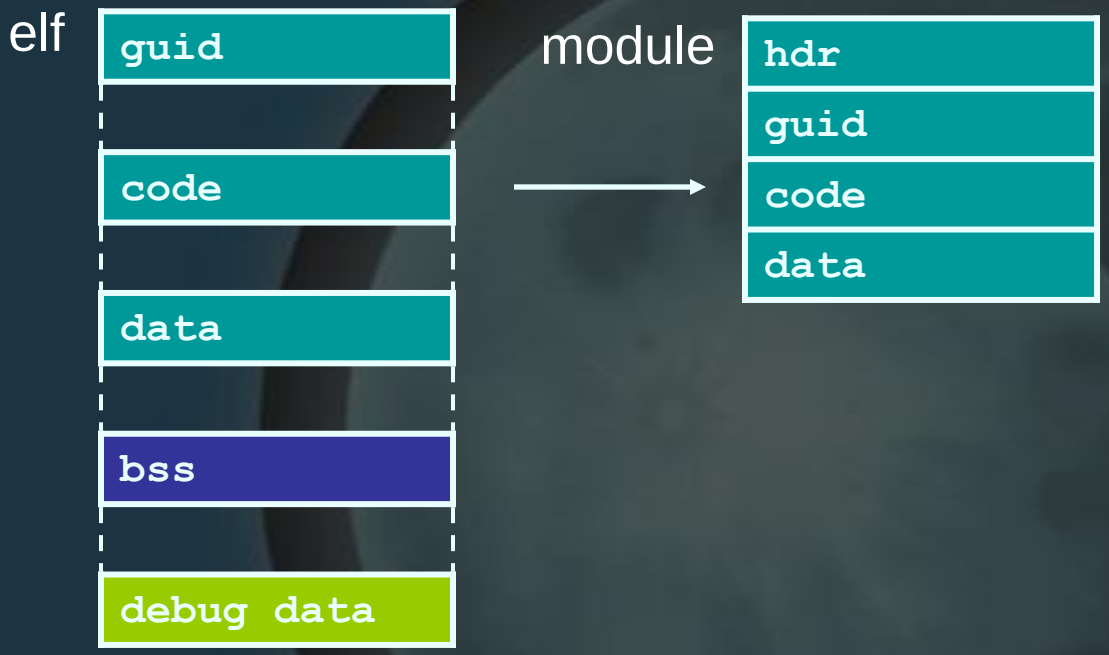
- linker outputs SPU elf (used by debugger)
- `spu-objcopy` to stripped block
- `ppu-objcopy` to output PPU linkable OBJ
- loader dmas block and clears bss

SPU elf



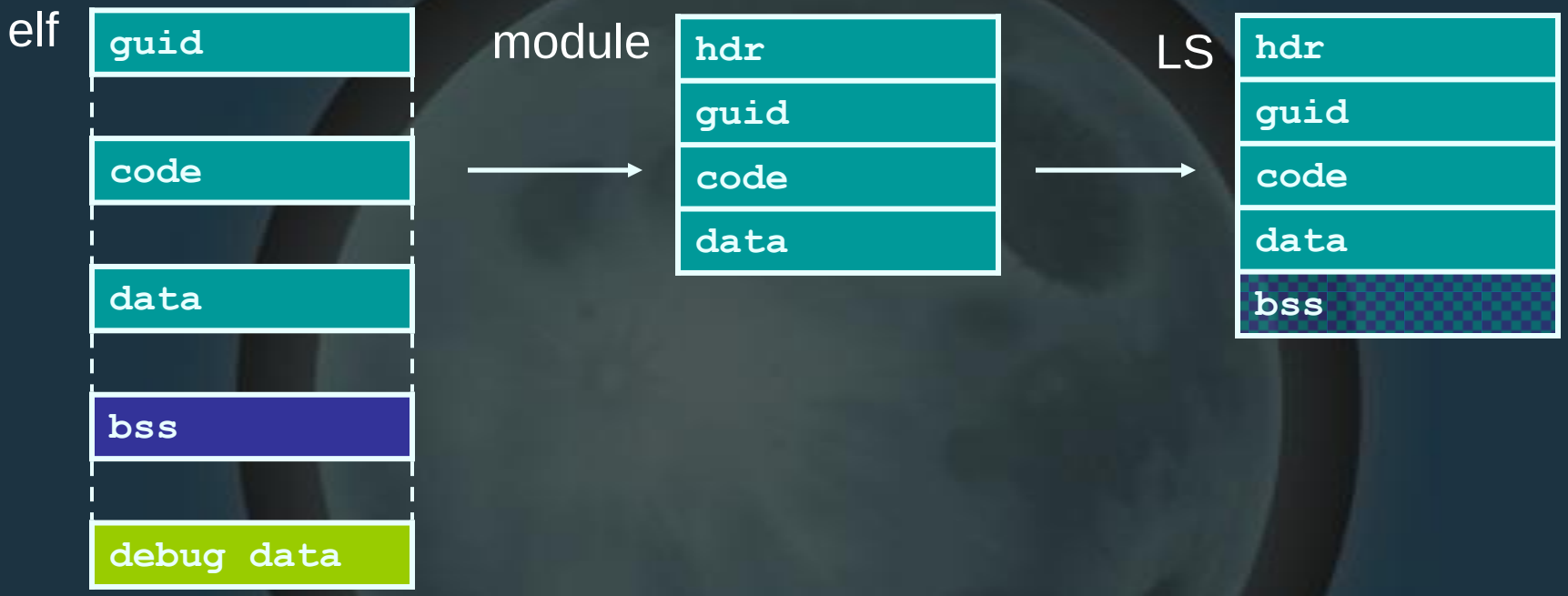
- linker outputs SPU elf (used by debugger)
- `spu-objcopy` to stripped block
- `ppu-objcopy` to output PPU linkable OBJ
- loader dmas block and clears bss

SPU elf



- linker outputs SPU elf (used by debugger)
- `spu-objcopy` to stripped block
- `ppu-objcopy` to output PPU linkable OBJ
- loader dmas block and clears bss

SPU elf



- linker outputs SPU elf (used by debugger)
- `spu-objcopy` to stripped block
- `ppu-objcopy` to output PPU linkable OBJ
- loader dmas block and clears bss

job-manager

- lives at top 2k of memory
  - jobs have LS access from `0x100` → `0x3f800`
- standard DMA calls can be used
  - free use of DMA tags - system uses 30 and 31
  - jobs can use these but may stall on a system DMA
- system handles job *start* / *stop* profile trace
  - api allows jobs to emit more trace packets (shaders)
  - support for profiler-tool trace visualization

job-manager

- user-specified stack-size
- or defaults to be from module-end to loader-start
- remaining LS used as a work-buffer
  - api to query base/size

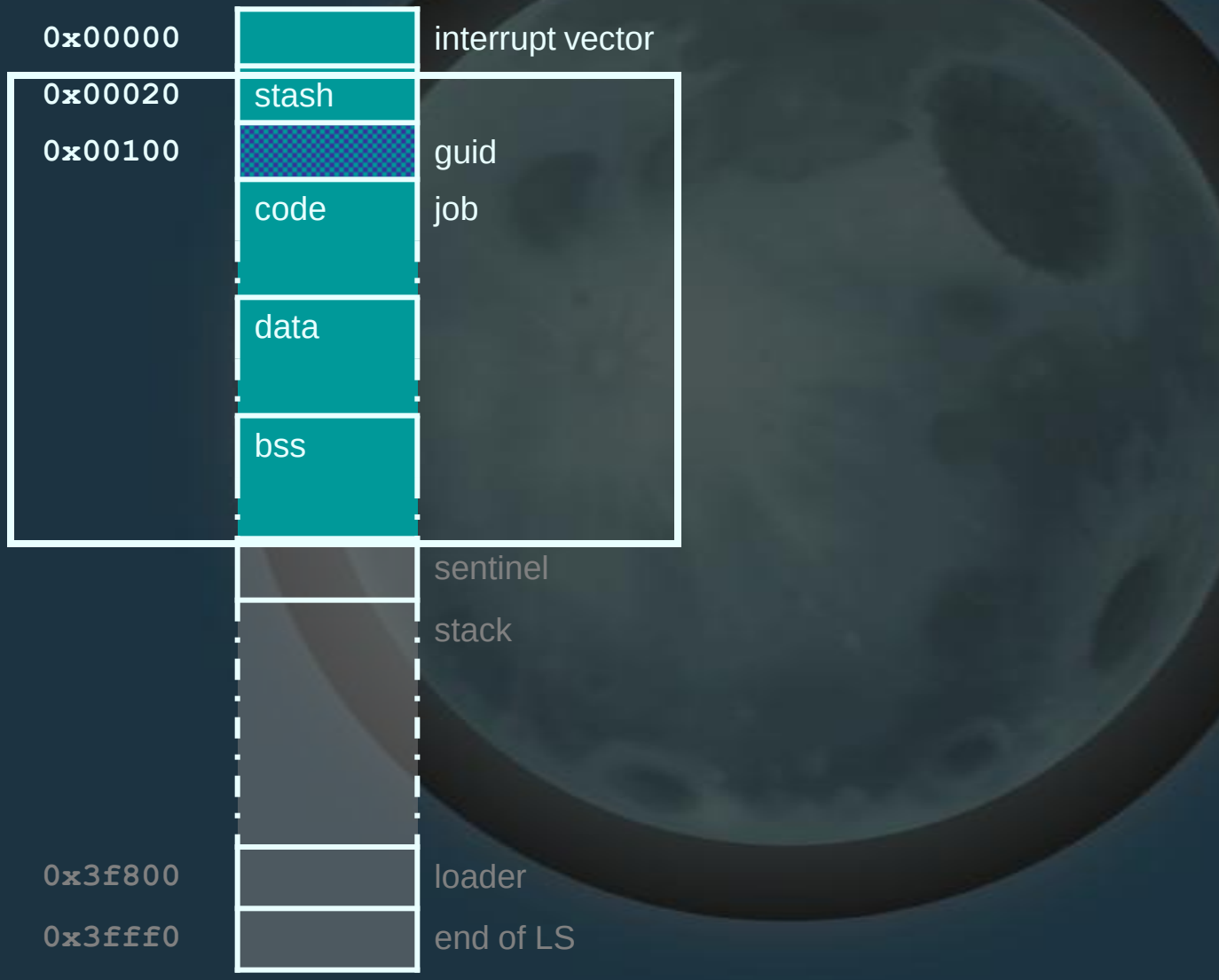
job-def

- **job-def** struct defines a job

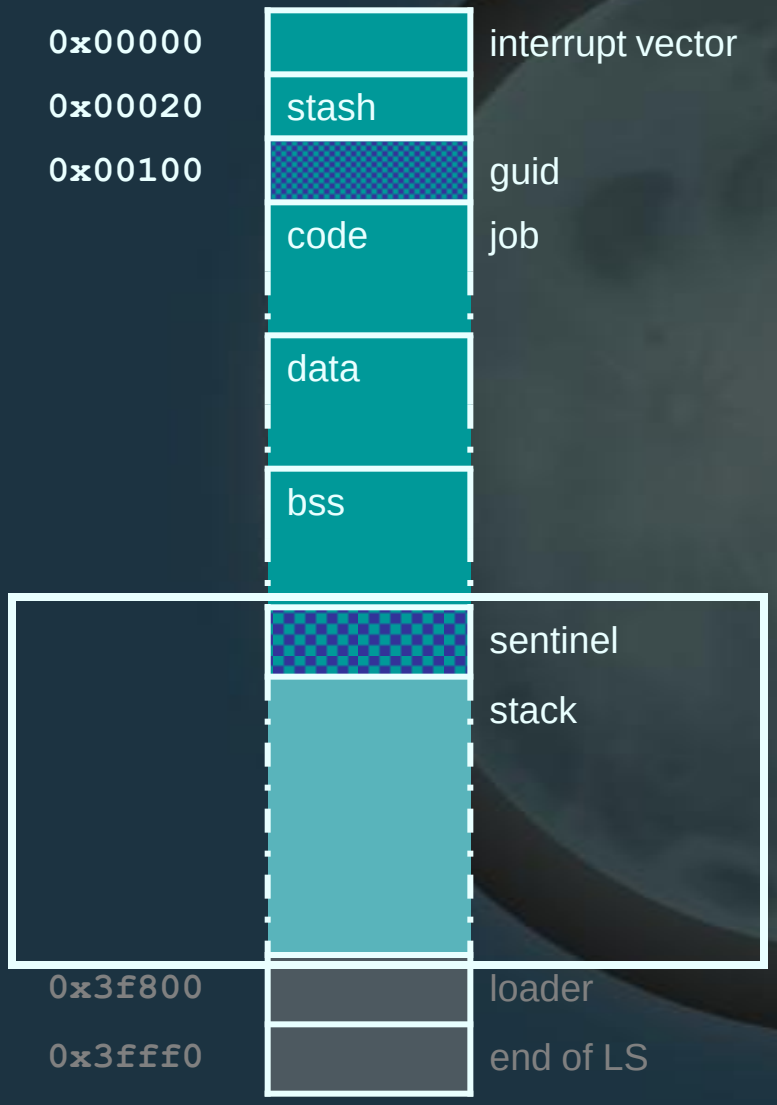
```
u16     m_ls;          //                    LS load address
u16     m_size;        //                    total LS size
u16     m_dma_size;    // dma_size >> 2      dma LS size
u16     m_bss;         // start bss >> 2     bss LS address
u16     m_bss_size;    // bss size >> 2      bss size
u16     m_entry;       // entry >> 2         module entry
u16     m_stack_size;  // stack_size >> 2    stack size (or 0)
u16     m_flags;       //                    misc flags
qword   m_params_a;    //                    JobMain parameters
qword   m_params_b;    //
```

- first block from SPU elf hdr (custom link-script)
- second block at runtime

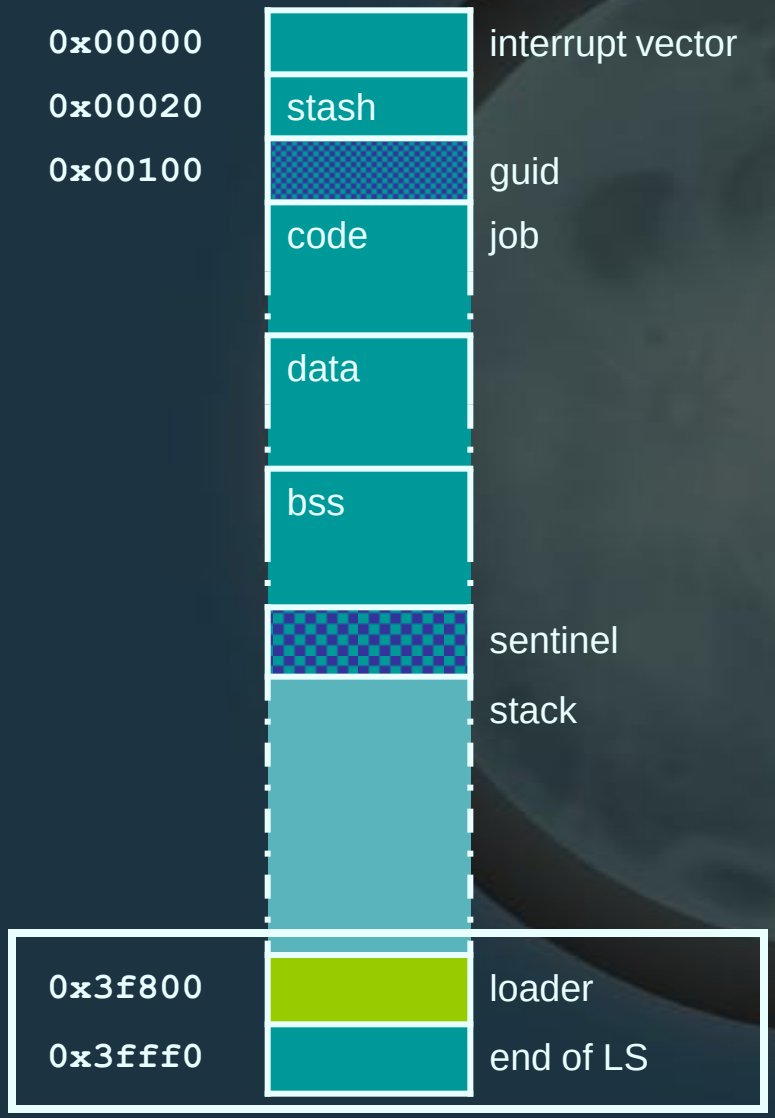
job-manager – LS layout



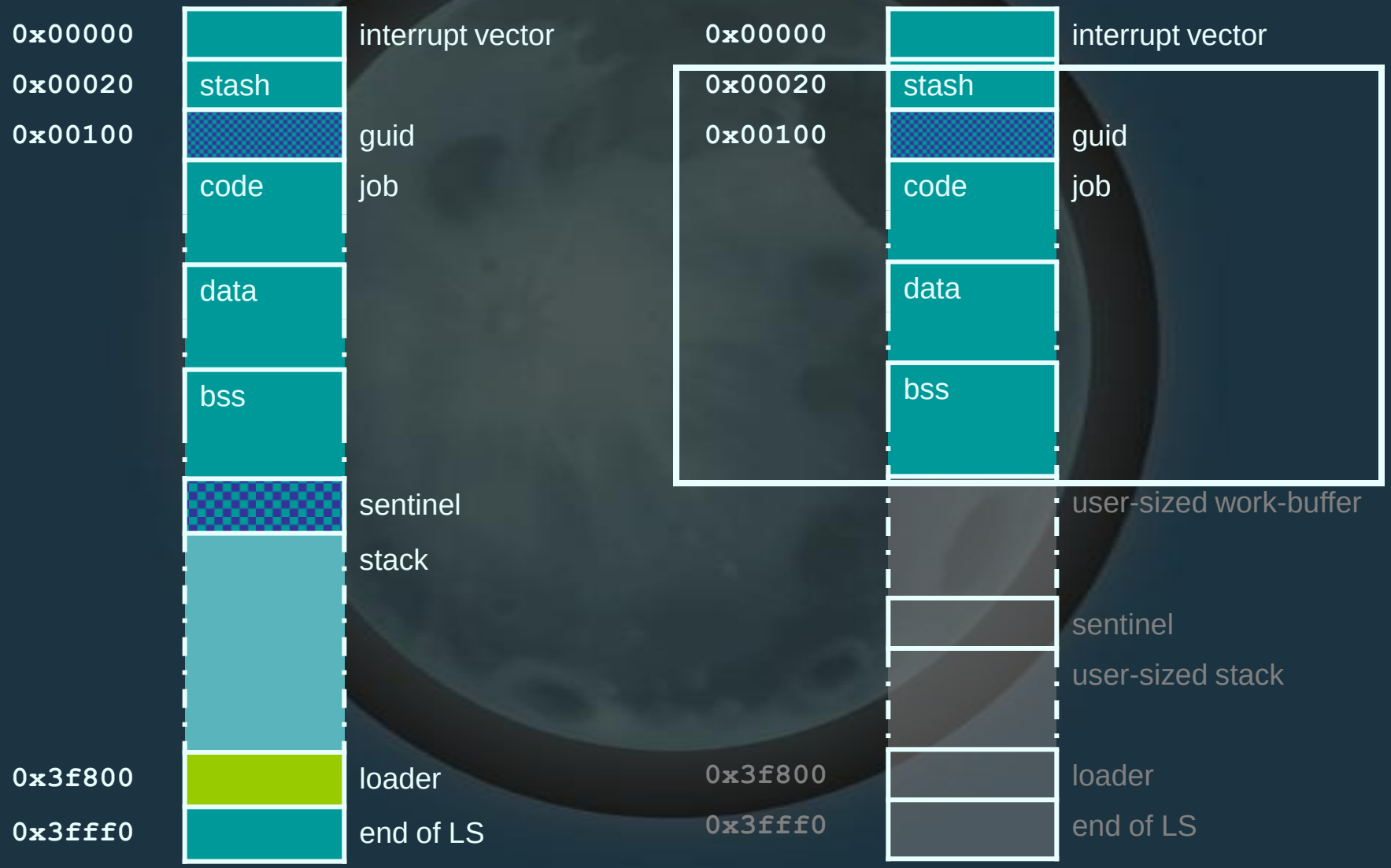
job-manager – LS layout



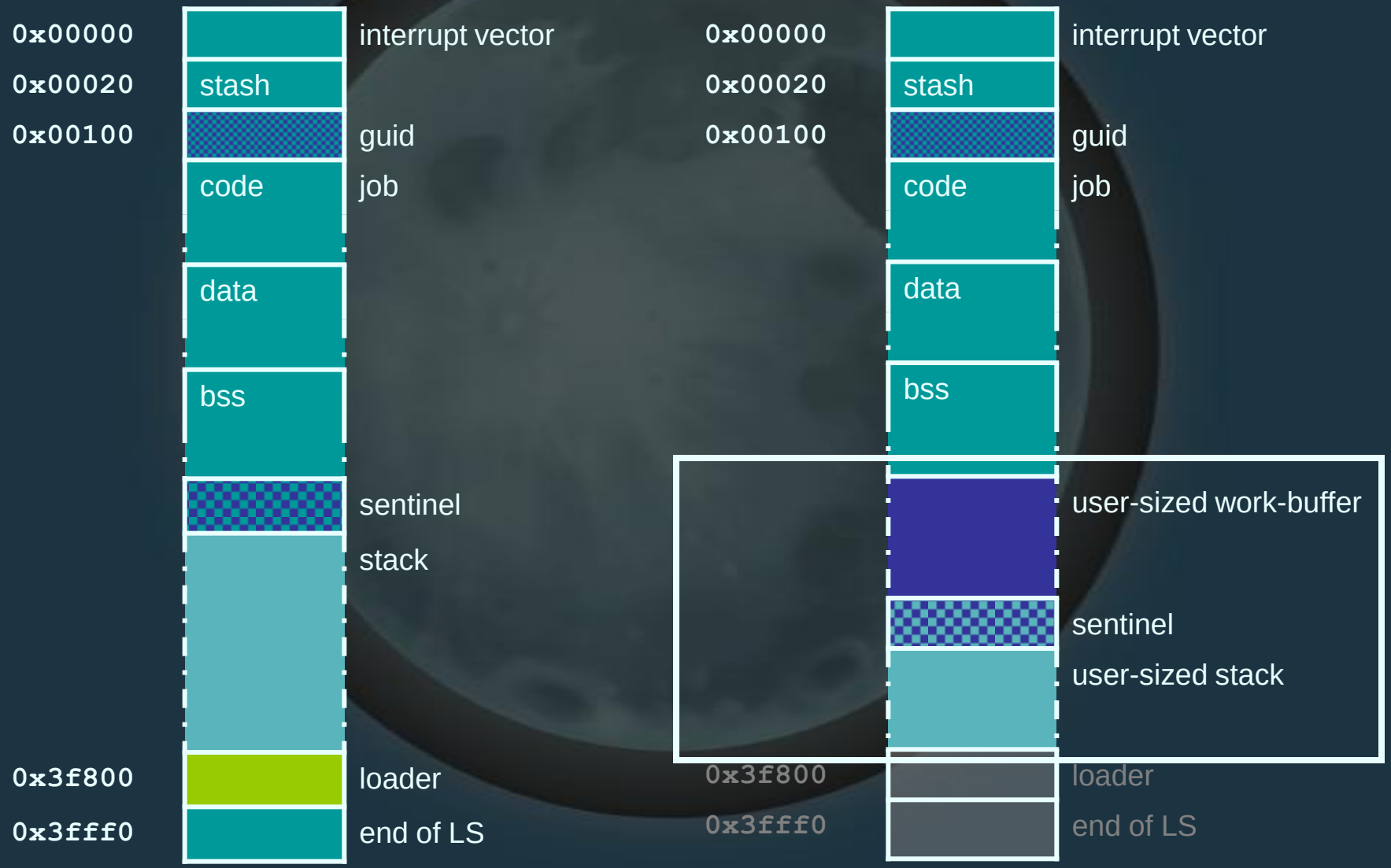
job-manager – LS layout



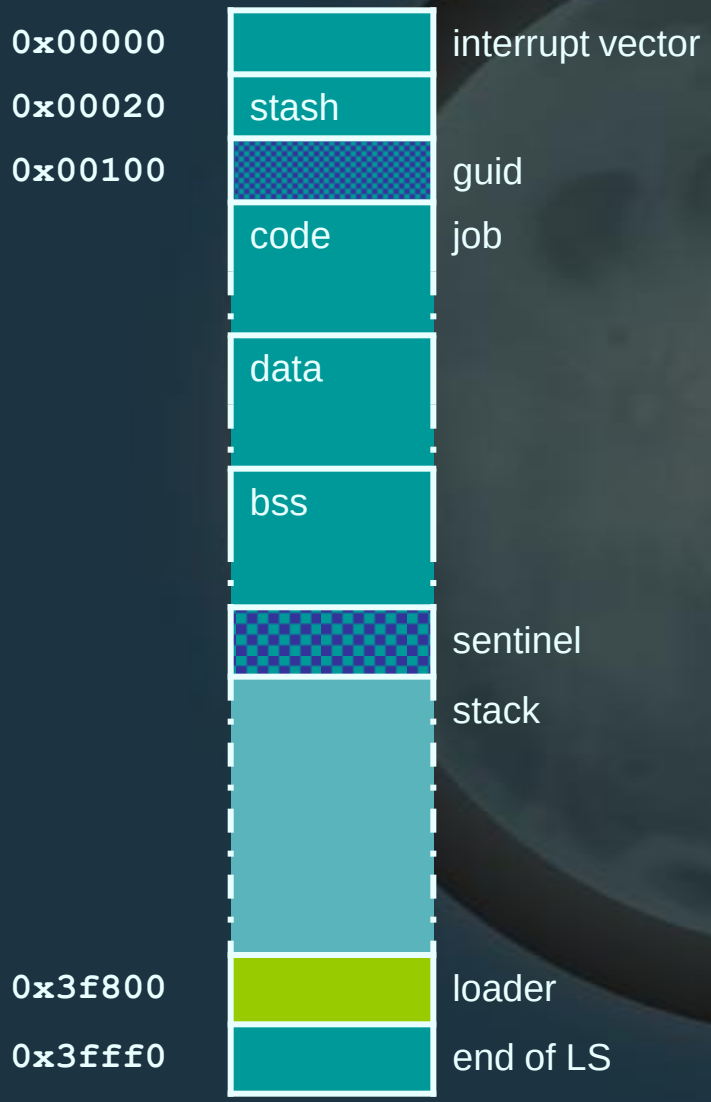
job-manager – LS layout

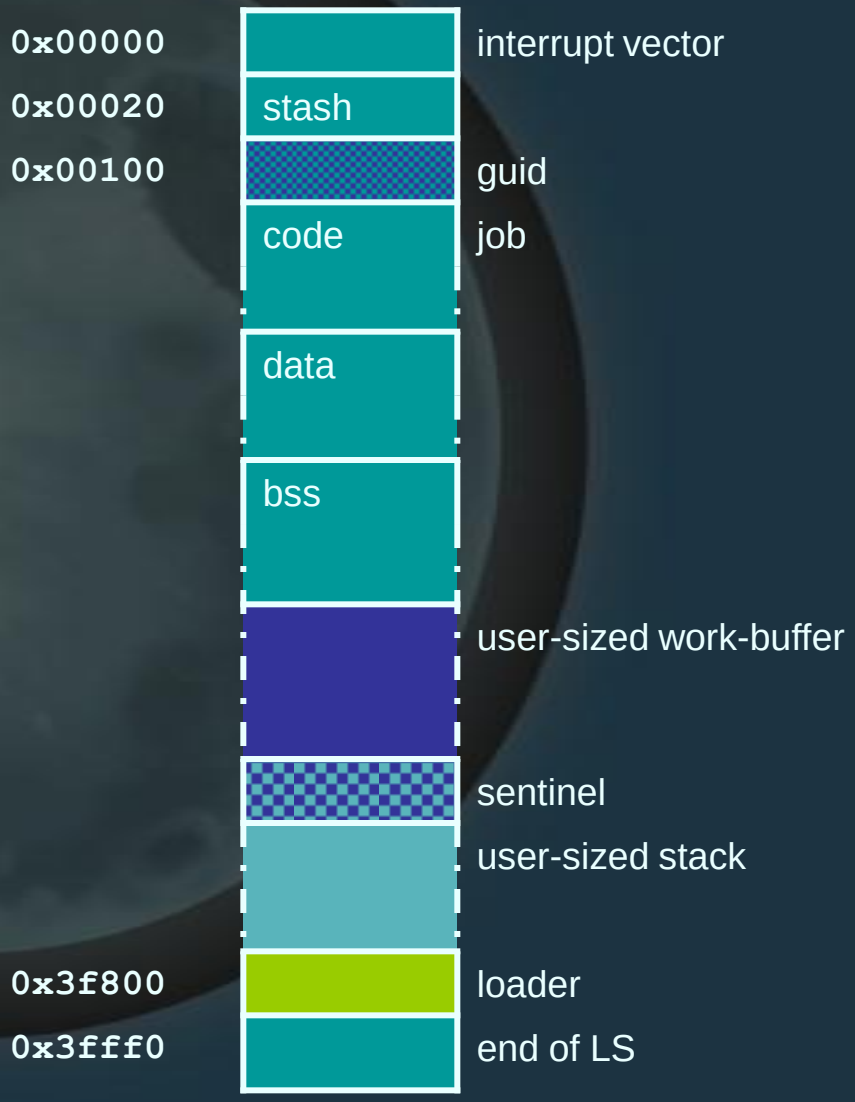


job-manager – LS layout



job-manager – LS layout





debugging support

- low memory used as a stash for job-related / debug data

`0x00020`	–	job-def
`0x00030`	–	job params-a
`0x00040`	–	job params-b
`0x00050`	–	stack-top
`0x00060`	–	stack-bottom
`0x00070`	–	LS-buffer-size
`0x00080 – 0x000f0`	–	user-debug stash

debugging support

- `job-def` flags allow breaking just before entry to `JobMain`
- can also break just before calls to “constructors” / “destructors”
  - step through asm to debug each one
- interrupts off - interrupt vector overwritten with `0x0000dead`
  - `stop 0x1ead` to trap jumps to `0x00000`

debugging support

- system stashes sentinel to stack-end
  - asserts it's intact on job-exit
- DMA wrappers validate calls
  - alignment, size, etc. – compiles out in FINAL
  - stashes arguments to globals

    `g_DEBUG_DmaEa`, `g_DEBUG_DmaLs`, `g_DEBUG_DmaSize`
  - software break (put / get – EA / LS addr)

    `g_DEBUG_DmaGetBreakLs`,

    `g_DEBUG_DmaGetBreakEa` etc.

timeouts

- PPU watchdog ensures SPU job completed within reasonable timeframe
  - either through frame or next frame
- dumps job-queues – shows which jobs have run and which have yet to run
- dumps *user-debug stash*
- helpful info from QA

asserts

- standard assert performs **`print`** and then **`stop`**s
  - print interrupts PPU – does the real work
  - SPU stacks args and issues mailbox-interrupt
  - we also stash **`SP`** and **`LR`**
- PPU identifies standard print vs assert print
  - assert walks stack using **`SP`**, **`LR`**
  - calls user-handler with debug-stash
    - eg. anim handler dumps specific stash entries (**`stash[0].m_u32[0]`** = moby ptr)
  - continuable from PPU (visual assert)
    - SPU waits on mailbox read – returns whether continue or stop

asserts

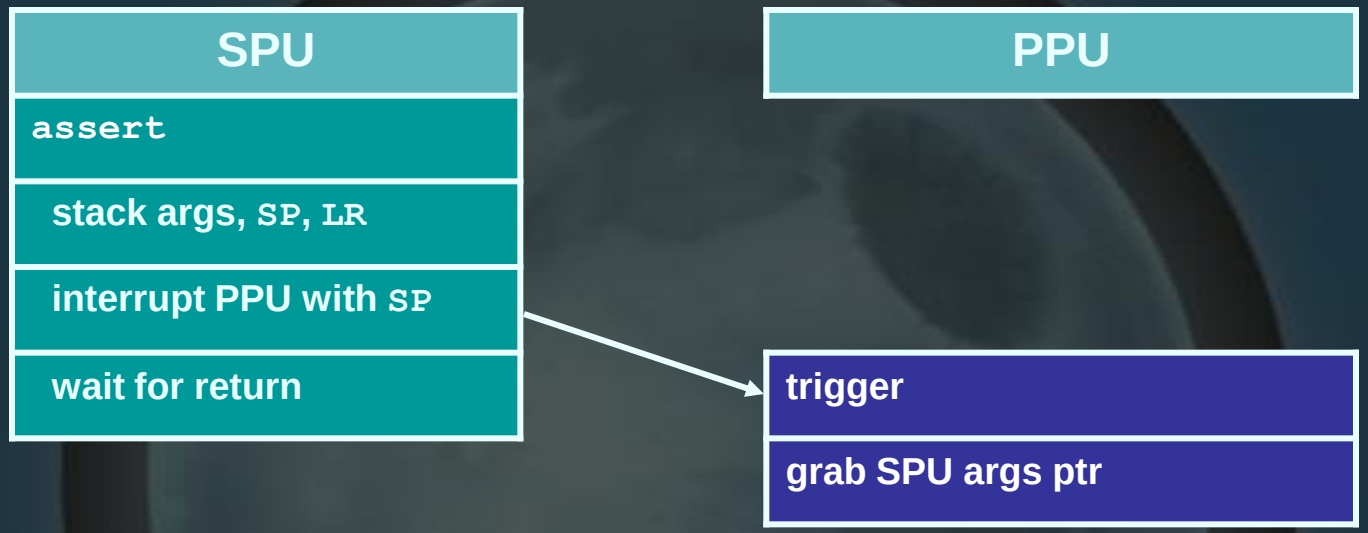
asserts

SPU
`assert`
stack args, `SP`, `LR`

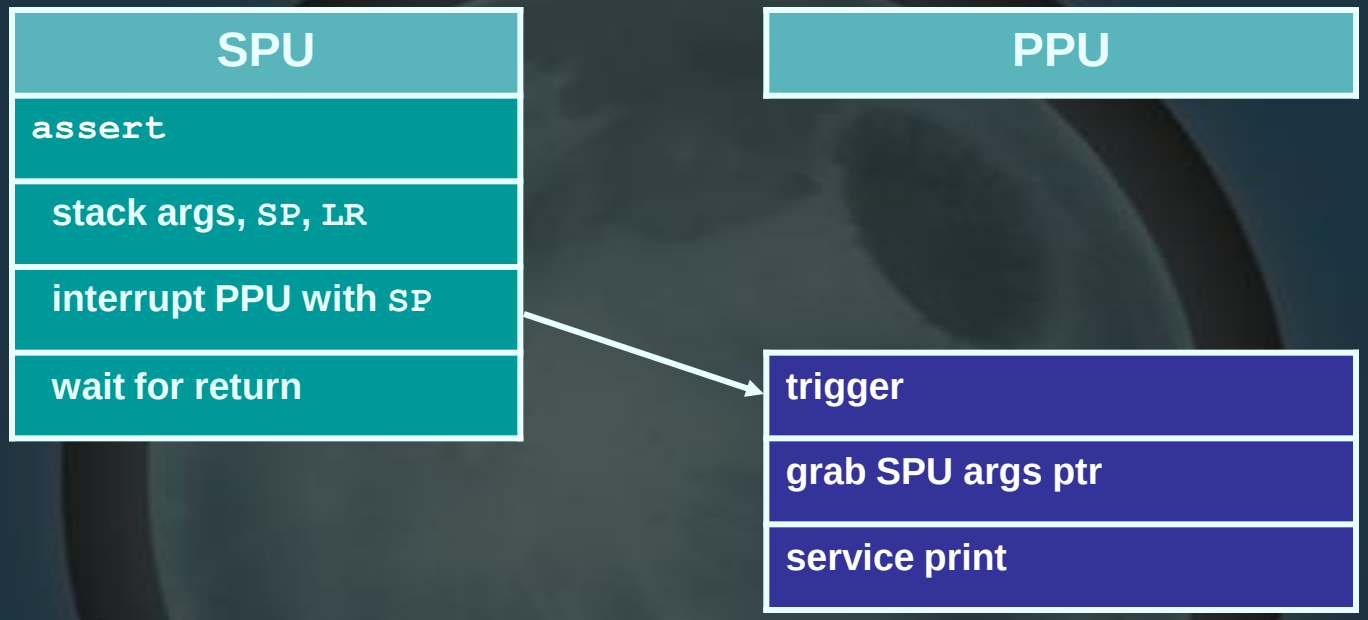
asserts

SPU
`assert`
stack args, `SP`, `LR`
interrupt PPU with `SP`

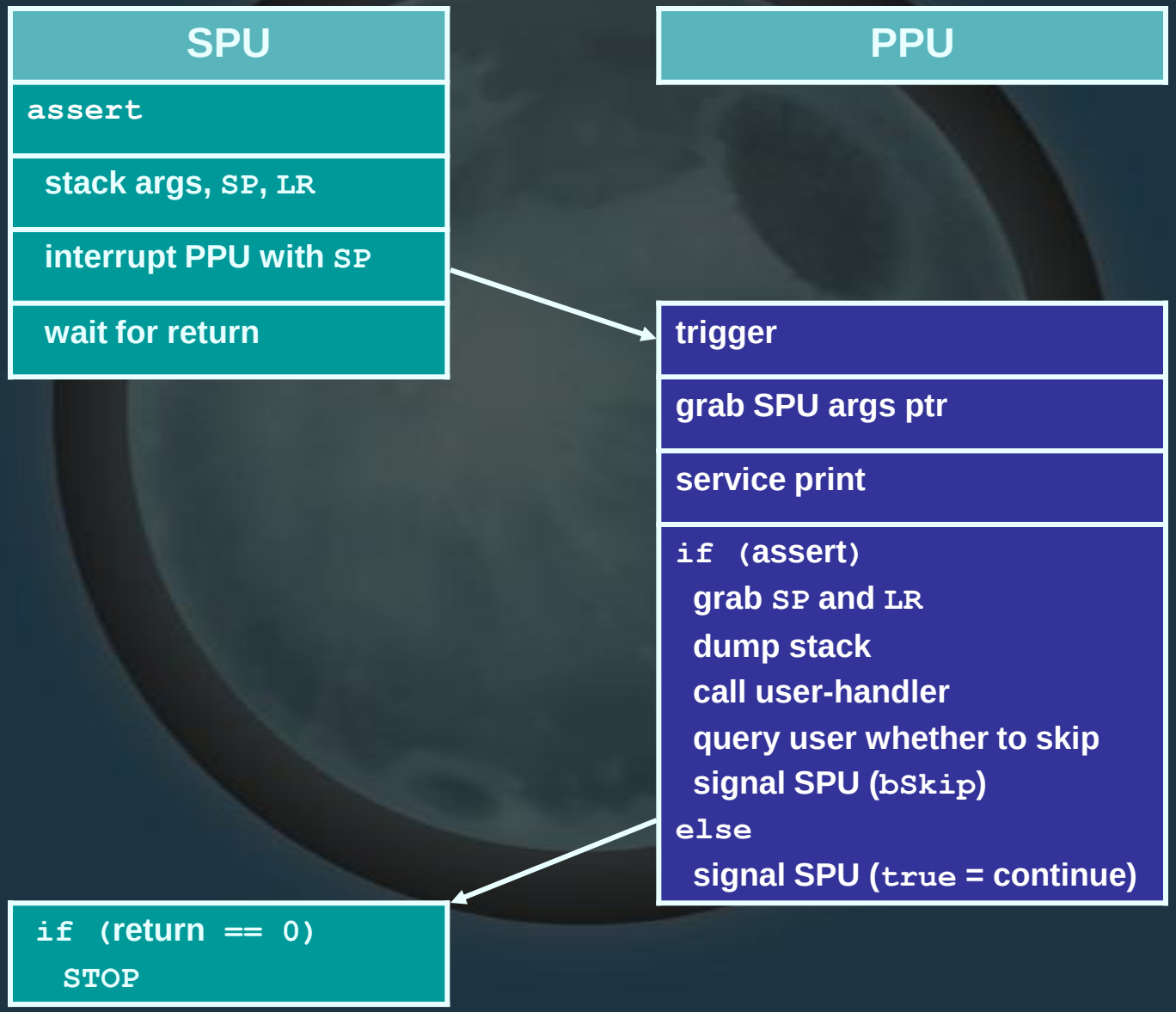
asserts


SPU
PPU
assert
stack args, SP, LR
interrupt PPU with SP
wait for return
trigger
grab SPU args ptr

asserts


SPU
assert
stack args, SP, LR
interrupt PPU with SP
wait for return
PPU
trigger
grab SPU args ptr
service print

asserts


SPU
assert
stack args, SP, LR
interrupt PPU with SP
wait for return
PPU
trigger
grab SPU args ptr
service print
if (assert)
grab SP and LR
dump stack
call user-handler
query user whether to skip
signal SPU (bSkip)
else
signal SPU (true = continue)
if (return == 0)
STOP

asserts

- asserts add bloating debug-only code
- standard assert (with / without skip-check):

```
#define IG_ASSERT(x_)
  if EXPECT_FALSE(!(x_))
  {
    if (ASSERT_ERR("ASSERT FAILED [expr='%s', %s (line %d)]",
                   #x_, __FILE__, __LINE__))
    {
      IG_STOP();
    }
  }
```

```
#define IG_ASSERT(x_)
  if EXPECT_FALSE(!(x_))
  {
    ASSERT_ERR("ASSERT FAILED [expr='%s', %s (line %d)]",
               #x_, __FILE__, __LINE__));
    IG_STOP();
  }
```

asserts

- asserts add bloating debug-only code
- standard assert (with / without skip-check):

```
#define IG_ASSERT(x_)
  if EXPECT_FALSE(!(x_))
  {
    if (ASSERT_ERR("ASSERT FAILED [expr='%s', %s (line %d)]",
                   #x_, __FILE__, __LINE__))
    {
      IG_STOP();
    }
  }
```

```
#define IG_ASSERT(x_)
  if EXPECT_FALSE(!(x_))
  {
    ASSERT_ERR("ASSERT FAILED [expr='%s', %s (line %d)]",
               #x_, __FILE__, __LINE__));
    IG_STOP();
  }
```

asserts

```
u32 func_a(u32 val_)
{
  IG_ASSERT((val_ & 0xf) == 0);
  return 123;
}
```

```
00000038 <_Z5func_aj>:
  38: andi    $2,$3,15                    $2 = (val_ & 0xf)
  3c: stqd    $0,16($1)
  40: ila     $4,0
  44: stqd    $1,-32($1)
  48: ila     $3,0
  4c: ai      $1,$1,-32
  50: brnz    $2,64                       branch if ((val_ & 0xf) != 0)
  54: ai      $1,$1,32
  58: il      $3,123                      load return value 123
  5c: lqd     $0,16($1)
  60: bi      $0                          return
  64: ila     $5,0
  68: il      $6,142
  6c: brsl    $0,0                        call print
  70: stopd   $0,$1,$1                    stop!
  74: br      54                          branch back to return
```

asserts

```
u32 func_a(u32 val_)
{
  IG_ASSERT((val_ & 0xf) == 0);
  return 123;
}
```

```
00000038 <_Z5func_aj>:
  38: andi    $2,$3,15                    $2 = (val_ & 0xf)
  3c: stqd    $0,16($1)
  40: ila     $4,0
  44: stqd    $1,-32($1)
  48: ila     $3,0
  4c: ai      $1,$1,-32
  50: brnz    $2,64                       branch if ((val_ & 0xf) != 0)
  54: ai      $1,$1,32
  58: il      $3,123                      load return value 123
  5c: lqd     $0,16($1)
  60: bi      $0                          return
  64: ila     $5,0
  68: il      $6,142
  6c: brsl    $0,0                        call print
  70: stopd   $0,$1,$1                    stop!
  74: br      54                          branch back to return
```

asserts

```
u32 func_a(u32 val_)
{
  IG_ASSERT((val_ & 0xf) == 0);
  return 123;
}
```

```
00000038 <_Z5func_aj>:
  38: andi    $2,$3,15                    $2 = (val_ & 0xf)
  3c: stqd    $0,16($1)
  40: ila     $4,0
  44: stqd    $1,-32($1)
  48: ila     $3,0
  4c: ai      $1,$1,-32
  50: brnz    $2,64                       branch if ((val  & 0xf) != 0)
  54: ai      $1,$1,32
  58: il      $3,123                      load return value 123
  5c: lqd     $0,16($1)
  60: bi      $0                          return
  64: ila     $5,0
  68: il      $6,142
  6c: brsl    $0,0                        call print
  70: stopd   $0,$1,$1                    stop!
  74: br      54                          branch back to return
```

asserts

```
u32 func_a(u32 val_)
{
  IG_ASSERT((val_ & 0xf) == 0);
  return 123;
}
```

```
00000038 <_Z5func_aj>:
  38: andi    $2,$3,15                    $2 = (val_ & 0xf)
  3c: stqd    $0,16($1)
  40: ila     $4,0
  44: stqd    $1,-32($1)
  48: ila     $3,0
  4c: ai      $1,$1,-32
  50: brnz    $2,64                       branch if ((val_ & 0xf) != 0)
  54: ai      $1,$1,32
  58: il      $3,123                      load return value 123
  5c: lqd     $0,16($1)
  60: bi      $0                          return
  64: ila     $5,0
  68: il      $6,142
  6c: brsl    $0,0                        call print
  70: stopd   $0,$1,$1                    stop!
  74: br      54                          branch back to return
```

asserts

- smaller asserts:

```
#define IG_ASSERT_FAST(cond_)    spu_hcmpeq((u32)(cond_), 0U)
```

```
u32 func_b(u32 val_)
{
  IG_ASSERT_FAST((val_ & 0xf) == 0);
  return 123;
}
```

```
000000c8 <_Z6func_bj>:
  c8: andi    $4,$3,15          $2 = (val_ & 0xf)
  cc: ceqi    $2,$4,0           (val_ & 0xf) == 0 ?
  d0: sfi     $3,$2,0           negate
  d4: heqi    $0,$3,0           halt if ((val & 0xf) != 0)
  d8: il      $3,123            load return value 123
  dc: bi      $0                return
```

asserts

- smaller asserts:

```
#define IG_ASSERT_FAST(cond_)    spu_hcmpeq((u32)(cond_), 0U)
```

```
u32 func_b(u32 val_)
{
  IG_ASSERT_FAST((val_ & 0xf) == 0);
  return 123;
}
```

```
000000c8 <_Z6func_bj>:
  c8: andi    $4,$3,15          $2 = (val_ & 0xf)
  cc: ceqi    $2,$4,0           (val_ & 0xf) == 0 ?
  d0: sfi     $3,$2,0           negate
  d4: heqi    $0,$3,0           halt if ((val & 0xf) != 0)
  d8: il      $3,123            load return value 123
  dc: bi      $0                return
```

asserts

- smaller asserts:

```
#define IG_ASSERT_FAST(cond_)    spu_hcmpeq((u32)(cond_), 0U)
```

```
u32 func_b(u32 val_)
{
  IG_ASSERT_FAST((val_ & 0xf) == 0);
  return 123;
}
```

```
000000c8 <_Z6func_bj>:
  c8: andi    $4,$3,15          $2 = (val_ & 0xf)
  cc: ceqi    $2,$4,0           (val_ & 0xf) == 0 ?
  d0: sfi     $3,$2,0           negate
  d4: heqi    $0,$3,0           halt if ((val & 0xf) != 0)
  d8: il      $3,123            load return value 123
  dc: bi      $0                return
```

asserts

- careful with **`halt`**
  - non-exact (PC stops a few instructions later)
  - can’t be continued
- another variant:

```
#define IG_ASSERT_ID(cond, id_) if (!((u32)cond_))
                                    asm ("stopd 0, "#id_", "#id_);
```

- both versions reduce code-bloat in debug and release builds

exceptions

- own exception-handler
  - runs after system handler
  - dumps any relevant data
    - state of job-queues
    - user-debug stash
    - working on stack walk
  - calls user-function with debug-stash
    - module specific – knows what to expect in debug-stash
      - anim: `stash[0] = moby ea`
- output added to QA reports

general SPU debugging

- SPU debugging strategies we use
- nothing special, but might be useful

general SPU debugging

- complications:
  - not sure when our module will run
  - not sure which SPU it'll run on
  - not sure when the thing we're interested in will be processed
  - have helpers in place
    - compile out in RELEASE / FINAL
    - but also need to debug FINAL

detour - abi

- often need to debug at the asm level
- very useful – don't be intimidated
- the more asm you know, the more sense it'll make
  - SDK / IBM docs have everything you need
- ABI defines register usage (including how parameters passed between functions)

`r0`	link register (**`LR`**)
`r1`	stack ptr (**`SP`**)
`r2`	volatile (caller save)
`r3`	volatile (caller save) – first function argument (or **`this`**) and return
`r4 – r79`	volatile (caller save) – next 76 function arguments
`r80 – r127`	non-volatile (callee save) - locals

general SPU debugging

- simplify (always the key!)
  - disable unrelated code
  - break on entry to our module / shader
  - change to run on a single SPU
  - embed break-flag in element struct
- embed debug info in your SPU structs
  - AnimStack - `m_pad` = ea of moby
  - Collision job – debug-sequence counter
  - ptr to locally allocated buffers in LS

general SPU debugging

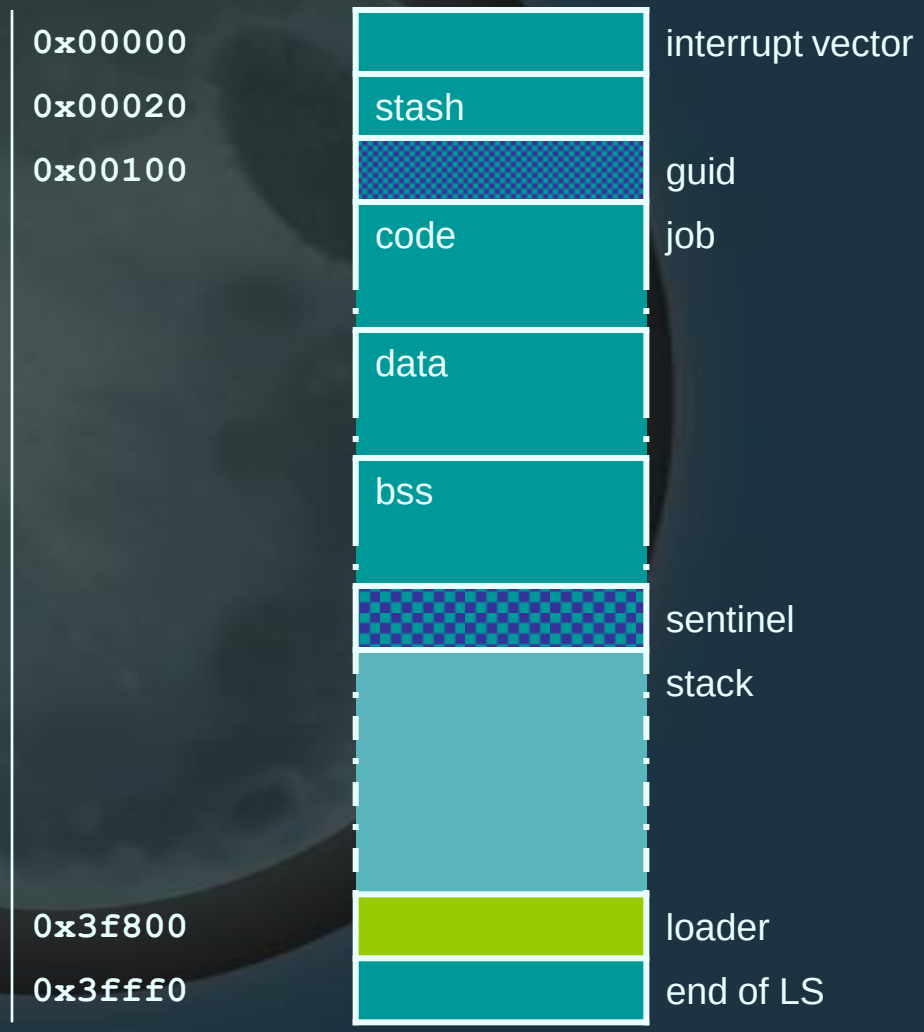
- PPU timeout fired – only a few causes:
  - jump to NULL (= `stop 0x1ead`)
  - “infinite” loop – try stopping the SPU
  - `stop` – from assert / manually placed `IG_STOP`
    - if assert, TTY might show output
  - bad DMA LS (wait on bad DMA)
  - `readch` – blocking wait for DMA complete
    - which ? – debugger help
  - other blocking `readch` (deadlock ?)
  - illegal instruction ? – memory stomp

general SPU debugging

- initial assumption is that the timeout system's module stopped
  - callstack / TTY should show which one
  - misleading - depends on submit vs sync order
  - debugger can tell us currently active module

general memory layout

- bottom = 0x00000
- top = 0x40000
- interrupt vector at 0x00000
- module layout:
  - code
  - data
  - bss
  - stack



general memory layout

- no protection - easy to trash memory
  - can write over bss, then data, then code
  - bad ptr can trash anything
  - bad dma-LS can trash anything
- memory wraps
  - can trash from low to high, all the way to `0x3fff0` and wrap to `0x00000`
  - or backwards and wrap in at top
- so many opportunities!

general memory layout

- stack grows from high address to low address
- frames *allocated* / *freed* on function entry / exit
  - function return address (**`LR`**) stored in caller's frame
  - stack frame created (**`SP`** updated)
  - function executed
  - return address loaded to **`LR`**
  - return

debugging - stack check

- compile option: `-fstack-check`
- standard SP:

  `r001` = `0x033f0` | `0x033f0` | `0x033f0` | `0x033f0`

- with `-fstack-check`:

  `r001` = `0x03ff0` | `0x04c00` | `0x03ff0` | `0x03ff0`

  `0x4c00` = 19k of stack free

- prolog: `r001.y -= stack_frame_size`
  `halt if r001.y -ve`
- epilog: `r001.y += stack_frame_size`

- useful - but if you're tight on space, the act of turning it on can push you over the edge!

trace

- trace all flow through your module
  - loads of output, but can be very useful
- TRACE macro (compiles out)

```
#ifdef DO_TRACE
  #define TRACE(fmt_, args_...) \
    Printf("\x1b[34m"                 \      ← blue
           "TRC_SPU{anim}: "          \      ← identify our module
           fmt_                       \
           "\x1b[30m", ##args_);             ← default color
#else
  #define TRACE(fmt, args...)
#endif
```

desperation

- `printf` – old friend! – surprisingly effective
  - if affects timing can dma back to PPU ringbuffer
- stash to persistent LS (debug stash)
  - 128 bytes from `0x00080` – `0x000f0` available:

```
*(u32*)0x00080 = my_id;
```

  - dumped to TTY by timeout code – helpful info in QA reports

debugging - stack walk

- walking SPU stack is easy:
  - `$001` (`SP`) points to most recent stack frame
  - view memory at `$001` as 4 columns of words
  - SP points to current stack-frame
  - 1st word at SP points to next stack-frame
  - 1st word at SP+16 points to function return
  - remember: return is actually stored in *parent's*-frame before current-frame created

stack walk

```
03E580 | 0003E620 00002CA0 0003E620 0003E620  link ↩
03E590 | 000060BC 00000000 00000000 00000000  return = 0x060bc
03E5A0 | 00000000 00000000 00000000 00000000  Upd_SimEmitterParticles
...                                            0060B8 brsl r000, Upd_SimBatch

03E620 | 0003E7A0 00002E20 0003E7A0 0003E7A0  link ↩
03E630 | 0000CC98 00000000 00000000 00000000  return = 0x0cc98
03E640 | 01E04000 005D0000 01000000 00000000  Upd_SimStep
...                                            00CC94 brsl r000,Upd_SimParticles

03E7A0 | 0003E8D0 00002F50 0003E8D0 0003E8D0  link ↩
03E7B0 | 00017364 00000000 00000000 00000000  return = 0x17364
03E7C0 | 40002400 00000000 00000000 00000000  Upd_RunProcessList
...                                            017360 brsl r000,Upd_SimStep

03E8D0 | 0003E910 00002F90 0003E910 0003E910  link ↩
03E8E0 | 0000E618 00000000 00000000 00000000  return = 0x0E618
03E8F0 | 4057C080 DEADDEAD 005E9380 00560528  JobMain
...                                            00E614 brsl r000,Upd_RunProcessList
```

stack walk

```
03E580 | 0003E620 00002CA0 0003E620 0003E620  link
03E590 | 000060BC 00000000 00000000 00000000  return = 0x060bc
03E5A0 | 00000000 00000000 00000000 00000000  Upd_SimEmitterParticles
...                                            0060B8 brsl r000, Upd_SimBatch

03E620 | 0003E7A0 00002E20 0003E7A0 0003E7A0  link
03E630 | 0000CC98 00000000 00000000 00000000  return = 0x0cc98
03E640 | 01E04000 005D0000 01000000 00000000  Upd_SimStep
...                                            00CC94 brsl r000,Upd_SimParticles

03E7A0 | 0003E8D0 00002F50 0003E8D0 0003E8D0  link
03E7B0 | 00017364 00000000 00000000 00000000  return = 0x17364
03E7C0 | 40002400 00000000 00000000 00000000  Upd_RunProcessList
...                                            017360 brsl r000,Upd_SimStep

03E8D0 | 0003E910 00002F90 0003E910 0003E910  link
03E8E0 | 0000E618 00000000 00000000 00000000  return = 0x0E618
03E8F0 | 4057C080 DEADDEAD 005E9380 00560528  JobMain
...                                            00E614 brsl r000,Upd_RunProcessList
```

stack walk

```
03E580 | 0003E620 00002CA0 0003E620 0003E620  link
03E590 | 000060BC 00000000 00000000 00000000  return = 0x060bc
03E5A0 | 00000000 00000000 00000000 00000000  Upd_SimEmitterParticles
...                                            0060B8 brsl r000, Upd_SimBatch

03E620 | 0003E7A0 00002E20 0003E7A0 0003E7A0  link
03E630 | 0000CC98 00000000 00000000 00000000  return = 0x0cc98
03E640 | 01E04000 005D0000 01000000 00000000  Upd_SimStep
...                                            00CC94 brsl r000,Upd_SimParticles

03E7A0 | 0003E8D0 00002F50 0003E8D0 0003E8D0  link
03E7B0 | 00017364 00000000 00000000 00000000  return = 0x17364
03E7C0 | 40002400 00000000 00000000 00000000  Upd_RunProcessList
...                                            017360 brsl r000,Upd_SimStep

03E8D0 | 0003E910 00002F90 0003E910 0003E910  link
03E8E0 | 0000E618 00000000 00000000 00000000  return = 0x0E618
03E8F0 | 4057C080 DEADDEAD 005E9380 00560528  JobMain
...                                            00E614 brsl r000,Upd_RunProcessList
```

stack walk

```
03E580 | 0003E620 00002CA0 0003E620 0003E620   link
03E590 | 000060BC 00000000 00000000 00000000   return = 0x060bc
03E5A0 | 00000000 00000000 00000000 00000000   Upd_SimEmitterParticles
...                                            0060B8 brsl r000, Upd_SimBatch

03E620 | 0003E7A0 00002E20 0003E7A0 0003E7A0   link
03E630 | 0000CC98 00000000 00000000 00000000   return = 0x0cc98
03E640 | 01E04000 005D0000 01000000 00000000   Upd_SimStep
...                                            00CC94 brsl r000,Upd_SimParticles

03E7A0 | 0003E8D0 00002F50 0003E8D0 0003E8D0   link
03E7B0 | 00017364 00000000 00000000 00000000   return = 0x17364
03E7C0 | 40002400 00000000 00000000 00000000   Upd_RunProcessList
...                                            017360 brsl r000,Upd_SimStep

03E8D0 | 0003E910 00002F90 0003E910 0003E910   link
03E8E0 | 0000E618 00000000 00000000 00000000   return = 0x0E618
03E8F0 | 4057C080 DEADDEAD 005E9380 00560528   JobMain
...                                            00E614 brsl r000,Upd_RunProcessList
```

stack walk

```
03E580 | 0003E620 00002CA0 0003E620 0003E620   link
03E590 | 000060BC 00000000 00000000 00000000   return = 0x060bc
03E5A0 | 00000000 00000000 00000000 00000000   Upd_SimEmitterParticles
...                                             0060B8 brsl r000, Upd_SimBatch

03E620 | 0003E7A0 00002E20 0003E7A0 0003E7A0   link
03E630 | 0000CC98 00000000 00000000 00000000   return = 0x0cc98
03E640 | 01E04000 005D0000 01000000 00000000   Upd_SimStep
...                                             00CC94 brsl r000,Upd_SimParticles

03E7A0 | 0003E8D0 00002F50 0003E8D0 0003E8D0   link
03E7B0 | 00017364 00000000 00000000 00000000   return = 0x17364
03E7C0 | 40002400 00000000 00000000 00000000   Upd_RunProcessList
...                                             017360 brsl r000,Upd_SimStep

03E8D0 | 0003E910 00002F90 0003E910 0003E910   link
03E8E0 | 0000E618 00000000 00000000 00000000   return = 0x0E618
03E8F0 | 4057C080 DEADDEAD 005E9380 00560528   JobMain
...                                             00E614 brsl r000,Upd_RunProcessList
```

stack walk

```
03E580 | 0003E620 00002CA0 0003E620 0003E620   link ↩
03E590 | 000060BC 00000000 00000000 00000000   return = 0x060bc
03E5A0 | 00000000 00000000 00000000 00000000   Upd_SimEmitterParticles
...                                             0060B8 brsl r000, Upd_SimBatch

03E620 | 0003E7A0 00002E20 0003E7A0 0003E7A0   link ↩
03E630 | 0000CC98 00000000 00000000 00000000   return = 0x0cc98
03E640 | 01E04000 005D0000 01000000 00000000   Upd_SimStep
...                                             00CC94 brsl r000,Upd_SimParticles

03E7A0 | 0003E8D0 00002F50 0003E8D0 0003E8D0   link ↩
03E7B0 | 00017364 00000000 00000000 00000000   return = 0x17364
03E7C0 | 40002400 00000000 00000000 00000000   Upd_RunProcessList
...                                             017360 brsl r000,Upd_SimStep

03E8D0 | 0003E910 00002F90 0003E910 0003E910   link ↩
03E8E0 | 0000E618 00000000 00000000 00000000   return = 0x0E618
03E8F0 | 4057C080 DEADDEAD 005E9380 00560528   JobMain
...                                             00E614 brsl r000,Upd_RunProcessList
```

stack walk

```
03E580 | 0003E620 00002CA0 0003E620 0003E620   link
03E590 | 000060BC 00000000 00000000 00000000   return = 0x060bc
03E5A0 | 00000000 00000000 00000000 00000000   Upd_SimEmitterParticles
...                                             0060B8 brsl r000, Upd_SimBatch

03E620 | 0003E7A0 00002E20 0003E7A0 0003E7A0   link
03E630 | 0000CC98 00000000 00000000 00000000   return = 0x0cc98
03E640 | 01E04000 005D0000 01000000 00000000   Upd_SimStep
...                                             00CC94 brsl r000,Upd_SimParticles

03E7A0 | 0003E8D0 00002F50 0003E8D0 0003E8D0   link
03E7B0 | 00017364 00000000 00000000 00000000   return = 0x17364
03E7C0 | 40002400 00000000 00000000 00000000   Upd_RunProcessList
...                                             017360 brsl r000,Upd_SimStep

03E8D0 | 0003E910 00002F90 0003E910 0003E910   link
03E8E0 | 0000E618 00000000 00000000 00000000   return = 0x0E618
03E8F0 | 4057C080 DEADDEAD 005E9380 00560528   JobMain
...                                             00E614 brsl r000,Upd_RunProcessList
```

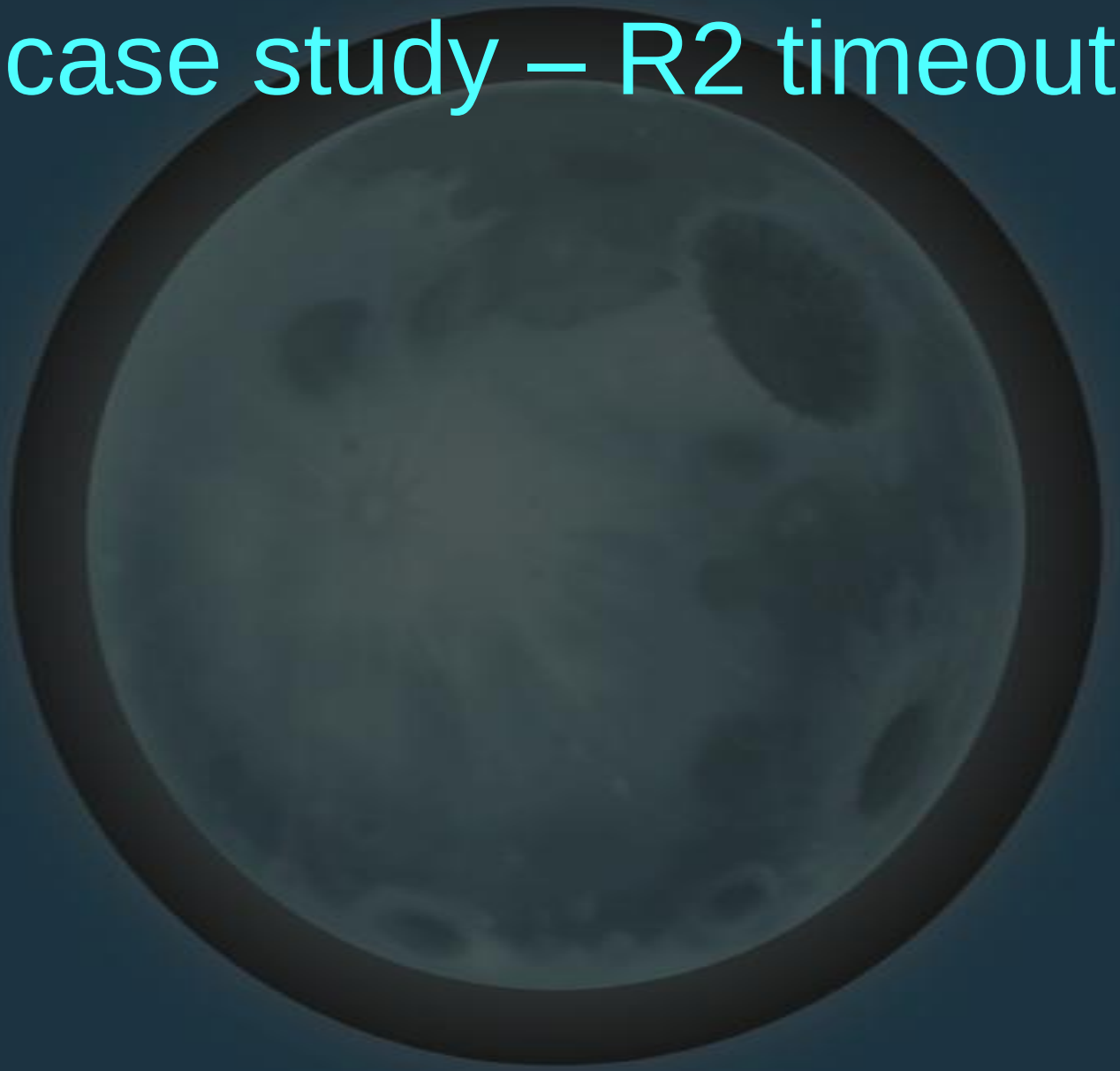
stack walk

```
03E580 | 0003E620 00002CA0 0003E620 0003E620   link ↩
03E590 | 000060BC 00000000 00000000 00000000   return = 0x060bc
03E5A0 | 00000000 00000000 00000000 00000000   Upd_SimEmitterParticles
...                                            0060B8 brsl r000, Upd_SimBatch

03E620 | 0003E7A0 00002E20 0003E7A0 0003E7A0   link ↩
03E630 | 0000CC98 00000000 00000000 00000000   return = 0x0cc98
03E640 | 01E04000 005D0000 01000000 00000000   Upd_SimStep
...                                            00CC94 brsl r000,Upd_SimParticles

03E7A0 | 0003E8D0 00002F50 0003E8D0 0003E8D0   link ↩
03E7B0 | 00017364 00000000 00000000 00000000   return = 0x17364
03E7C0 | 40002400 00000000 00000000 00000000   Upd_RunProcessList
...                                            017360 brsl r000,Upd_SimStep

03E8D0 | 0003E910 00002F90 0003E910 0003E910   link ↩
03E8E0 | 0000E618 00000000 00000000 00000000   return = 0x0E618
03E8F0 | 4057C080 DEADDEAD 005E9380 00560528   JobMain
...                                            00E614 brsl r000,Upd_RunProcessList
```

tight on LS

- modules are often very tight on LS in debug builds – options:
  - smaller buffers in DEBUG build – smaller batches - more looping (collision)
  - most files `-O2/3`, turn on `-O0` for the one(s) want to debug (specify per file options in devstudio / makefile)
  - create a separate file which is always compiled `-O0` and temporarily move your function(s) into it

case study – R2 timeout



case study – R2 timeout

- SPU timeout running from a FINAL-ish disc
- IG prints disabled
- no debugger access (wasn’t launched with “enable debugging of module”)
- do have OS TTY though (what luxury!)

case study – R2 timeout

- exception trace:

```
lv2(2): spu_thread (xxx) stopped due to exceptions
lv2(2):   thread: 0x00010100 (CellXXX0)
lv2(2):    group: 0x04010100 (CellXXXGroup)
lv2(2):  process: 0x01010500 (/dev_...)
lv2(2): exception causes:
lv2(2):  Stop break
lv2(2): SPU context:
lv2(2):  SPU_NPC    : 0x0001e459
lv2(2):  SPU_Status: 0x3fff0002
...
```

case study – R2 timeout

- exception trace:

```
lv2(2): spu_thread (xxx) stopped due to exceptions
lv2(2):   thread: 0x00010100 (CellXXX0)
lv2(2):    group: 0x04010100 (CellXXXGroup)
lv2(2):  process: 0x01010500 (/dev_...)
lv2(2): exception causes:
lv2(2):  Stop break
lv2(2): SPU context:
lv2(2):  SPU_NPC    : 0x0001e459
lv2(2):  SPU_Status: 0x3fff0002
...
```

case study – R2 timeout

- easy – we hit a `STOP`
- try find a `STOP` at PC `0x01e459`
- search our spu modules

case study – R2 timeout

- disassemble all modules:

```
spudisall elf\spu\*.elf
```

batchfile:

```
for %%f in (%1) do (
  echo spu-objdump -d %%f to %%f_dis.s
  spu-objdump -d %%f > %%f_dis.s
)
```

case study – R2 timeout

- trace: **`SPU_NPC: 0x0001e459`**
- is fibbing - PC is really **`0x1e458`**

  (4-byte alignment)
- and for a **`STOP`** is probably **`0x1e454`**

  (PC will be **`+4`** from the **`STOP`**)

case study – R2 timeout

- search our disassembled modules:

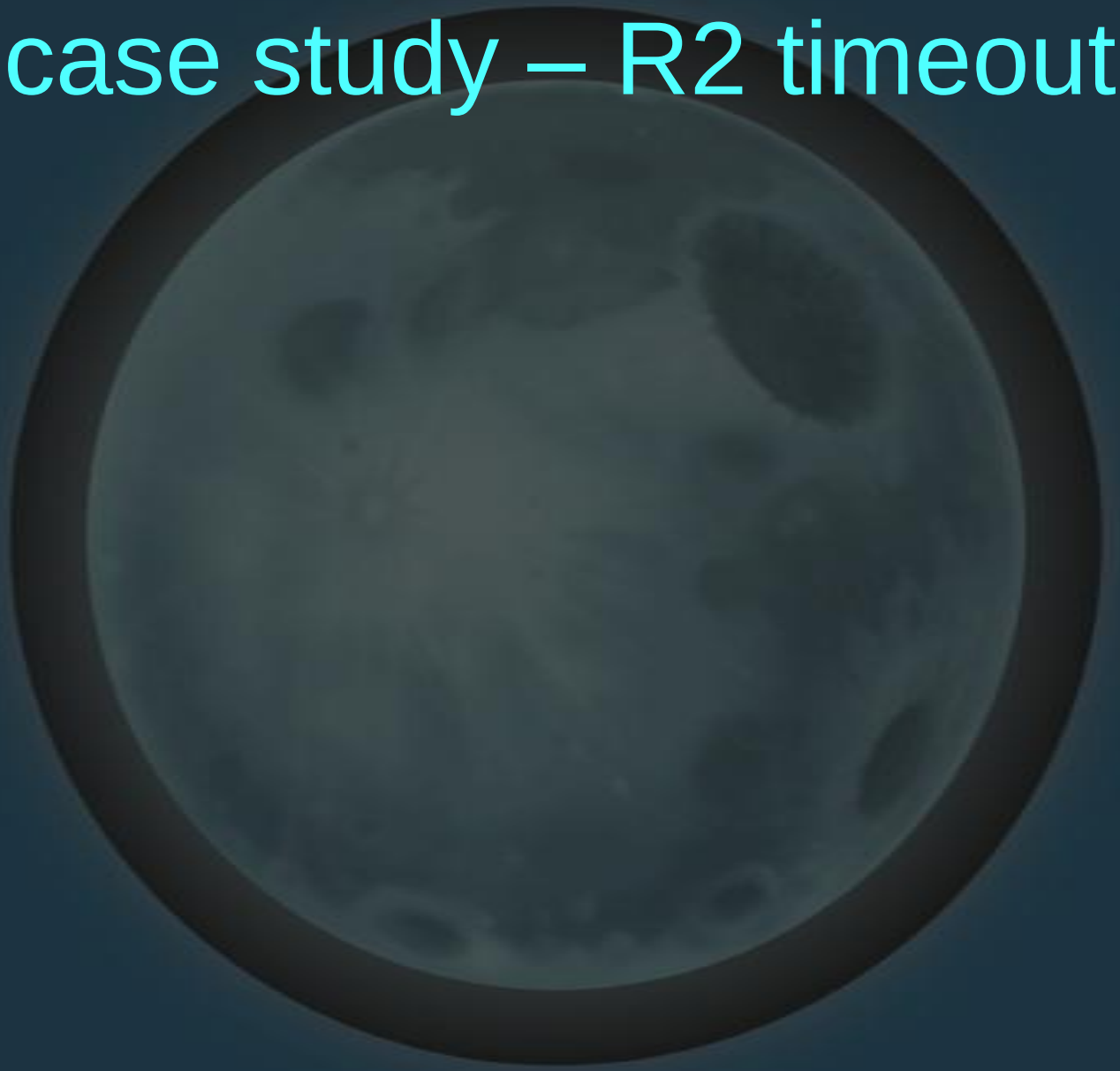
```
grep -i 1e458: -A 4 -B 4 elf\spu\*.s
```

```
igFXVisSpu.elf_dis.s-0x0001E448: ceqbi r002,r002,0x0000
igFXVisSpu.elf_dis.s-0x0001E44C: xsbh  r002,r002
igFXVisSpu.elf_dis.s-0x0001E450: brhnz r002,0x01E458
igFXVisSpu.elf_dis.s-0x0001E454: stopd
igFXVisSpu.elf_dis.s:0x0001E458: lqd   r002,0x00B0(r001)
igFXVisSpu.elf_dis.s-0x0001E45C: lr    r003,r002
igFXVisSpu.elf_dis.s-0x0001E460: il    r004,0x0010
igFXVisSpu.elf_dis.s-0x0001E464: brsl  r000,DMA_IsAligned(...)
```

case study – R2 timeout

- load **`igFXVisSpu.elf_dis.s`**
  - search up from the **`0x0001E454`** line
  - find **`DmaLargeGet`**
- assert in dma wrapper
- misaligned ptr being passed to a large-dma get / put
- easy fix

case study – R2 timeout



case study – R2 timeout

- PPU timeout code fired
  - no assert – launched from debugger – no exception dump - examine SPUs

```
...
FFFFFC | 00000000 stop  0x0000
000000 | 0000DEAD stop  0x1EAD
000004 | 0000DEAD stop  0x1EAD
000008 | 0000DEAD stop  0x1EAD          ← pc
00000C | 0000DEAD stop  0x1EAD
000010 | 42002800 ila   r000,0x000050
...
```

- callstack:

```
Type    Function
            ???   0x00000008
```

  - how helpful!

case study – R2 timeout

- registers:

```
r000 [004C0003 00000000 00000000 00000000] = LR = return address
r001 [0003D8E0 00001F60 0003D8E0 0003D8E0] = SP
```

- SPU memory wraps:

```
        0x004c0003
  AND   0x0003ffff
        0x00000003
```

- looks like we branched to `0x00003` then stopped - PC:

```
000004 0000DEAD stop 0x1EAD
000008 0000DEAD stop 0x1EAD ← pc
```

- lets dump the stack (from **SP** till we feel like stopping)

```
03D8E0 | 0003DDA0 00002420 0003DDA0 0003DDA0 link
03D8F0 | 004C0003 00000000 00000000 00000000 return
...    |
       | 0003E360 000029E0 0003E360 0003E360 link
03DDA0 | 000270F8 00000000 00000000 00000000 return
03DDB0 |
...    | 0003E580 00002C00 0003E580 0003E580 ...
       | 00004F98 00000000 00000000 00000000 ...
03E360 |
03E370 | 0003E620 00002CA0 0003E620 0003E620
...    | 000060BC 00000000 00000000 00000000

03E580 | 0003E7A0 00002E20 0003E7A0 0003E7A0
03E590 | 0000CC98 00000000 00000000 00000000
...    |

03E620 | 0003E7A0 00002E20 0003E7A0 0003E7A0 ...
03E630 | 0000CC98 00000000 00000000 00000000 ...
...    |
```

```
03D8E0 | 0003DDA0 00002420 0003DDA0 0003DDA0 link
03D8F0 | 004C0003 00000000 00000000 00000000 return = ???
...    |
       |
03DDA0 | 0003E360 000029E0 0003E360 0003E360 link
03DDB0 | 000270F8 00000000 00000000 00000000 return 0x270F8
...    |                                      shader_update_particles_simple
       |
03E360 | 0003E580 00002C00 0003E580 0003E580 ...
03E370 | 00004F98 00000000 00000000 00000000 return 0x04F98
...    |                                      Upd_SimPrepedParticleBatch
       |
03E580 | 0003E620 00002CA0 0003E620 0003E620 ...
03E590 | 000060BC 00000000 00000000 00000000 return 0x060BC
...    |                                      Upd_SimEmitterParticles
       |
03E620 | 0003E7A0 00002E20 0003E7A0 0003E7A0 ...
03E630 | 0000CC98 00000000 00000000 00000000 return 0x0CC98
...    |                                      Upd_SimStep
```

case study – R2 timeout

- stomp is return of current stack-frame:

```
03D8E0 | 0003DDA0 00002420 0003DDA0 0003DDA0 link
03D8F0 | 004C0003 00000000 00000000 00000000 return ← stomp!
...    |
```

- remember the ABI
  - return adddress stashed in *parent's* frame before new frame created
- `0x3d8f0` is the return for the function after this

case study – R2 timeout

- frame before the stomp was **fxvis_shader_update_particles_simple**

```
03D8E0 | 0003DDA0 00002420 0003DDA0 0003DDA0 link
03D8F0 | 004C0003 00000000 00000000 00000000 return = ???
...    |
       |
03DDA0 | 0003E360 000029E0 0003E360 0003E360 link
03DDB0 | 000270F8 00000000 00000000 00000000 return 0x270F8
...    |                                      shader_update_particles_simple
```

- calls **update_api.Emitter_StandardParticleSpawn();**

```
0270E8 lqx    r009,r088,r011
0270EC stqd   r010,0x0260(r001)
0270F0 rotqby r003,r009,r004
0270F4 bisl   r000,r003
0270F8 ilhu   r013,0x3B80      ← return
```

- from the code, this is **Upd_StandardParticleSpawn**

case study – R2 timeout

- frame with the stomp:

```
03D8E0 | 0003DDA0 00002420 0003DDA0 0003DDA0 link
03D8F0 | 004C0003 00000000 00000000 00000000 return = ???
03D900 | 404641D4 40B1AFCD 40C4D07C 40A609F6
...    |
```

- so this is the stack frame for `Upd_StandardParticleSpawn`
- quick check:

```
Upd_StandardParticleSpawn()
   007228 40FDA00B il   r011,-0x04C0
   ...
   007310 1802C081 a    r001,r001,r011
```

- allocates a `0x4c0` (1216) byte stack frame
- frame we dumped is `0x03d8e0` -> `0x03dda0` = `0x4c0` bytes - huzzah!

case study – R2 timeout

- so we called a function
  - it stashed the return address into this frame
- at some point that return was trashed
- tried to return to bad address
- lets try find out which function we called
- `Upd_StandardParticleSpawn` has about 15 branches (joy!)
- lets look at the most recent (freed) stack frame

case study – R2 timeout

```
03D790 | 0003D8E0 00001F60 0003D8E0 0003D8E0 link
03D7A0 | 0002A3F0 00000000 00000000 00000000 return 0x2a3f0
...                                          shader_particle_spawn_style_disc_perp
                                             02A3E8 nop
                                             02A3EC bisl  r000,r029
                                             02A3F0 ai    r001,r001,0x0150

03D8E0 | 0003DDA0 00002420 0003DDA0 0003DDA0 link
03D8F0 | 004C0003 00000000 00000000 00000000 return = ???
```

- so **Upd_StandardParticleSpawn** called something
- that function called **shader_particle_spawn_style_disc_perp**
- previous stack-frame is valid
  - **fxvis_shader_particle_spawn_style_disc_perp** also called / returned
  - we know this as the return was stashed into the parent frame and actual stack ptr was adjusted

case study – R2 timeout

```
03D8E0 | 0003DDA0 00002420 0003DDA0 0003DDA0 link
03D8F0 | 004C0003 00000000 00000000 00000000 return = ???
```

- this is the stack frame of `Upd_StandardParticleSpawn`
- assume `shader_particle_spawn_style_disc_perp` or something it called trashed the parent stack-frame
- let's look at `shader_particle_spawn_style_disc_perp`

case study – R2 timeout

```
shader_particle_spawn_style_disc_perp
 029DB0 nop
 029DB4 stqd r094,-0x00F0(r001)
 029DB8 il   r094,0x0034
 029DBC stqd r000,0x0010(r001)   1
 ...
 029E50 ai   r001,r001,-0x0150   2
...
 <stuff here>                    3
...
```

```
 02A3F0 ai   r001,r001,0x0150    4
 02A5EC lqd  r000,0x0010(r001)   5
 02A5F0 lqd  r080,-0x0010(r001)
 ...
 02A640 bi   r000                6
```

1: preserve LR and friends

2: *allocate* frame

3: do stuff

4: *free* frame

5: restore LR and friends

6: return

case study – R2 timeout

- so, we called `shader_particle_spawn_style_disc_perp`
  - it preserved the return address and set up a stack-frame
  - it called some other functions – they returned
- either one of those functions or this function itself trashed the parents stack-frame
- we then restored the LR and jumped to the trashed address (`0x004C0003` = `0x00003`)
- what fun!

summary

- looked at Insomniac SPU job-manager
  - very simple - fits our model well
- discussed some of our SPU debugging strategies
- looked at some case-studies
- what are your SPU debugging tricks and tips ?

end!

- thanks for turning up
- thanks to everyone on the Insomniac engine team
- questions ?

jonathan garrett

jonny@insomniacgames.com