i915 command submission via gem_exec_nop

Author : Ben Widawsky

Disclaimer: Everything documented below is included in the Intel public documentation. Anything I say which offends you are my own words and not those of Intel. Sadly, anything I say that is of monetary value are those if Intel.

Goal

My goal is to lay down a basic understanding of how GEN GPU execution works using gem_exec_nop from the intel-gpu-tools suite as an example. One who puts in the time to read this should understand how command submission works for the i915 driver, and how gem_exec_nop tests command submission. You should also have a decent idea of how the hardware handles execution. I intentionally skip topics like relocations, and how graphics virtual addresses are maintained. They are not directly related towards execution, and would make the blog entry too long.

Ideally, I am hoping this will enable people who are interested to file better bugs, improve our tests, or write their own tests.

Terminology

- i915: The name of the Linux kernel driver for Intel GEN graphics. i915 is the name of an ancient chipset that was one of the first supported by the driver. The driver itself supports chipsets both before, and after i915.
- BO: Buffer Object. GEM uses handles to identify the buffers used as graphics operands in order to avoidly costly copies from userspace to kernel space. BO is the thing which is encapsulated by that handle.
- GEM: Graphics Execution Manager. The name of a design and API to give userspace GPU clients the ability to execute work on a GPU (the API is technically not specific to GEN).
- GEN: The name of the Graphics IP developed by Intel Corporation.
- GPU client: A userspace application or library that submits GPU work.
- Graphics [virtual] Address: Address space used by the GPU for mapping system memory. GEN is an UMA architecture with regard to the CPU.
- NOP/NOOP: An assembly instruction mnemonic for a machine opcode that does no work. Note that this is not the same as a lack of work. The instruction is indeed executed, it simply has no side-effects. The execution latency is strictly greater than zero.
- relocations: The way in which GEM manages to make GPU clients agnostic to where the buffers are actually mapped by the GPU. Out of scope for this blog entry.

Code

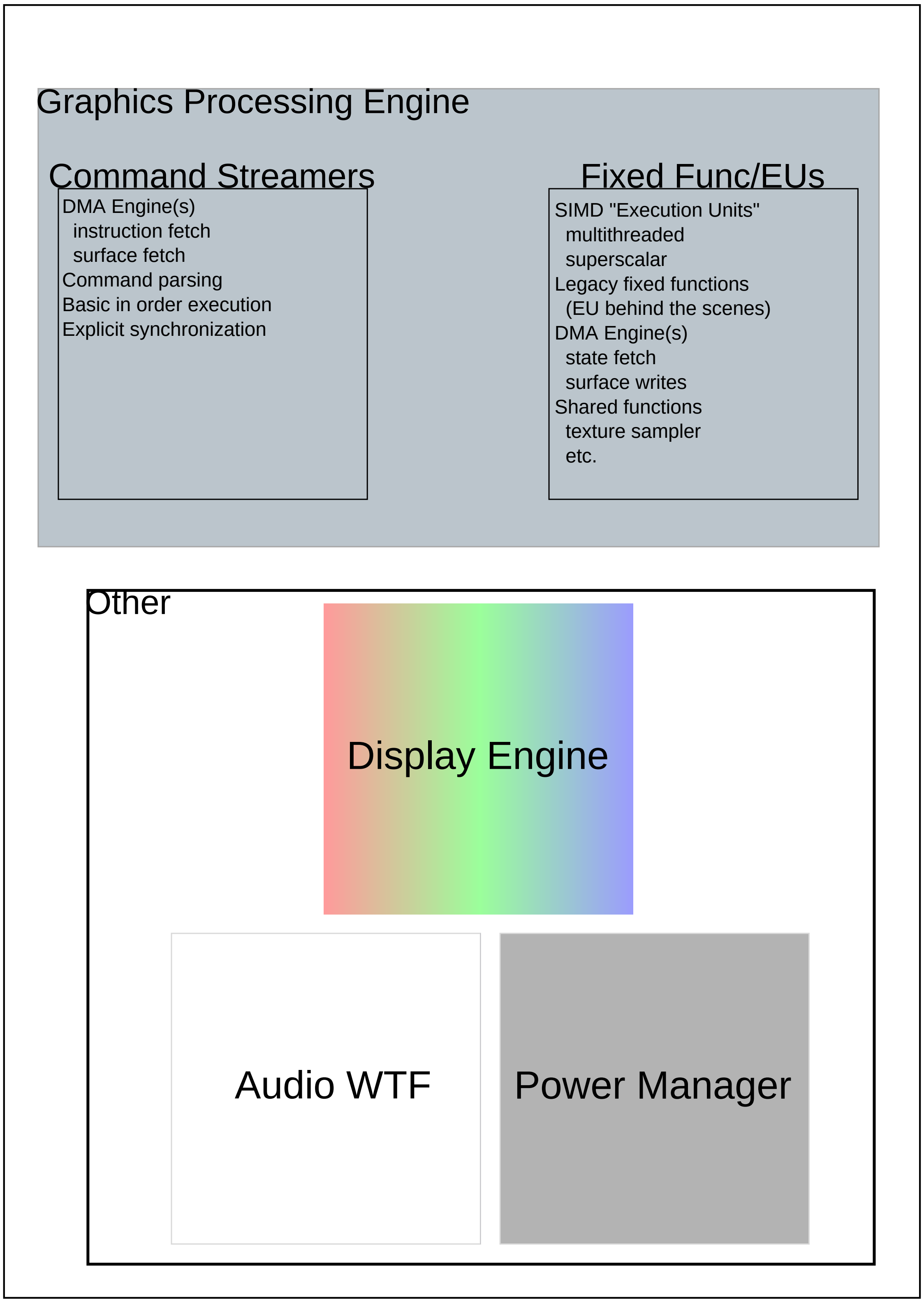
The source code in this post is found primarily in two places. Note that the links below are both from very fast moving code bases.

The test case: http://cgit.freedesktop.org/xorg/app/intel-gpu-tools/tree/tests/gem_exec_nop.c

The driver internals: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/gpu/drm/i915/i915_gem_execbuffer.c

GEN Hardware

Before going over gem_exec_nop, I'd like to give an overview of modern GEN hardware:


Graphics Processing Engine
Command Streamers
DMA Engine(s)
instruction fetch
surface fetch
Command parsing
Basic in order execution
Explicit synchronization
Fixed Func/EUs
SIMD "Execution Units"
multithreaded
superscalar
Legacy fixed functions
(EU behind the scenes)
DMA Engine(s)
state fetch
surface writes
Shared functions
texture sampler
etc.
Other
Display Engine
Audio WTF
Power Manager

I don't want to say this is the exhaustive list, and indeed, each block above has many sub-components. In the part of the driver I work on this is a pretty logical way to split it. Each of the blocks share very little. The common denominator is a **Graphics Virtual Address** which is understood by all blocks. This provides an easy communication for work needing to be sent between components. As an example, the command streamer might want the display engine to flip to a new surface. It does so by sending a special message to the display engine along with the address of the surface to flip to. The display engine may respond "out of band" via interrupts (flip completion). There are also built in synchronization primitives that allow the command streamer to wait on events sent by the display engine (we'll get to the command streamer in more detail later).

Excluding audio, since I know nothing about audio… by a very rough estimate, 85% of the Linux i915.ko code falls into "Other." Of the remaining 15% in graphics processing engine, the kernel driver tends to utilize very little of the Fixed Func/EU block above. Total lines of code outside of the kernel driver for the EU block is enormous, given that the X 2d driver (DDX), mesa, libva, and beignet all have tons of lines of code just for utilizing that part of the hardware.

gem_exec_nop features

gem_exec_nop is one of my favorite tests. For me, it's the first test I run to determine whether or not to even bother with the rest of the test suite.

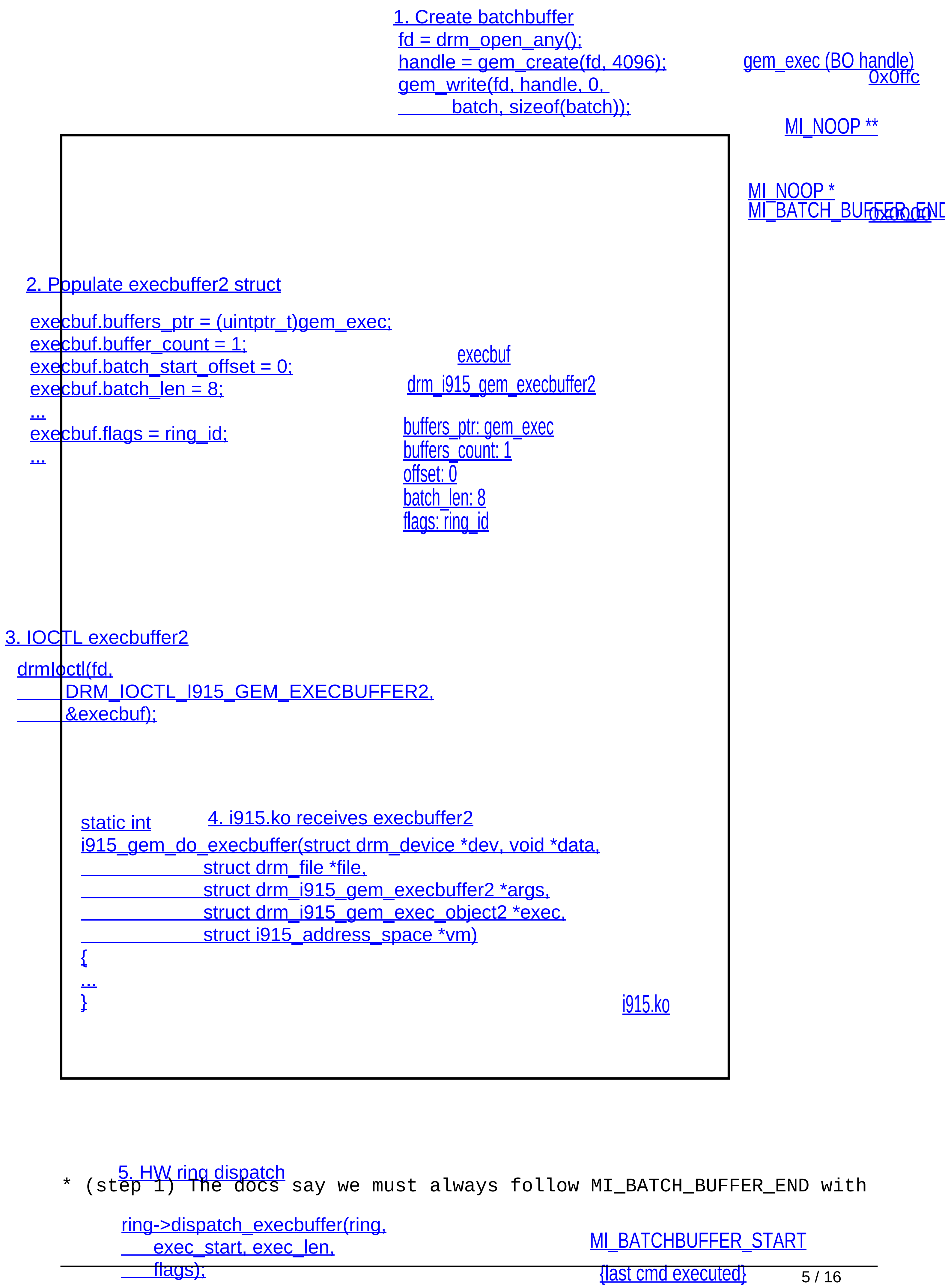
- It's dead simple.
- It's fast.
- It tests a surprisingly large amount of the hardware, and software.
- Gives some indication of performance
- It's deader than dead simple

It's not a perfect test, some of the things which are missing:

- Handling under memory pressure (relocs, swaps, etc.)
- Tiling formats
- Explicit testing of cacheability types, and coherency (LLC et al.)
- several GEM interfaces
- The aforementioned 85% of the driver
- It doesn't even execute a NOP instruction!!!

gem_exec_nop flowchart thing

NOTE: I will explain more about what a batchbuffer is later.


1. Create batchbuffer

```
fd = drm_open_any();
handle = gem_create(fd, 4096);
gem_write(fd, handle, 0,
          batch, sizeof(batch));
```

2. Populate execbuffer2 struct

```
execbuf.buffers_ptr = (uintptr_t)gem_exec;
execbuf.buffer_count = 1;
execbuf.batch_start_offset = 0;
execbuf.batch_len = 8;
...
execbuf.flags = ring_id;
...
```

3. IOCTL execbuffer2

```
drmIoctl(fd,
         DRM_IOCTL_I915_GEM_EXECBUFFER2,
         &execbuf);
```

4. i915.ko receives execbuffer2

```
static int
i915_gem_do_execbuffer(struct drm_device *dev, void *data,
                       struct drm_file *file,
                       struct drm_i915_gem_execbuffer2 *args,
                       struct drm_i915_gem_exec_object2 *exec,
                       struct i915_address_space *vm)
{
...
}
```

5. HW ring dispatch

```
ring->dispatch_execbuffer(ring,
     exec_start, exec_len,
     flags);
```

* (step 1) The docs say we must always follow MI_BATCH_BUFFER_END with

```
 an MI_NOOP. The presumed reason for this is that the hardware may pre
fetch the next instruction, and I think the designers wanted to dumb d
own the fact that they can't handle a pagefault on the prefetch, so th
ey simply demand a MI_NOOP.  ** (step1) MI_NOOP is defined as a dword
of value 0x00000000. GEM BOs are 0 by default. So we have an implicit
buffer full of MI_NOOP.
```

1. Creating a batchbuffer is done using GEM APIs. Here we create a batchbuffer of size 4096, and fill in two instructions. The batchbuffer is the basic unit of execution. The only pertinent point to keep in mind is this is the only buffer being created for this test. Note that this step, or a similar one, is done in almost every test.
2. Here we set up the data structure that will be passed to the kernel in an IOCTL. There's a pointer to the list of buffers, in our case, just the one batchbuffer created instead one. The size of 8 (each of the two instructions is 4 bytes), and some flags which we'll skip for now are also included in the struct.
3. The dotted line through step 3 denotes the userspace/kernel barrier. Above the line is gem_exec_nop.c, below is i915_gem_execbuffer.c. DRM, which is a common subsystem interface, actually dispatches the IOCTLs to the i915 driver.
4. The kernel handling the data is received. Talked about in more detail later.
5. Submit to the GPU for execution. Also, detailed later.

i915.ko execbuffer2 handling (step 4 and 5 in the picture above)

The eventual goal of the kernel driver is to take the batchbuffer passed in from userspace, make sure it is visible to the GPU by mapping it, and then submit it to the GPU for execution. The aforementioned operations are synchronous with respect to the IOCTL[1]. In other words, by the time the execution returns to the application, the GPU knows about the work. The work is completed asynchronously.

I'll detail some of the steps a bit. Unfortunately, I do not have pretty pictures for this one. You can follow along in i915_gem_execbuffer.c; i915_gem_do_execbuffer()

1. copy_from_user – Copy the BOs in from userspace. Remember that the BO is a handle and not actual memory being copied; this allows a relatively small and fast copy to take place. In gem_exec_nop, there is exactly 1 BO: the batchbuffer.
2. some sanity checks – not interesting
3. look up – Do a lookup of all the handles for the BOs passed in via the buffers_ptr member (copied in during #1). Make sure the buffers still exist and so on. In our case this is only one buffer and it's unlikely that it would be destroyed before execbuffer completes[2].
4. Space reservation – Make sure there is enough address space in the GPU for the objects. This also includes checking for various alignment restrictions, and a few other

details not really relevant to this specific topic. For our example, we'll have to make sure we have enough space for 1 buffer of size 4096, and no special alignment requirements. It's the second simplest request possible (first would be to have no buffers).
5. Relocations – save for another day.
6. Ring synchronization – Also not pertinent to gem_exec_nop. Since it involves the command streamer, I'll include a brief description as a footnote [3]
7. Dispatch – Finally we can tell the GEN hardware about the work that we just got. This means using some architectural registers to point the hardware at the batchbuffer which was submitted by userspace. More on this shortly…
8. Some more relocation stuff – save for another day

Execution part I (Command Streamer/Ringbuffer)

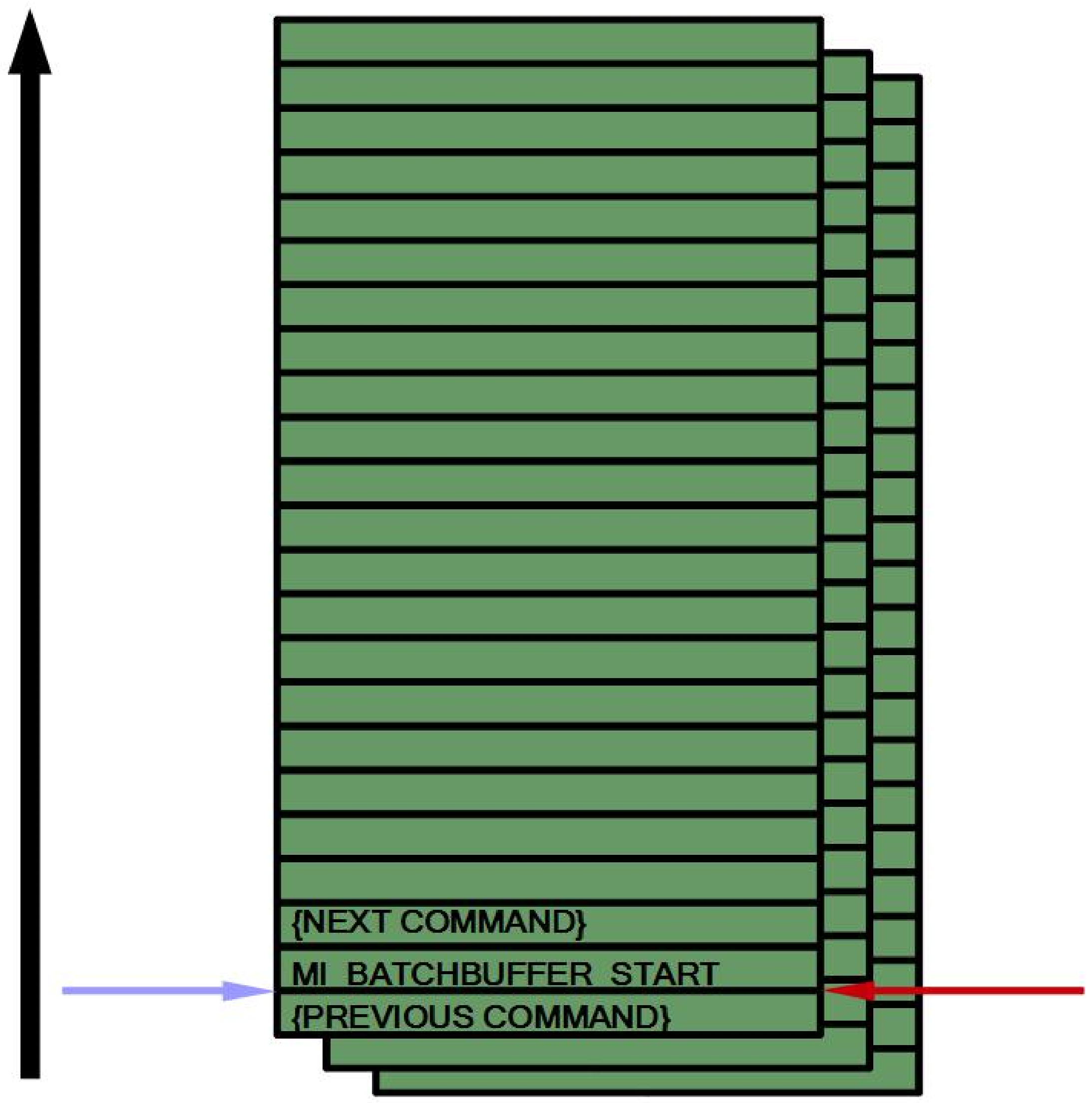
Fundamentally, all work is submitted via a hardware ringbuffer, and fetching via the command streamer. A command streamer is many things, but for now, saying it's a DMA engine for copying in commands and associated data is a good enough definition. The ringbuffer is a canonical ringbuffer with a HEAD and TAIL pointer (to be clear: TAIL is the one incremented by the CPU, and read by the GPU. HEAD is written by the GPU and read by the CPU). There is a third pointer known as ACTHD (or Active HEAD) – more on this later. At driver initialization, the space for the ringbuffer is allocated, and the address and size is written to hardware registers. When the driver wants to submit work, it writes data at the current TAIL pointer, and increments the TAIL pointer. Once the TAIL is incremented, the hardware will start reading in commands (via DMA), and increment the HEAD (and ACTHD) pointer as commands are retired.

Early GEN hardware had only 1 command streamer. It was referred to as, "CS." When Ironlake introduced the VCS, or video engine command streamer, they renamed (in some places) the original CS to RCS, for render engine command streamer. Sandybridge introduced the blit engine command streamer BCS, and Haswell the video enhancement command streamer, or VECS. Each command streamer supports its own instruction set, though many instructions are the same on multiple command streamers, MI_NOOP is supported on all of them

Having multiple command streamers not only provides an easy way to add new instructions, but it also allows an asynchronous way to submit work, which can be very useful if you are trying to do two orthogonal tasks. As an example, take an OpenCL application running in conjunction with your favorite 3d benchmark. The 3d benchmark internally will only use the 3d and blit hardware, while the OCL application will use the GPGPU hardware. It doesn't make sense to have either one wait for a single command streamer to fetch the data (especially since I glossed over some other details which make it an even worse idea) if there won't be any [or few] data dependencies.

The kernel driver is the only entity which can insert commands into the ringbuffer. The ringbuffer is therefore considered trusted, and all commands supported by the hardware may be run here (the docs use the word, "secure" but this gets confusing quickly) [4]. The way in which the batchbuffer we created in gem_exec_nop gets executed will be explained a bit further shortly, but the contents of that batchbuffer **are not** directly inserted into the ringbuffer. Take a quick peek at the text in the image below for how it works.

Here is a pretty basic picture describing the above. The HEAD and TAIL point to the next instruction to be executed, therefore this would be midway through step #5 in the flowchart above.

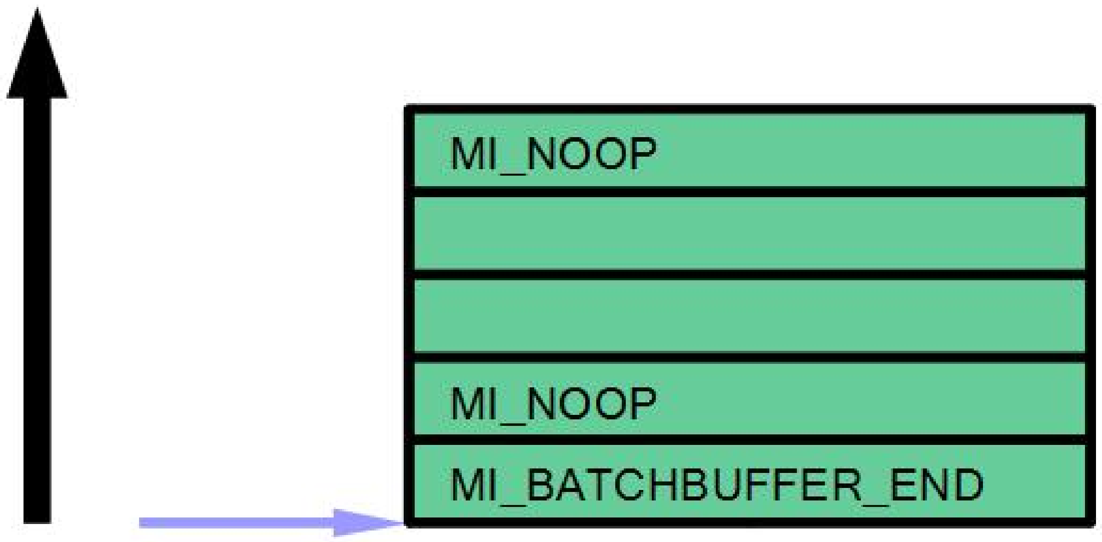


Execution part II (MI_BATCH_BUFFER_START,

batchbuffer)

A batchbuffer is the way in which we can submit work to the GPU without having to write into the hardware ringbuffer (since only the kernel driver can do that). A batchbuffer is submitted to the GPU for execution via a command called **MI_BATCH_BUFFER_START** which is inserted into the ringbuffer and read by the command streamer. Batchbuffers share an instruction set with the command streamer that dispatched them (ie. batches run by the blit engine can issue blit commands), and the execution flow is very similar to that of the command streamer as described in the first diagram, and subsequently. On the other hand, there are quite a few differences. Batchbuffer execution is not guided by HEAD, and TAIL pointers. The hardware will continue to execute every instruction in a batchbuffer until it hits another MI_BATCH_BUFFER_START command, or an MI_BATCH_BUFFER_END. Yes, you can get into an infinite loop of batchbuffers with this nesting of MI_BATCH_BUFFER_START commands. The hardware has an internal HEAD pointer which is exposed for debug purposes called ACTHD. This pointer works exactly like a HEAD point would, except it is never compared against TAIL to determine the end of execution[5]. MI_BATCH_BUFFER_END will directly guide execution back the hardware ringbuffer. In other words you need only one MI_BATCH_BUFFER_END to break the chain of n MI_BATCH_BUFFER_STARTs.

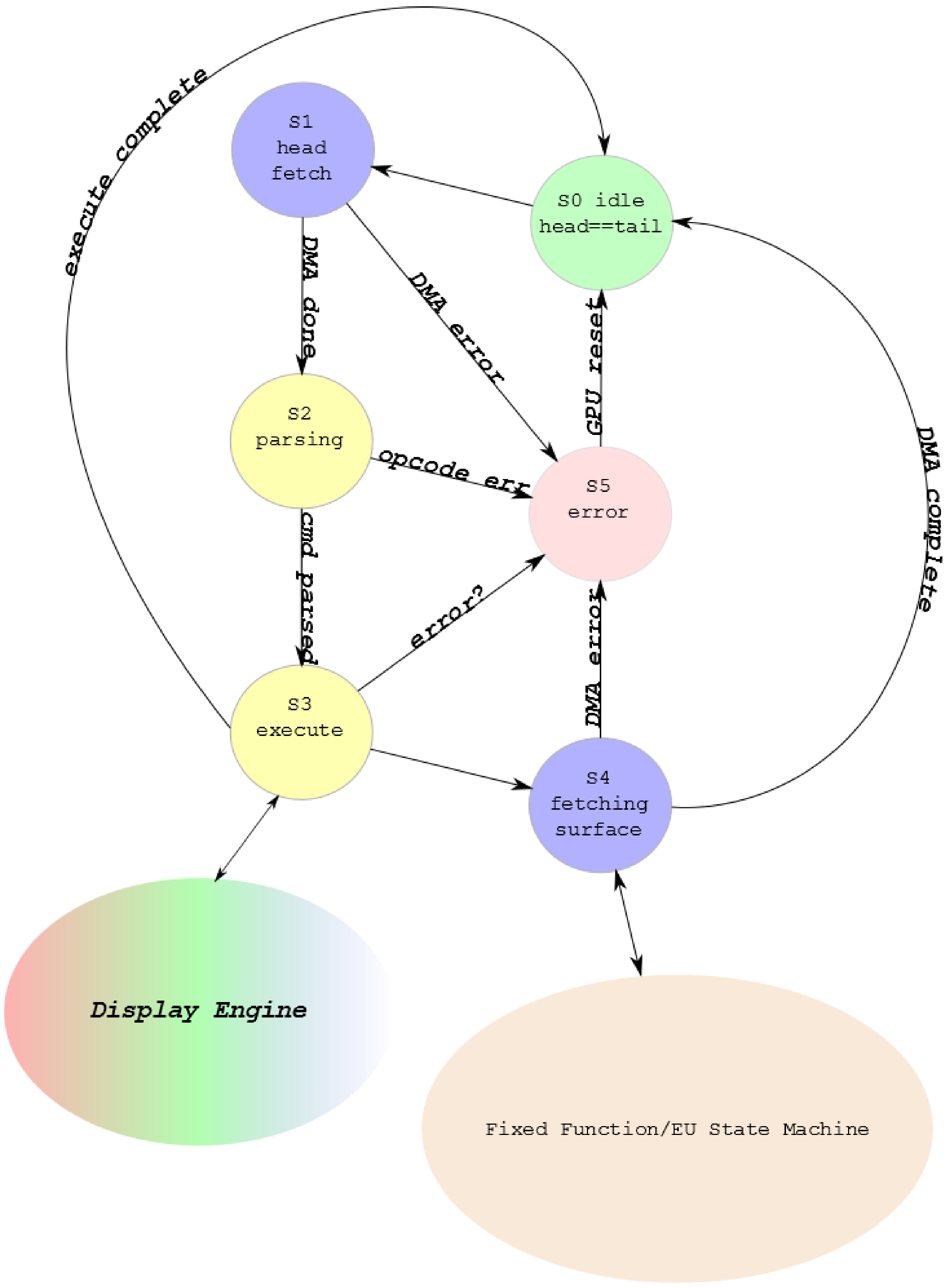
Getting back to gem_exec_nop specifically for a sec: this is what we set up in step #1. Recall it had 2 instructions MI_BATCH_BUFFER_END, MI_NOOP.



Here is our graphical representation of the batchbuffer from gem_exec_nop. Notice that the batchbuffer doesn't have a tail pointer, only ACTHD.

Hardware states

The follow macro-level state machine/flowchart hybrid can be used to describe both ringbuffer execution and batchbuffer execution, though the descriptions differ slightly. By “macro-level” I mean each state may not match exactly to a state within the hardware’s state machines. It’s more of a state in the data flow. The “state machines” for both ringbuffers and batchbuffers are pretty similar. What follows is a diagram that mostly works for both, and a description of each state.


S1
head
fetch
S0 idle
head==tail
S2
parsing
S5
error
S3
execute
S4
fetching
surface
execute complete
DMA done
DMA error
GPU reset
opcode err
cmd parsed
error?
DMA error
DMA complete
Display Engine
Fixed Function/EU State Machine

I’ll use “RSn” for ringbuffer state n, and “BSn” for batchbuffer state n.

- RS0: Idle state, HEAD == TAIL. Waiting for driver to increment tail.
- RS1: TAIL has changed. Fetch some amount between HEAD and TAIL (I’d guess it fetches the whole thing since the ringbuffer size is strictly limited).
- RS2: Fetch has completed, and command parsing can begin. Command parsing here is relatively easy. Every command is 4b aligned, and has the total command length embedded in the first 4th (1 based) byte of the opcode. Once it has determined the length, it can send that many dwords to the next stage.
- RS3: 1 command has been parsed and sent to be executed (pun intended).
- RS4: Execute phase required some more work, if the command executed in RS3 requires some extra data, now is when it will get fetched – and AFAICT, the hardware will stall waiting for the fetch to complete. If there is nothing left to do for the command, HEAD is incremented. Most commands will be done and increment HEAD. MI_BATCH_BUFFER_START is a common exception. I wish I could easily change the image… this is really RS3.5.
- RS5: An error state requiring a GPU reset.
- BS0: ASSERT(last command != MI_BATCH_BUFFER_END) This isn’t a real state. While executing a batchbuffer, you’re never idle. We can use this state as a place to update ACTHD though, so let’s say ACTHD := batchbuffer start address.
- BS1: Similar to RS1, fetch the data. Hopefully most of it exists in some internal cache since we had to fetch some amount of it in RS4, but I don’t claim to know the micro-architecture details on this.
- BS2: Just like RS2
- BS3: Just like RS3
- BS4: Just like RS4

gem_exec_nop state walkthrough

With the above knowledge, we can now step through the actual *stuff* from gem_exec_nop. This combines pretty much all the diagrams above (ie. you might want to reference them), I tried to keep everything factually correct along the way minus the address I make up below. Assume HEAD = 0×30, TAIL = 0×30, ACTHD=0×30

1. Hardware is in Rs0.
2. gem_exec_nop runs; submits previously discussed setup to i915.
3. *** kernel picks address 0×22000 for the batchbuffer (remember I said we’re ignoring how graphics addresses work for now, so just play along)
4. i915.ko writes 4bytes, MI_BATCH_BUFFER_START to hardware ringbuffer.
5. i915.ko writes 4 bytes, 0×22000 to hardware ringbuffer.
6. i915.ko increments the tail pointer by command length (8). TAIL := 0×38

7. RS0->RS1: DMA fetches TAIL-HEAD bytes. (0×38-0×30) = 8b
8. RS1->RS2: DMA completes. Parsing will find that the command is MI_BATCH_BUFFER_START, and it needs 1 extra dword to proceed. This 8b command is then ready to move on.
9. RS2->RS3: Command was successfully parsed. There is a batchbuffer to be fetched, and once that completes we need to execute it.
10. RS3->RS4: Execution was okay, DMA fetch of the batchbuffer at 0×22000 starts…completes
11. RS4->BS0: ACTHD := 0×22000
12. BS0->BS1: We're in a batchbuffer. The commands we need to fetch are in our local cache, fetched by the ringbuffer just before so no need to do anything more.
13. BS1->BS2: Parsing of the batchbuffer begins. The first command pointed to by ACTHD is MI_BATCH_BUFFER_END. It is only 4b.
14. BS2->BS3: Parse was successful. Execute the command MI_BATCH_BUFFER_END. ACTHD := 4. There are no extra requirements for this command.
15. BS3->RS0: Batchbuffer told us to end, so we go back to the ring. Increment our HEAD pointer by the size of the last command (8b). Set ACTHD equal to HEAD. HEAD := 0×38. ACTHD := 0×38.
16. HEAD == TAIL… we're idle.

Summary

User space builds up a command, and list of buffers. Then the userspace tells the kernel about it via IOCTL. Kernel does some work on the command to find all the buffers and so on, then submits it to the hardware. Some time later, userspace can see the results of the commands (not discussed in detail). On the hardware side, we've got a ringbuffer with a head and tail pointer, a way to dispatch commands which are located sparsely in our address space, and a way to get execution back to the ringbuffer.

footnotes

[1] – click to return

The synchronous nature of the IOCTL is something which has been discussed several times. One usage model which would really like to break that is a GPU scheduler. In the case of a scheduler, we'd want to queue up work and return to userspace as soon as possible; but that work may not yet make it into the hardware.

[2] – click to return

Buffer objects are managed with a reference count. When a buffer is created, it gets a ref count of 1, and the refcount is decremented either when the object is explicitly destroyed, or the application ceases to exist. Therefore, the only way gem_exec_nop can fail during the look up portion of execbuffer, is if the application somehow dies after creating the batchbuffer, but before calling the execbuffer IOCTL.

[3] – click to return

As I showed in the first diagram we consider the command executed to be “in order.” Here this means that commands are executed sequentially, (and hand waving over some caching stuff) the side effects of commands are completed by execution of the later commands. This made the implicit synchronization that is baked in to the GEM API really easy to handle (the API has no ways to explicitly add synchronization objects). To put this another way, if a GPU client submits a command that operates on object X, then a second command also operating on object X; it was guaranteed to execute in that order (as long as there was no race condition in userspace submitting commands). However, when you have multiple instances of the in-order command streamers, synchronization is no longer free. If a command is submitted to command streamer1 referencing object X, and then a second command is submitted to command streamer2 also referencing object X… no guarantees are made by hardware about the order the of the commands. In this case, synchronization can be achieved in two ways: hardware based semaphores, or by stalling on the second command until that first one completes.

[4] – click to return

Certain commands which may provide security risks are not allowed to be executed by untrusted entities. If the

hardware parses such a command from an untrusted entity, it will convert it into an MI_NOOP. Batchbuffers can be executed in a trusted manner, but implementing such a thing is complex.

[5] – click to return

When the CS is execution from the ring, HEAD == ACTHD. Once the CS jumps into the batchbuffer, ACTHD

will take on the address within the batchbuffer, while HEAD will remain only relevant to it’s position in the ring.

We use this fact to help us debug whether we hung in the batch, or in the ring.