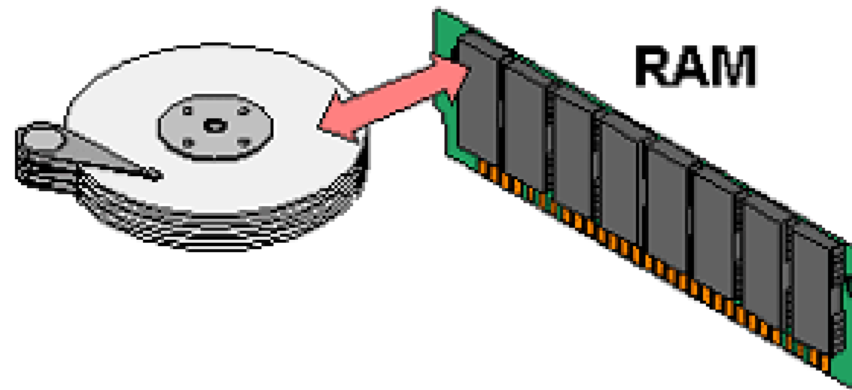
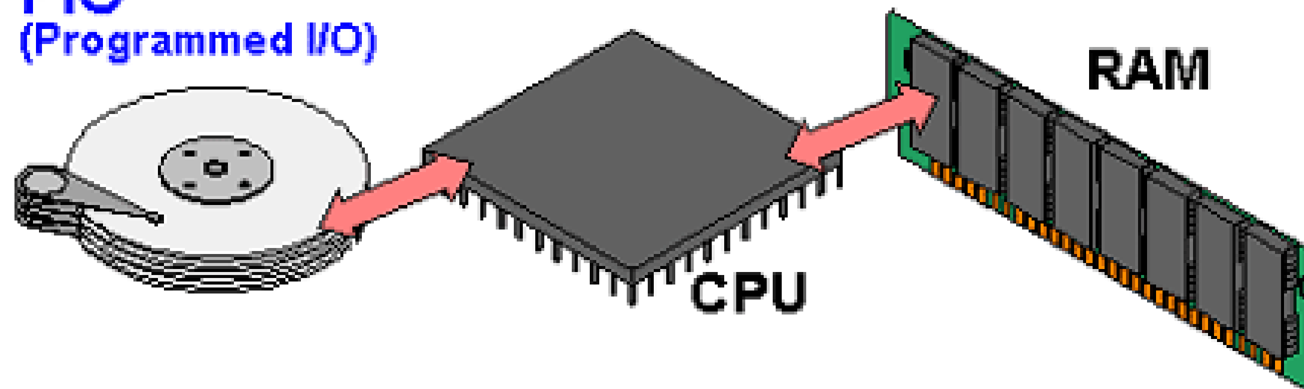
# Direct Memory Access

- https://www.kernel.org/doc/Documentation/DMA-API.txt
- https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

# Direct Memory Access (DMA)



# Free Up CPU to do other work

CPU cannot be accessing the memory during a DMA transfer. However there are two factors which in combination allow apparent parallel memory access by the CPU and the device performing the DMA transfer:

- Memory might have multiple access channels, CPU will use one and DMA another
- The CPU takes multiple clock cycles to execute an instruction. Once it has fetched the instruction, which takes maybe one or two cycles, it can often execute the entire instruction without further memory access (unless it is an instruction which itself access memory, such as a mov instruction with an indirect operand)
- The device performing the DMA transfer is significantly slower than the CPU speed, so the CPU will not need to halt on every instruction but just occasionally when the DMA device is accessing the memory (cycle stealing)
- CPU might also use the cache

In combination, these factors mean that the device performing the DMA transfer will have little impact on the CPU speed.

# Interesting Example of DMA being slower for RPI GPIO (no caching)

https://github.com/hzeller/rpi-gpio-dma-demo

# Steps Before DMA

- When any device has to send data between the device and the memory, the device has to send DMA request (DRQ) to DMA controller.
- The DMA controller sends Hold request (HRQ) to the CPU and waits for the CPU to assert the HLDA.
- Then the microprocessor tri-states all the data bus, address bus, and control bus. The CPU leaves the control over bus and acknowledges the HOLD request through HLDA signal.
- Now the CPU is in HOLD state and the DMA controller has to manage the operations over buses between the CPU, memory, and I/O devices.

# Steps in DMA

Device driver talks to DMA controller using the DMA API to do the following.

- Allocate DMA buffers in RAM (DMA Mapping)
- Perform DMA (with sync-ing as necessary)
- Deallocate DMA buffers in RAM (DMA Unmapping)

Unmapping is necessary as DMA address space is a shared resource and you could render the machine unusable by consuming all DMA addresses.

# DMA controller in device tree

Required property:

- #dma-cells: Must be at least 1. Used to provide DMA controller specific information. See DMA client binding below for more details.

Optional properties:

- dma-channels: Number of DMA channels supported by the controller.
- dma-requests: Number of DMA request signals supported by the controller.

Example:

```
dma: dma@48000000 {
compatible = "ti,omap-sdma";
reg = <0x48000000 0x1000>;
interrupts = <0 12 0x4
     0 13 0x4
     0 14 0x4
     0 15 0x4>;
#dma-cells = <1>;
dma-channels = <32>;
dma-requests = <127>;
};
```

# DMA client (device) in device tree

1. A device with one DMA read channel, one DMA write channel:

```
i2c1: i2c@1 {
...
dmas = <&dma 2    /* read channel */
&dma 3>;/* write channel */
dma-names = "rx", "tx";
...
};
```

2. A single read-write channel with three alternative DMA controllers:

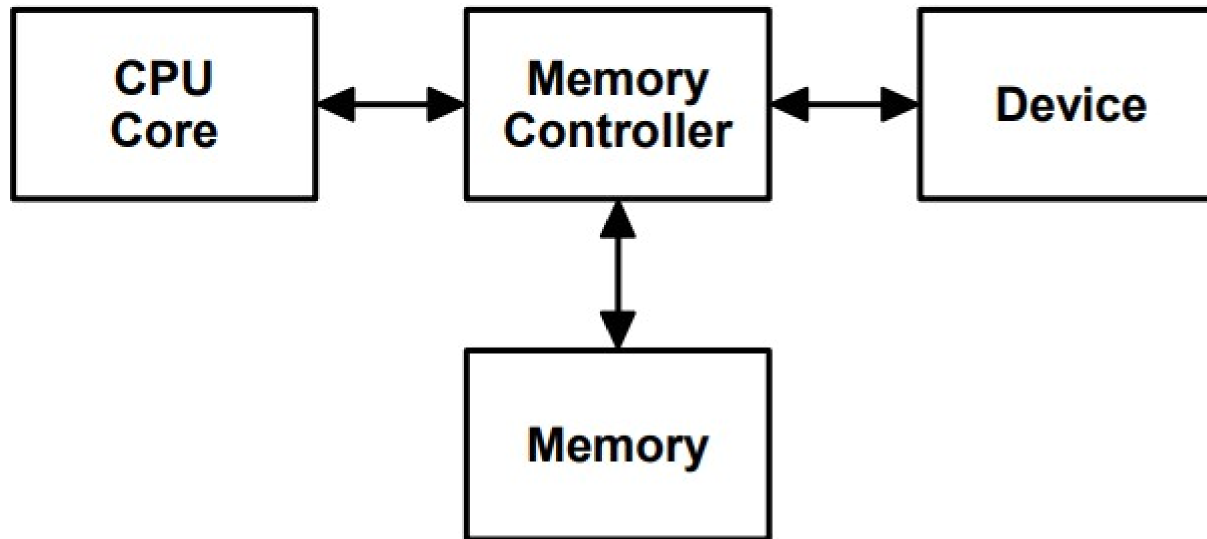
```
dmas = <&dma1 5
&dma2 7
&dma3 2>;
dma-names = "rx-tx", "rx-tx", "rx-tx";
```

3. A device with three channels, one of which has two alternatives:

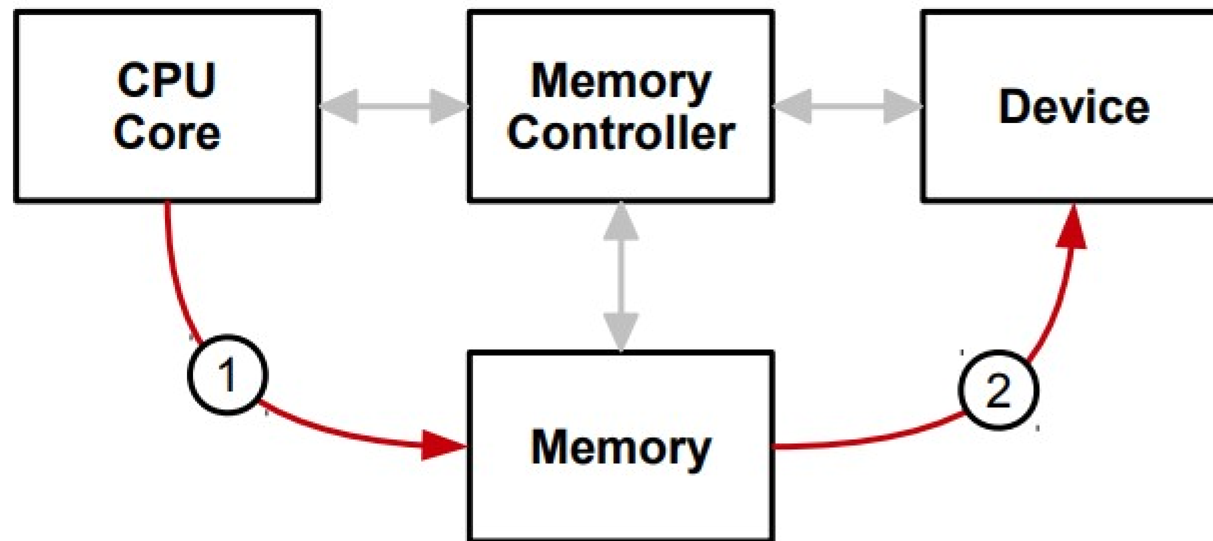
```
dmas = <&dma1 2         /* read channel */
&dma1 3        /* write channel */
&dma2 0        /* error read */
&dma3 0>;      /* alternative error read */
dma-names = "rx", "tx", "error", "error";
```

# **Cache Coherency** Decides DMA Buffer Type and Sync Requirements

# Simple Case

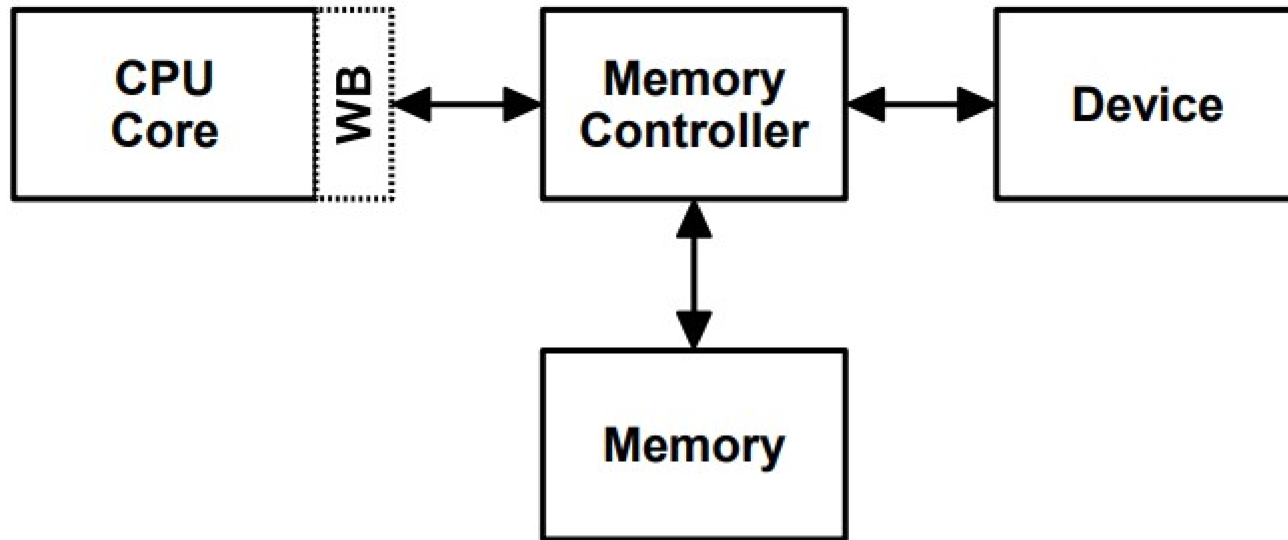


# Simple Case

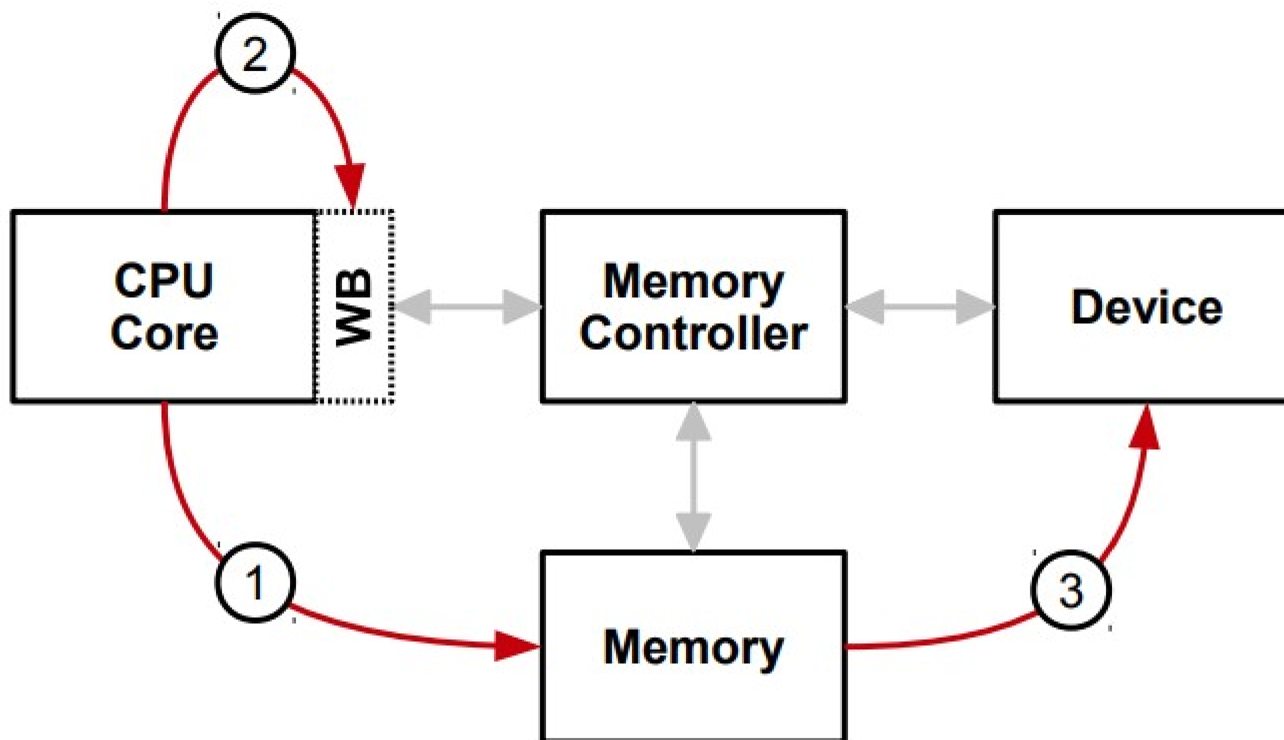



(1) CPU writes to memory
(2) Device reads from memory

# Write Buffer

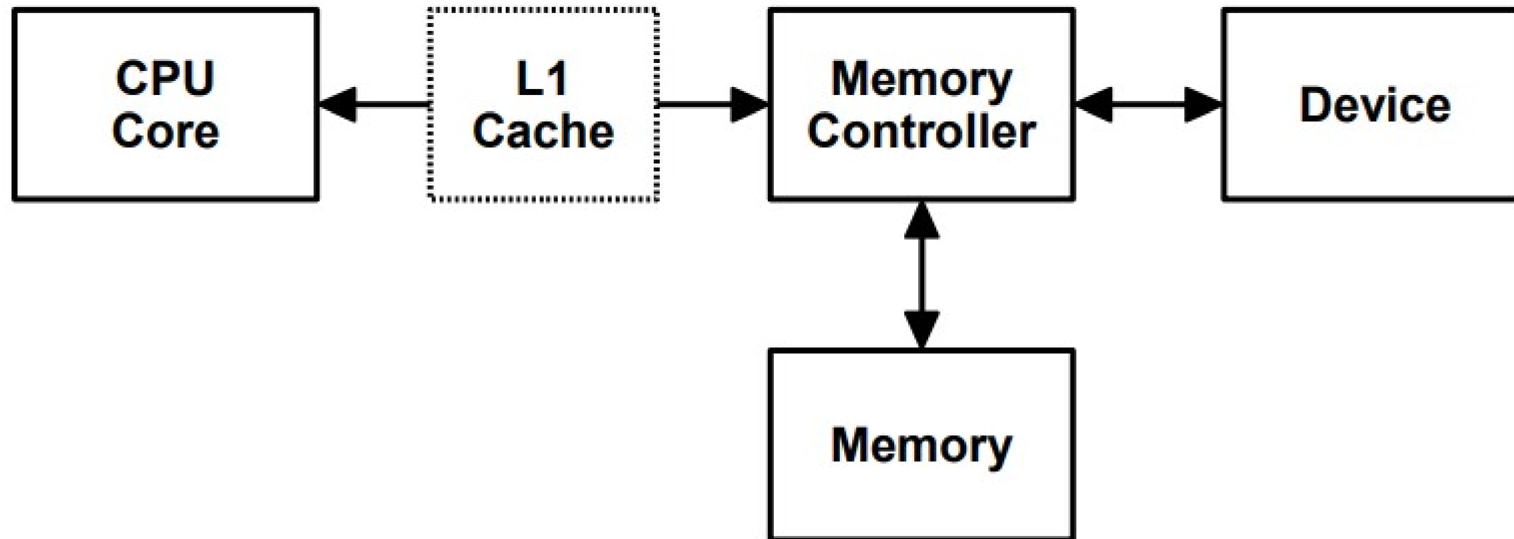


# Write Buffer

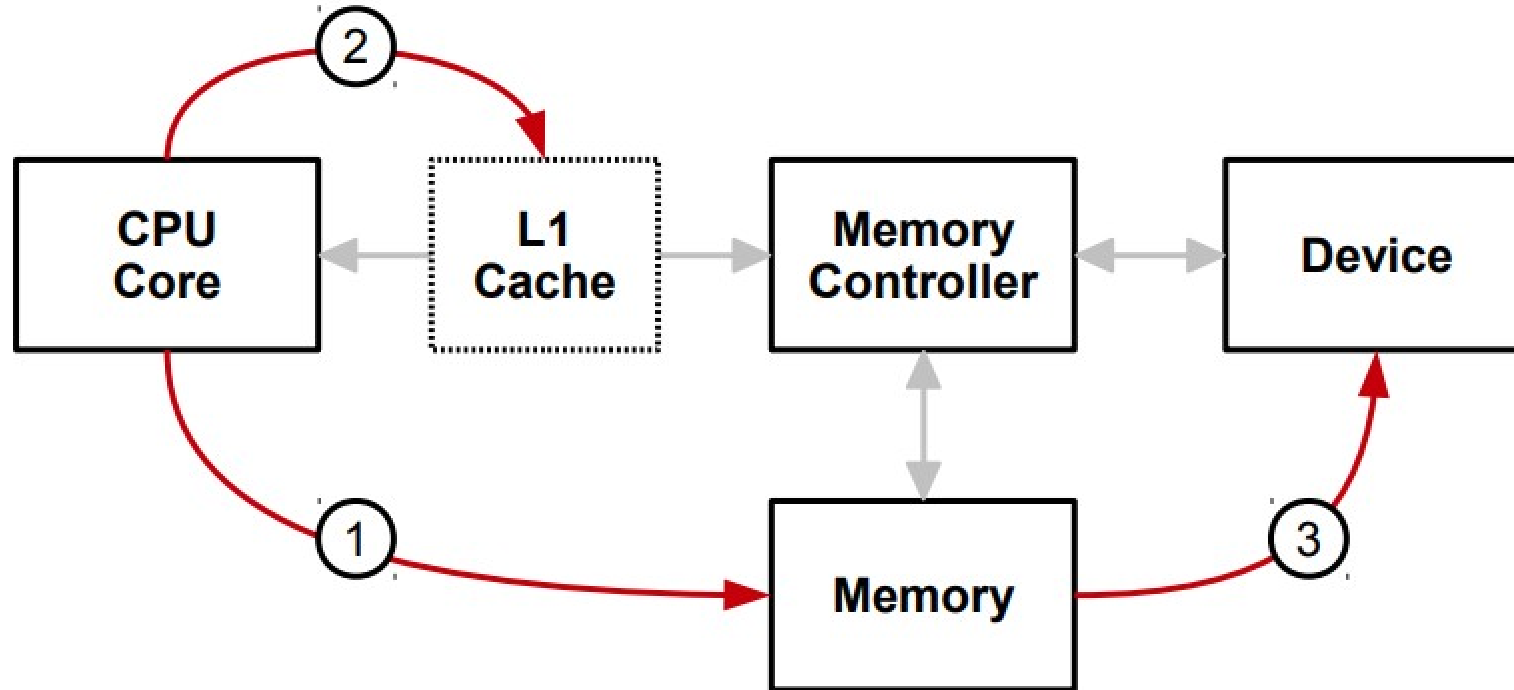



(1) CPU writes to memory
(2) CPU flushes its write buffers
(3) Device reads from memory

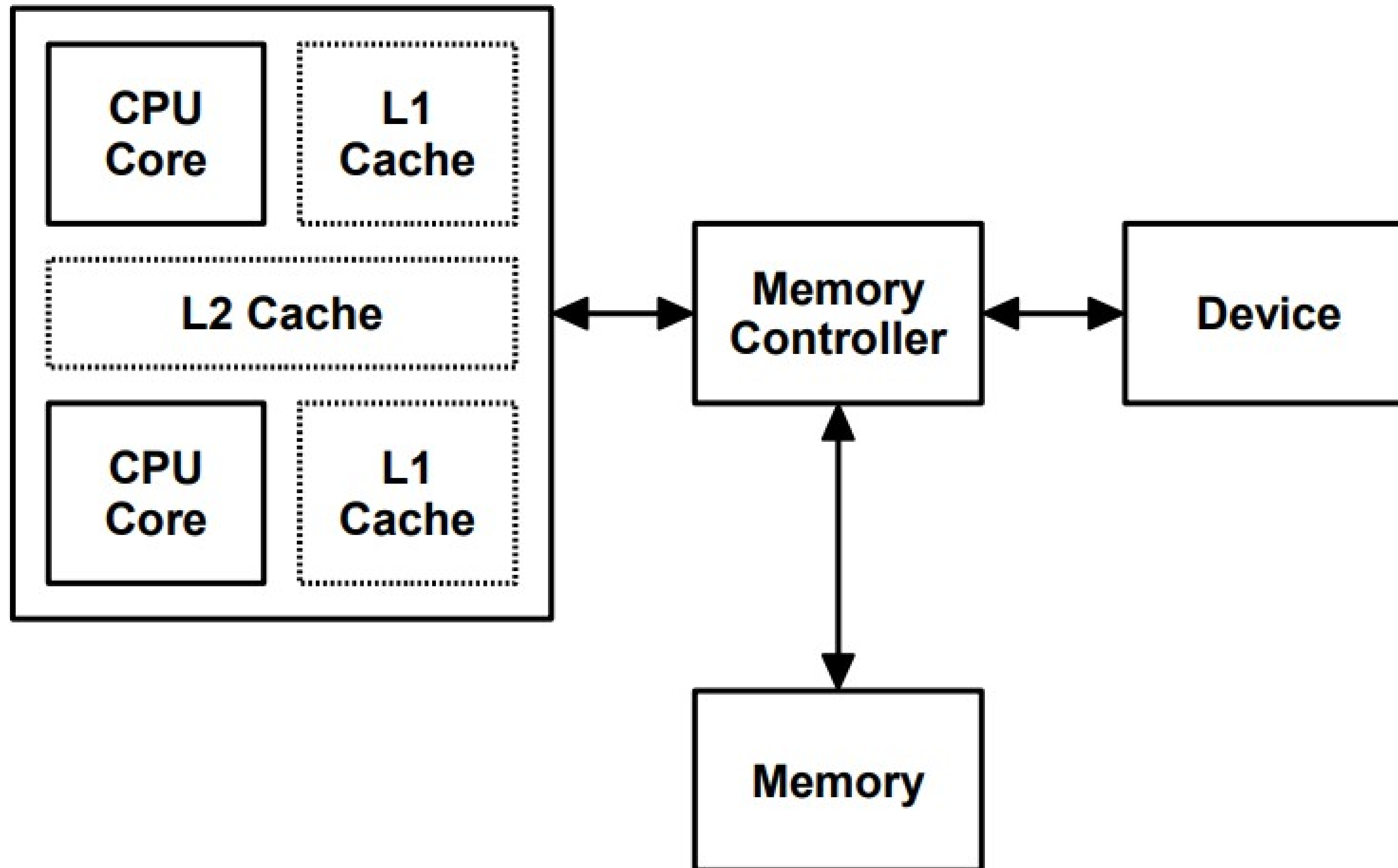
# L1 Cache



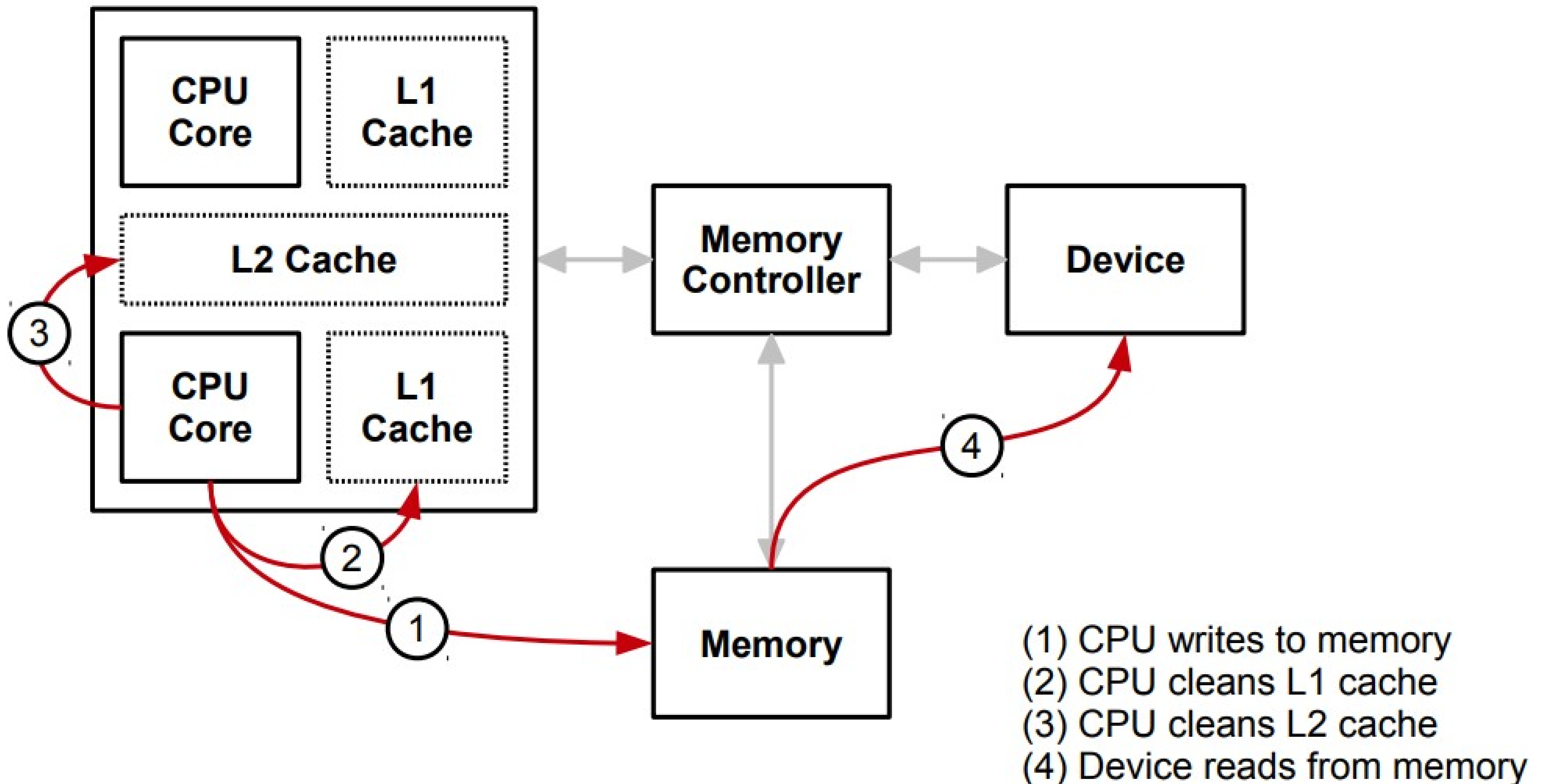
# L1 Cache



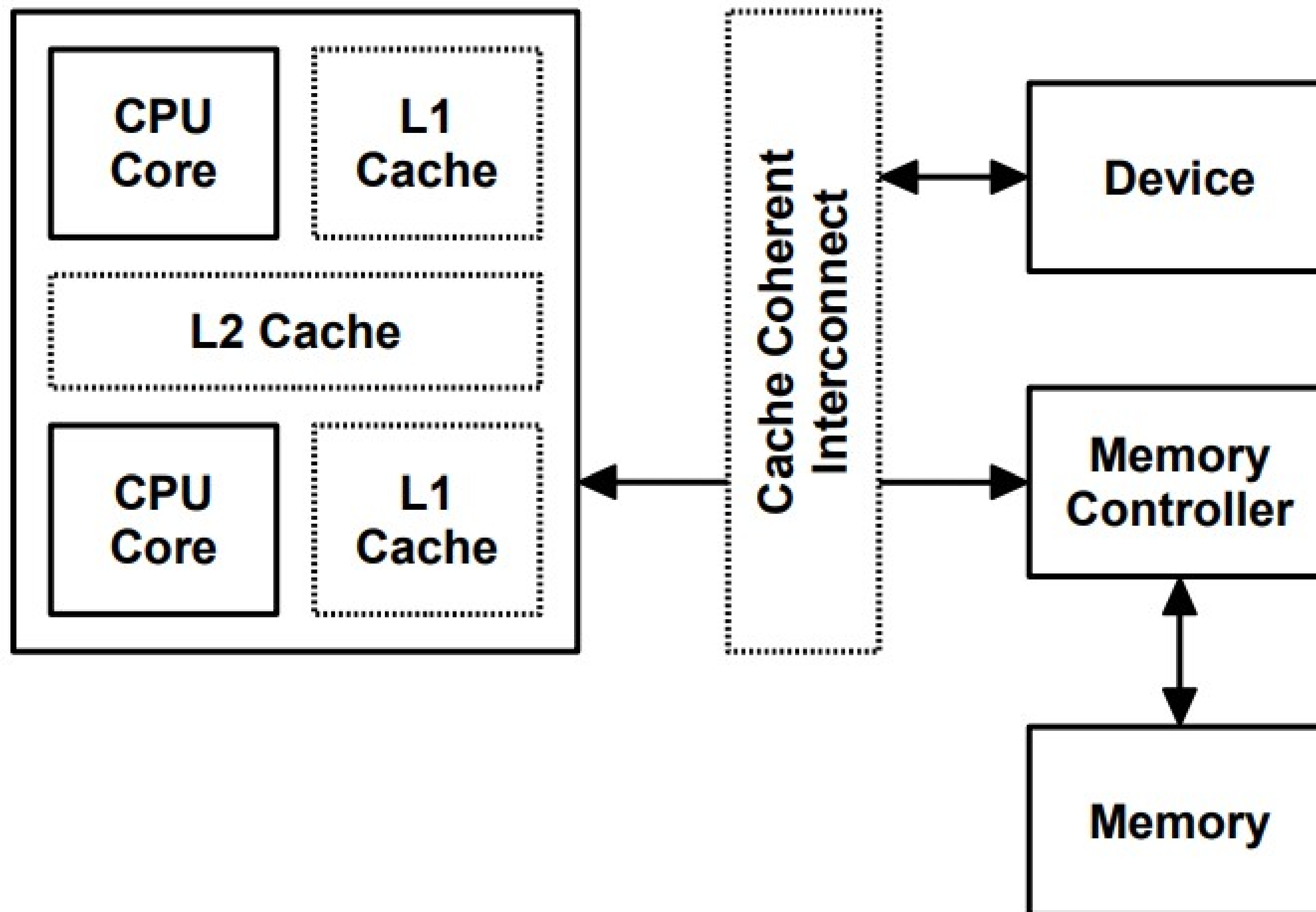
# L2 Cache



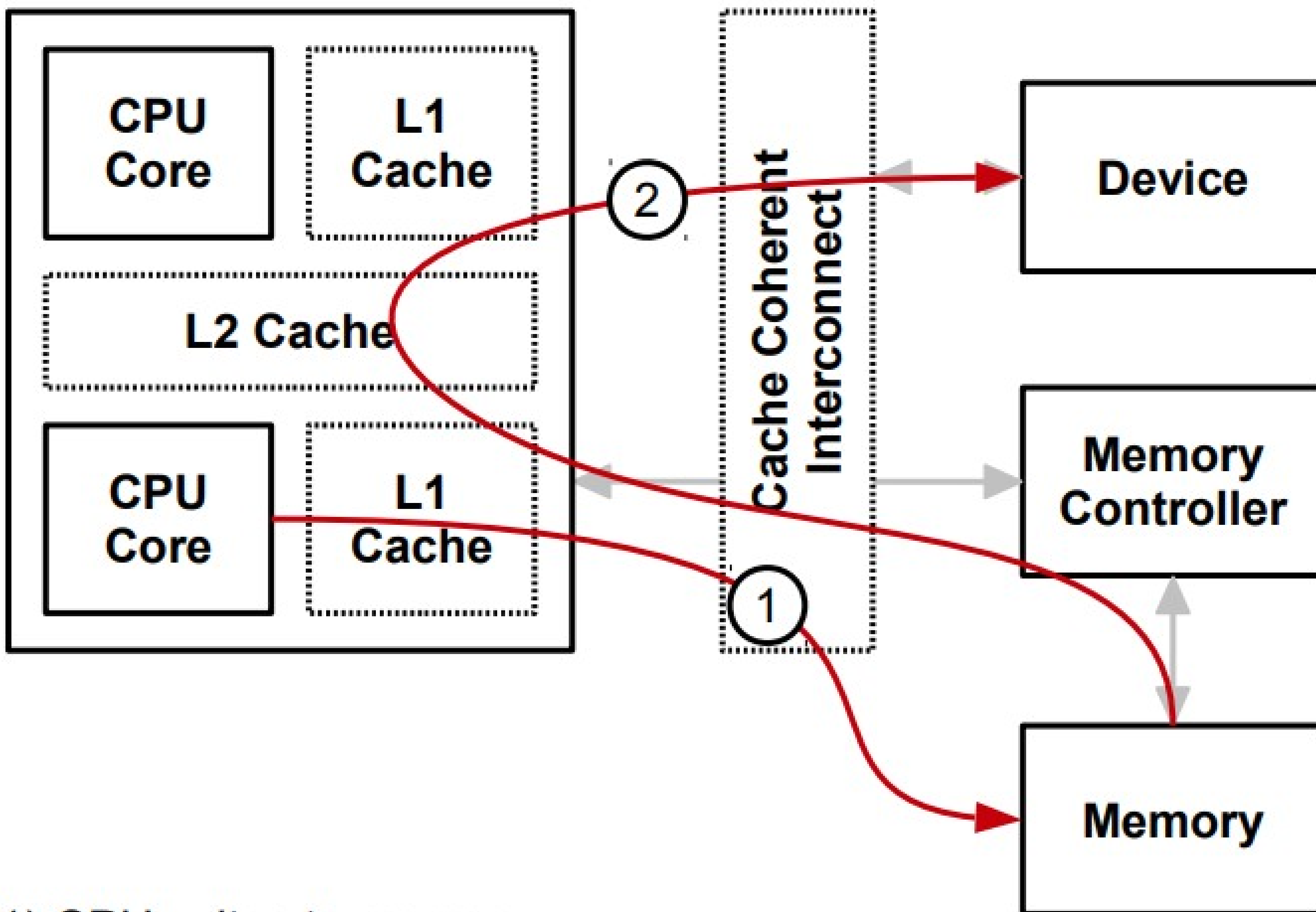
# L2 Cache



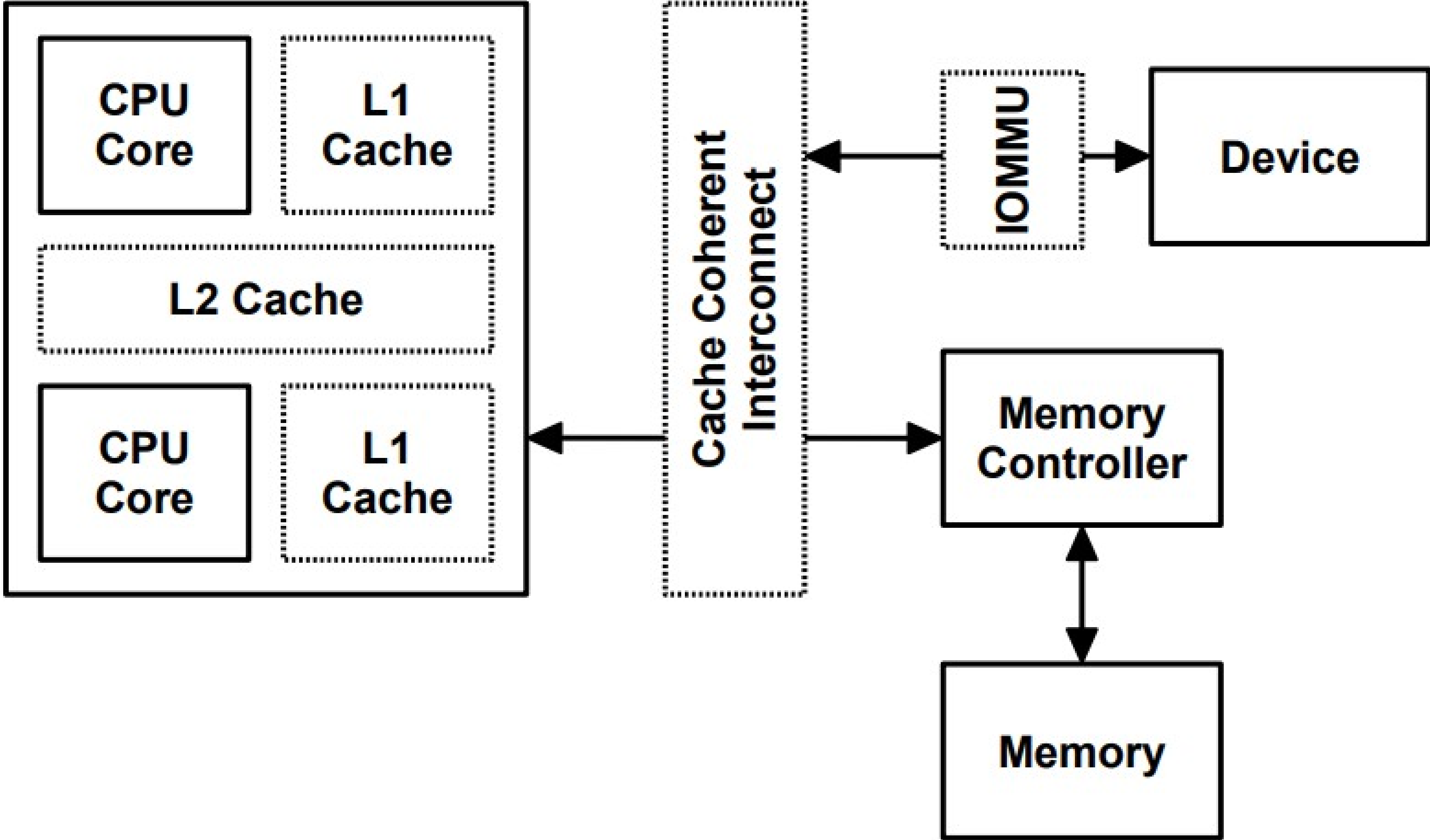
# Cache Coherent Interconnect



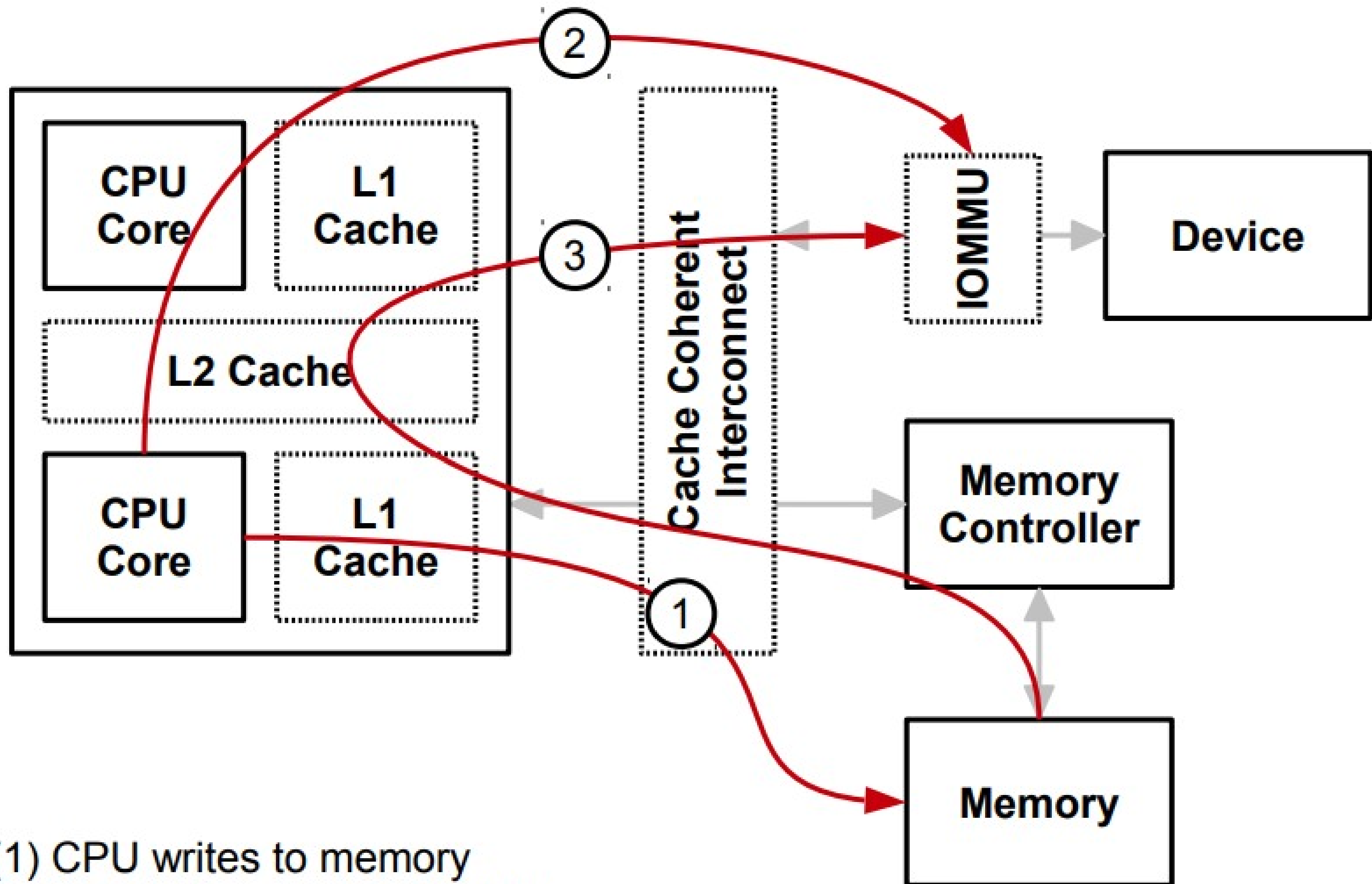
# Cache Coherent Interconnect



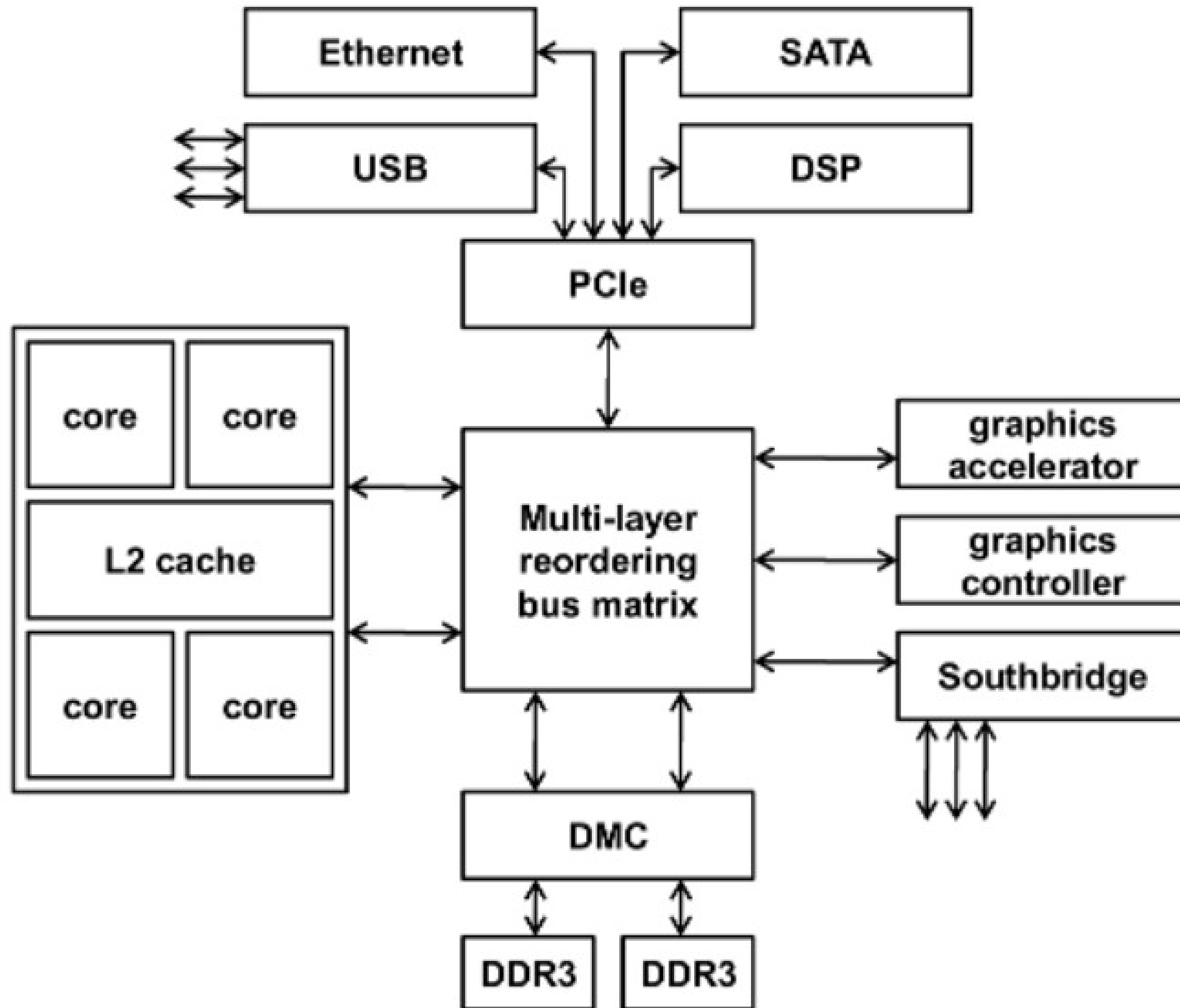
# IOMMU



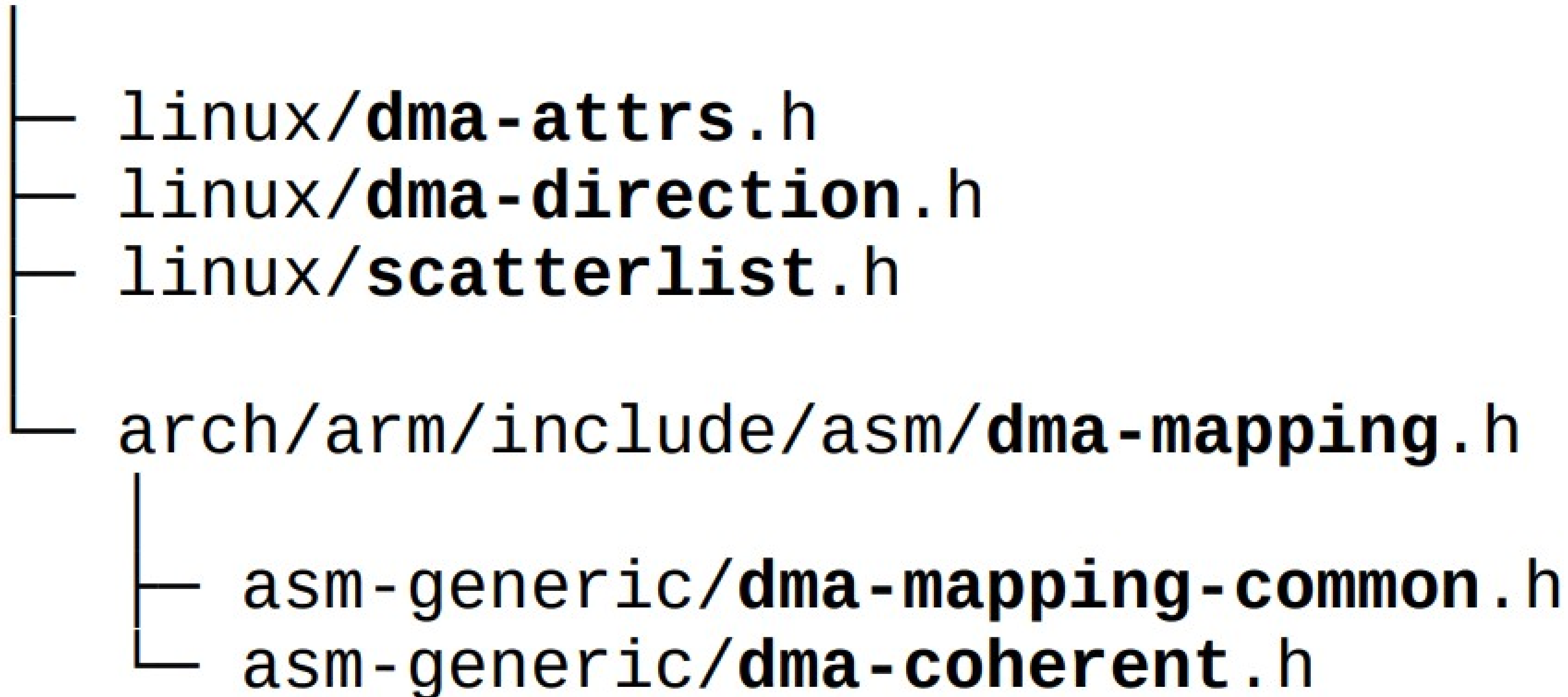
# IOMMU



# More Complex Architecture



# ARM Specific Mappings



# Buffer Types Based on Cache Coherency

- Coherent/Consistent DMA mapping
  - — Usually long lasting.
  - — Can be accessed by both ends.
  - — No-caching or uses cache coherent interconnect hardware*
  - — At least page sized
- Streaming DMA mapping
  - — Usually singly used and freed.
  - — Architecture/Platform optimized.
  - — Direction must be defined explicitly.

# DMA Direction

- DMA_BIDIRECTIONAL
- DMA_TO_DEVICE
- DMA_FROM_DEVICE
- DMA_NONE

For Networking drivers,

- For transmit packets, map/unmap them with the DMA_TO_DEVICE direction specifier.
- For receive packets, map/unmap them with the DMA_FROM_DEVICE direction specifier.

Streaming mappings specify a direction.

Coherent mappings implicitly have a direction attribute setting of DMA_BIDIRECTIONAL.

# Coherent DMA Mappings

- dma_addr_t dma_handle;
- cpu_addr = dma_alloc_coherent(dev, size, &dma_handle, gfp);
  - returns two values:
    - the virtual address which you can use to access it from the CPU
    - dma_handle which you pass to thecard.
- Acts similar to __get_free_pages(), use pool option if memory size smaller than a page is needed
- dma_free_coherent(dev, size, cpu_addr, dma_handle);

# DMA pool API

- If your driver needs lots of smaller memory regions, you can
  - write custom code to subdivide pages returned by dma_alloc_coherent()
  - use the dma_pool API to do that

- struct dma_pool *pool;
- pool = dma_pool_create(name, dev, size, align, boundary);
- cpu_addr = dma_pool_alloc(pool, flags, &dma_handle);
- dma_pool_free(pool, cpu_addr, dma_handle);
- dma_pool_destroy(pool);

# Streaming DMA Mappings

- Map Single Region

```
struct device *dev = &my_dev->dev;
dma_addr_t dma_handle;
void *addr = buffer->ptr;
size_t size = buffer->len;

dma_handle = dma_map_single(dev, addr, size, direction);
if (dma_mapping_error(dev, dma_handle)) {
/* reduce current DMA mapping usage, delay and try again later or reset driver. */
    goto map_error_handling;
}
```

- Unmap Single Region

```
dma_unmap_single(dev, dma_handle, size, direction);
```

# Streaming DMA Mappings

- Map Scatterlist

```
int i, count = dma_map_sg(dev, sglist, nents, direction);
struct scatterlist *sg;

for_each_sg(sglist, sg, count, i) {
    hw_address[i] = sg_dma_address(sg);
    hw_len[i] = sg_dma_len(sg);
}
```

- Unmap Scatterlist

```
dma_unmap_sg(dev, sglist, nents, direction);
```

# Explicit Cache Coherency for Stream Mapping

If you need to use the same streaming DMA region multiple times and touch the data in between the DMA transfers, the buffer needs to be synced properly in order for the CPU and device to see the most up-to-date and correct copy of the DMA buffer.

So, firstly, just map it with dma_map_{single,sg}(), and after each DMA transfer call either::

```
dma_sync_single_for_cpu(dev, dma_handle, size, direction);
```

or::

```
dma_sync_sg_for_cpu(dev, sglist, nents, direction);
```

Then, if you wish to let the device get at the DMA area again, finish accessing the data with the CPU, and then before actually giving the buffer to the hardware call either::

```
dma_sync_single_for_device(dev, dma_handle, size, direction);
```

or::

```
dma_sync_sg_for_device(dev, sglist, nents, direction);
```

# Explicit Sync Example in Network Card

```
my_card_setup_receive_buffer(struct my_card *cp, char *buffer, int len)
{
dma_addr_t mapping;

mapping = dma_map_single(cp->dev, buffer, len, DMA_FROM_DEVICE);

if (dma_mapping_error(cp->dev, mapping)) {
/* reduce current DMA mapping usage, delay and try again later or reset driver. */
        goto map_error_handling;
}

cp->rx_buf = buffer;
cp->rx_len = len;
cp->rx_dma = mapping;

give_rx_buf_to_card(cp);
}
```

# Explicit Sync Example in Network Card

```
my_card_interrupt_handler(int irq, void *devid, struct pt_regs *regs)
{
struct my_card *cp = devid;

...
if (read_card_status(cp) == RX_BUF_TRANSFERRED) {
struct my_card_header *hp;

/* Examine the header to see if we wish to accept the data.  But synchronize the DMA transfer with the CPU
first so that we see updated contents. */
dma_sync_single_for_cpu(&cp->dev, cp->rx_dma, cp->rx_len, DMA_FROM_DEVICE);

/* Now it is safe to examine the buffer. */
hp = (struct my_card_header *) cp->rx_buf;
if (header_is_ok(hp)) {
    dma_unmap_single(&cp->dev, cp->rx_dma, cp->rx_len, DMA_FROM_DEVICE);
    pass_to_upper_layers(cp->rx_buf);
    make_and_setup_new_rx_buf(cp);
} else {
/* CPU should not write to DMA_FROM_DEVICE-mapped area, so dma_sync_single_for_device() is not
needed here. It would be required for DMA_BIDIRECTIONAL mapping if the memory was modified. */
give_rx_buf_to_card(cp);
}
}
}
```

# Need for separate map and unmap functions: Memory Address Translations

- If the device supports DMA, if the driver sets up a buffer using kmalloc(), it will get a virtual address (X).
- The virtual memory system maps X to a physical address (Y) in system RAM.
- The driver can use virtual address X to access the buffer, but the device itself cannot because DMA doesn't go through the CPU virtual memory system.

# Need for separate map and unmap functions: Memory Address Translations

- In some simple systems, the device can do DMA directly to physical address Y.
- But in many others, there is IOMMU hardware that translates DMA addresses to physical addresses, e.g., it translates Z to Y.
- This is part of the reason for the DMA API:
  - the driver can give a virtual address X to an interface like dma_map_single()
  - it sets up any required IOMMU mapping and returns the DMA address Z
  - the driver then tells the device to do DMA to Z
  - the IOMMU maps it to the buffer at address Y in system RAM.

# Addressing Limitations of Device

- Is your device only capable of driving the low order 24-bits of address? If so, you need to inform the kernel of this fact.
- By default, the kernel assumes that your device can address the full 32-bits.
- For a 64-bit capable device, this needs to be increased.

# Device Probe

- Interrogate kernel in probe() to see if the DMA controller on the machine can properly support the DMA addressing limitation your device has.
  - int dma_set_mask_and_coherent(struct device *dev, u64 mask);
  - int dma_set_mask(struct device *dev, u64 mask);
  - int dma_set_coherent_mask(struct device *dev, u64 mask);

# Parameters and Return Values

- Params
  - dev is a pointer to the device struct of your device
  - mask is a bit mask describing which bits of an address your device supports.
- Return Value
  - zero if your card can perform DMA properly on the machine given the address mask you provided
  - non-zero, your device cannot perform DMA properly on this platform, and attempting to do so will result in undefined behavior. You must either use a different mask, or not use DMA.

# Probe Examples

- 32 bit addressing support
  - if (dma_set_mask_and_coherent(dev, DMA_BIT_MASK(32))) {

```
    dev_warn(dev, "mydev: No suitable DMA available\n");
    goto ignore_this_device;
}
```

- 64 bit addressing support
  - int using_dac;

```
if (!dma_set_mask(dev, DMA_BIT_MASK(64))) {
        using_dac = 1;
} else if (!dma_set_mask(dev, DMA_BIT_MASK(32))) {
        using_dac = 0;
} else {
        dev_warn(dev, "mydev: No suitable DMA available\n");
        goto ignore_this_device;
}
```

# Probe example of sound card

```
#define PLAYBACK_ADDRESS_BITS   DMA_BIT_MASK(32)
#define RECORD_ADDRESS_BITS  DMA_BIT_MASK(24)

struct my_sound_card *card;
struct device *dev;


...
if (!dma_set_mask(dev, PLAYBACK_ADDRESS_BITS)) {
card->playback_enabled = 1;
} else {
card->playback_enabled = 0;
dev_warn(dev, "%s: Playback disabled due to DMA limitations\n",
       card->name);
}
if (!dma_set_mask(dev, RECORD_ADDRESS_BITS)) {
card->record_enabled = 1;
} else {
card->record_enabled = 0;
dev_warn(dev, "%s: Record disabled due to DMA limitations\n",
       card->name);
}
```