Concurrent Computing (Operating Systems)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨Daniel.Page@bristol.ac.uk⟩

February 10, 2016

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

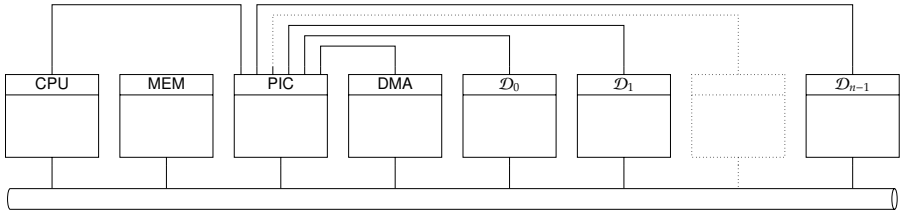
Notes:

Continued from last lecture ...

Notes:

COMS20001 lecture: week #14

- Recall: our goal is to flesh out some details wrt.



noting that

1. the top-most connections model the interrupt mechanism, and
2. the bottom-most connections model a communication **bus**.

Notes:

Concepts (1)

Definition (**bus**)

A **bus** is basically just a structured set of wires, allowing communication between one or more attached components:

- any subset of the total **bus width** w may be classed as a
  - **control bus** which communicate control or signalling information,
  - **address bus** which communicate addresses, or
  - **data bus** which communicate data
- each access (or operation) must adhere to a **bus protocol**, and occurs during a **bus cycle** which may be
  - synchronous, implying a clock and hence a **bus frequency** which governs the (fixed) length of each bus cycle, or
  - asynchronous, implying a need for extra control signals, an potentially a variable-length bus cycle

  and
- attached components may be classed as
  - an active **bus master**, which can both transmit and receive via the bus, or
  - a passive **bus slave**, which can only receive via the bus.

Notes:

- Focusing on the bus itself, the distinction between addresses and data is fairly loose: *everything* the bus communicates could be thought of as data, with the attached components interpreting that however they want. However, when discussing how components are connected the terms become more useful.
- When the bus width $w = 1$, we say this is a **serial bus** because at a given point in time it can communicate at most 1 bit; where $w > 1$ the bus is a **parallel bus**, because all w bits can be communicated at the same time.
- Hopefully the need for a bus protocol is clear even if there are only two end-point components attached to it (e.g., a processor attached to a memory). When *more* components are added, the issue of shared access becomes harder to deal with: the bus protocol is tasked with managing access, st. two components cannot simultaneously transmit for example (which is problematic, since they will try to do so using the same physical wires).
- You might see the terms **internal bus** and **external bus** used as a further form of classification. Both are loosely defined, and obviously depend on the context and hence a point of reference: they could be used to mean internal or external to a processor, or wider system for example. As a rule of thumb, however, it is common to associate the term external bus with some form of expansion mechanism, e.g., a PCI bus which allows expansion cards (which are just components) to communicate with and hence extend the functionality offered by the processor alone.
- The term bus cycle can be equated to an instruction (or machine cycle) in the context of processor design, which describes the period in which one instruction is executed (i.e., goes through a fetch-decode-execute cycle). That is, both capture the duration of one operation (wrt. the bus, or processor). However, it is important to note that the two are not necessarily *equal* even if the bus and processor discussed are attached: it is common, and useful, for the processor to operate at a different frequency than the bus (which connects it to other components) might.
- If the bus connects to a memory, it is common to use the terms **write cycle** (or **store cycle**) and **read cycle** (or **load cycle**) in place of the more general reference to transmitting and receiving. In fact, this is also common when memory it not involved per se: focusing on the bus, components could be said to write to it (i.e., transmit) and read from it (i.e., receive) even if such terms arguably mask the fact it is a communication not storage medium.

Concepts (2)

Definition (**device controller**)

A given hardware **device** (or **peripheral**) may be composed from two parts, namely

1. the electronic or electro-mechanical **device mechanism** (or innards, e.g., the physical disk, drive motors, read/write head), and
2. the **device controller**, which offers a high-level electronic interface via one or more **device registers**.

Definition (**block device** and **character device**)

A given device is typically and imperfectly classified as either

1. a **block device** where
   - random access to data is via addressable multi-byte blocks, and
   - data may be cached or buffered,
2. a **character device** where
   - sequential access to data is via a non-addressable byte stream, and
   - data is not cached or buffered

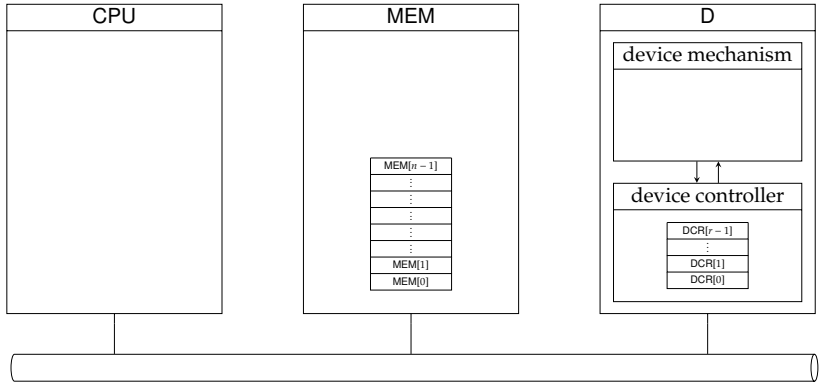
   or
3. a **network device**.

Notes:

- Various aspects of this classification could be described as imperfect. For example, random vs. sequential access order and byte vs. multi-byte block access type should be viewed as orthogonal features in theory; in the same way, the decision to cache data or not adds another orthogonal feature. Read it therefore as *a* not *the* classification, but one used by the majority of kernels you will encounter.
- There are various implications for treating a device as either block- or character-based. For example, the former may support the `seek` system call st. the access point can be moved backward or forward (this is essentially what allows random access) whereas the latter will not; the latter probably supports `getc` and `putc`, but not a lot else.
- It *can* be useful to distinguish between data- and control-related device registers: we ignore the distinction here, assuming the former is implied when discussing access to data, although they are sometimes interfaced with differently.

Concepts (3)

- Idea: device registers are treated as sort of pseudo-memories



yielding a design space wrt. (at least)

1. single or multiple address spaces, and
2. single or multiple buses

and hence either **memory-mapped I/O** or **port-mapped I/O**.

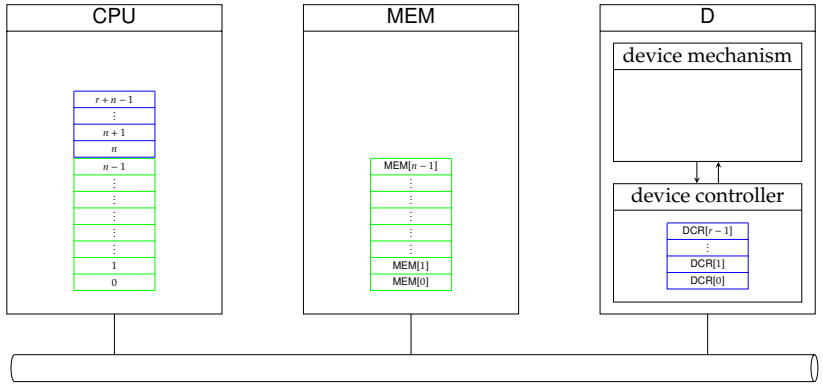
Notes:

- You could argue the device, as illustrated, is missing a component that acts as the **bus interface**. In reality this *is* likely to be a separate component, but we ignore that here and assume the connection to the bus always includes this.
- An obvious feature of this architecture is that the device controller can offer a high-level, fairly generic interface to the underlying mechanism; in contrast, the interface between device controller and mechanism is at least lower-level, and more specific. For example, a disk controller can offer a uniform interface to differing disk mechanisms (e.g., which have different geometries, or capacities), and hide complications such as the use (or not) of error detection and/or correction.
- The choice between memory-mapped and port mapped I/O must be based on a range of trade-offs:
  - Use of port-mapped I/O typically demands special-purpose instructions (to move data between the port and general-purpose registers), whereas memory-mapped I/O can use standard memory access instructions.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms (e.g., the MMU will naturally protect access to a mapped address by an unprivileged instruction), whereas port-mapped I/O may not: especially where multiple buses are used, there might be a need for a dedicated I/O MMU.
  - In using memory-mapped I/O, it is obviously important that mapped addresses are protected from use as "normal" memory even though they appear, by design, in the accessible address space. This is easier to solve than it seems: the MMU offers a natural solution via virtual memory.
  - Use of a single bus implies that all devices connected to it must operate at or above the bus frequency: if they didn't (i.e., their clock frequency was *lower* than the bus frequency) they would be unable to access it.
  - Unless a single bus is wide or has a high frequency, it will often provide less communication bandwidth than multiple buses. That is, it provides a less performant solution: less communication can potentially happen per unit of time, in part because more devices share access.
  - Access to memory typically assumes there are no side-effects (bar an update to the content). With memory-mapped I/O, however, an address is mapped to a device register whose content may be updated by the device independently from any loads and stores. This has (at least) two implications, namely a) the mapped address might be marked using the `volatile` keyword (in C) to signal this fact, b) so-called barrier instructions might be required to ensure any pending load or store instructions are completed (in the correct order), and, perhaps most importantly, c) there needs to be a mechanism to bypass or cope with effects of any caches (since cached data may not reflect the actual device register state): the MMU offers a natural solution via virtual memory. With port-mapped I/O, especially when the address space is segregated and there is a dedicated I/O bus, this may not be as problematic. For example, using special-purpose instructions explicitly hints (to the compiler) at `volatile`-esque content.
  - Use of a single bus is simpler, both in terms of the resources needed plus the ease by which *all* attached devices get a consistent view of access activity. This is harder when multiple buses are used, because devices attached to one bus somehow need to see activity on the other. Imagine for example that memory-mapped I/O is used, but there is one I/O bus and a separate memory bus. When the processor accesses an address in memory, the memory bus is used. But if an address mapped to a device register is used, the device has to see this access if it is to act correctly.

  Even then, the terminology used is a little inexact because various hybrid options are possible. For example, x86 opts for port-mapped I/O (by using `inb` and `outb` instructions, plus variants, to either receive and transmit bytes to a given port). However, although there is a separate port address space, this is for control-oriented device registers: data-oriented device registers (e.g., buffers) are memory-mapped instead.
- As a result of the trade-offs involved, one can easily identify examples of any combination of design choices. For instance, the PDP-8 computer used a port-mapped approach while the later PDP-11 was instrumental in introducing memory-mapped I/O.

Concepts (3)

- Idea: device registers are treated as sort of pseudo-memories



yielding a design space wrt. (at least)

1. single or multiple address spaces, and
2. single or multiple buses

and hence either **memory-mapped I/O** or **port-mapped I/O**.

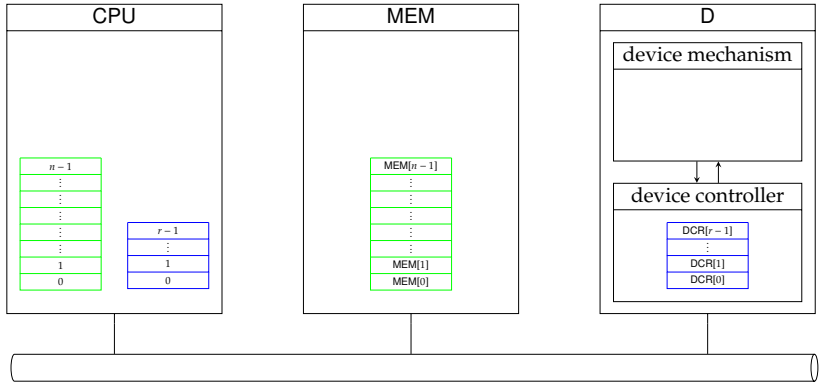
Notes:

- You could argue the device, as illustrated, is missing a component that acts as the **bus interface**. In reality this *is* likely to be a separate component, but we ignore that here and assume the connection to the bus always includes this.
- An obvious feature of this architecture is that the device controller can offer a high-level, fairly generic interface to the underlying mechanism; in contrast, the interface between device controller and mechanism is at least lower-level, and more specific. For example, a disk controller can offer a uniform interface to differing disk mechanisms (e.g., which have different geometries, or capacities), and hide complications such as the use (or not) of error detection and/or correction.
- The choice between memory-mapped and port mapped I/O must be based on a range of trade-offs:
  - Use of port-mapped I/O typically demands special-purpose instructions (to move data between the port and general-purpose registers), whereas memory-mapped I/O can use standard memory access instructions.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms (e.g., the MMU will naturally protect access to a mapped address by an unprivileged instruction), whereas port-mapped I/O may not: especially where multiple buses are used, there might be a need for a dedicated I/O MMU.
  - In using memory-mapped I/O, it is obviously important that mapped addresses are protected from use as "normal" memory even though they appear, by design, in the accessible address space. This is easier to solve than it seems: the MMU offers a natural solution via virtual memory.
  - Use of a single bus implies that all devices connected to it must operate at or above the bus frequency: if they didn't (i.e., their clock frequency was *lower* than the bus frequency) they would be unable to access it.
  - Unless a single bus is wide or has a high frequency, it will often provide less communication bandwidth than multiple buses. That is, it provides a less performant solution: less communication can potentially happen per unit of time, in part because more devices share access.
  - Access to memory typically assumes there are no side-effects (bar an update to the content). With memory-mapped I/O, however, an address is mapped to a device register whose content may be updated by the device independently from any loads and stores. This has (at least) two implications, namely a) the mapped address might be marked using the `volatile` keyword (in C) to signal this fact, b) so-called barrier instructions might be required to ensure any pending load or store instructions are completed (in the correct order), and, perhaps most importantly, c) there needs to be a mechanism to bypass or cope with effects of any caches (since cached data may not reflect the actual device register state): the MMU offers a natural solution via virtual memory. With port-mapped I/O, especially when the address space is segregated and there is a dedicated I/O bus, this may not be as problematic. For example, using special-purpose instructions explicitly hints (to the compiler) at `volatile`-esque content.
  - Use of a single bus is simpler, both in terms of the resources needed plus the ease by which *all* attached devices get a consistent view of access activity. This is harder when multiple buses are used, because devices attached to one bus somehow need to see activity on the other. Imagine for example that memory-mapped I/O is used, but there is one I/O bus and a separate memory bus. When the processor accesses an address in memory, the memory bus is used. But if an address mapped to a device register is used, the device has to see this access if it is to act correctly.

  Even then, the terminology used is a little inexact because various hybrid options are possible. For example, x86 opts for port-mapped I/O (by using `inb` and `outb` instructions, plus variants, to either receive and transmit bytes to a given port). However, although there is a separate port address space, this is for control-oriented device registers: data-oriented device registers (e.g., buffers) are memory-mapped instead.
- As a result of the trade-offs involved, one can easily identify examples of any combination of design choices. For instance, the PDP-8 computer used a port-mapped approach while the later PDP-11 was instrumental in introducing memory-mapped I/O.

Concepts (3)

- Idea: device registers are treated as sort of pseudo-memories



yielding a design space wrt. (at least)

1. single or multiple address spaces, and
2. single or multiple buses

and hence either **memory-mapped I/O** or **port-mapped I/O**.

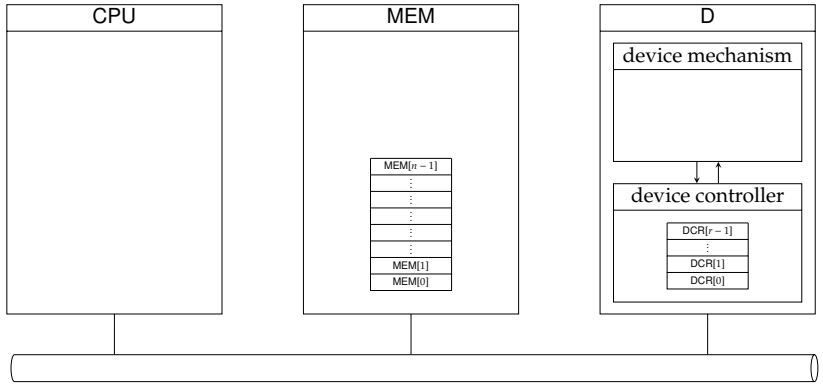
Notes:

- You could argue the device, as illustrated, is missing a component that acts as the **bus interface**. In reality this *is* likely to be a separate component, but we ignore that here and assume the connection to the bus always includes this.
- An obvious feature of this architecture is that the device controller can offer a high-level, fairly generic interface to the underlying mechanism; in contrast, the interface between device controller and mechanism is at least lower-level, and more specific. For example, a disk controller can offer a uniform interface to differing disk mechanisms (e.g., which have different geometries, or capacities), and hide complications such as the use (or not) of error detection and/or correction.
- The choice between memory-mapped and port mapped I/O must be based on a range of trade-offs:
  - Use of port-mapped I/O typically demands special-purpose instructions (to move data between the port and general-purpose registers), whereas memory-mapped I/O can use standard memory access instructions.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms (e.g., the MMU will naturally protect access to a mapped address by an unprivileged instruction), whereas port-mapped I/O may not: especially where multiple buses are used, there might be a need for a dedicated I/O MMU.
  - In using memory-mapped I/O, it is obviously important that mapped addresses are protected from use as "normal" memory even though they appear, by design, in the accessible address space. This is easier to solve than it seems: the MMU offers a natural solution via virtual memory.
  - Use of a single bus implies that all devices connected to it must operate at or above the bus frequency: if they didn't (i.e., their clock frequency was *lower* than the bus frequency) they would be unable to access it.
  - Unless a single bus is wide or has a high frequency, it will often provide less communication bandwidth than multiple buses. That is, it provides a less performant solution: less communication can potentially happen per unit of time, in part because more devices share access.
  - Access to memory typically assumes there are no side-effects (bar an update to the content). With memory-mapped I/O, however, an address is mapped to a device register whose content may be updated by the device independently from any loads and stores. This has (at least) two implications, namely a) the mapped address might be marked using the `volatile` keyword (in C) to signal this fact, b) so-called barrier instructions might be required to ensure any pending load or store instructions are completed (in the correct order), and, perhaps most importantly, c) there needs to be a mechanism to bypass or cope with effects of any caches (since cached data may not reflect the actual device register state): the MMU offers a natural solution via virtual memory. With port-mapped I/O, especially when the address space is segregated and there is a dedicated I/O bus, this may not be as problematic. For example, using special-purpose instructions explicitly hints (to the compiler) at `volatile`-esque content.
  - Use of a single bus is simpler, both in terms of the resources needed plus the ease by which *all* attached devices get a consistent view of access activity. This is harder when multiple buses are used, because devices attached to one bus somehow need to see activity on the other. Imagine for example that memory-mapped I/O is used, but there is one I/O bus and a separate memory bus. When the processor accesses an address in memory, the memory bus is used. But if an address mapped to a device register is used, the device has to see this access if it is to act correctly.

  Even then, the terminology used is a little inexact because various hybrid options are possible. For example, x86 opts for port-mapped I/O (by using `inb` and `outb` instructions, plus variants, to either receive and transmit bytes to a given port). However, although there is a separate port address space, this is for control-oriented device registers: data-oriented device registers (e.g., buffers) are memory-mapped instead.
- As a result of the trade-offs involved, one can easily identify examples of any combination of design choices. For instance, the PDP-8 computer used a port-mapped approach while the later PDP-11 was instrumental in introducing memory-mapped I/O.

Concepts (3)

- Idea: device registers are treated as sort of pseudo-memories



yielding a design space wrt. (at least)

1. single or multiple address spaces, and
2. single or multiple buses

and hence either **memory-mapped I/O** or **port-mapped I/O**.

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # f12b939 @ 2016-02-08 University of BRISTOL

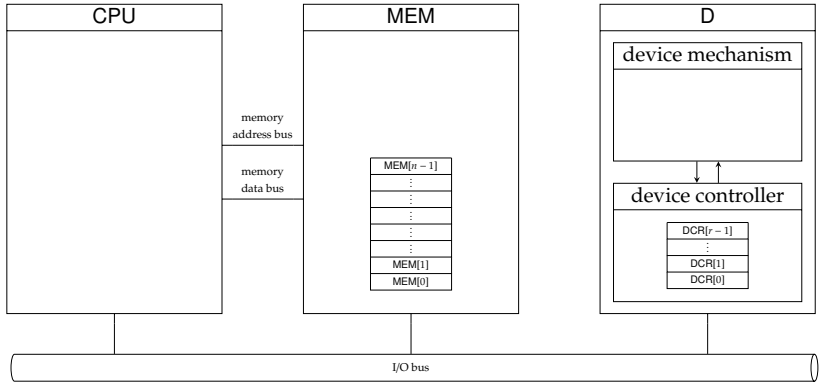
Notes:

- You could argue the device, as illustrated, is missing a component that acts as the **bus interface**. In reality this *is* likely to be a separate component, but we ignore that here and assume the connection to the bus always includes this.
- An obvious feature of this architecture is that the device controller can offer a high-level, fairly generic interface to the underlying mechanism; in contrast, the interface between device controller and mechanism is at least lower-level, and more specific. For example, a disk controller can offer a uniform interface to differing disk mechanisms (e.g., which have different geometries, or capacities), and hide complications such as the use (or not) of error detection and/or correction.
- The choice between memory-mapped and port mapped I/O must be based on a range of trade-offs:
  - Use of port-mapped I/O typically demands special-purpose instructions (to move data between the port and general-purpose registers), whereas memory-mapped I/O can use standard memory access instructions.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms (e.g., the MMU will naturally protect access to a mapped address by an unprivileged instruction), whereas port-mapped I/O may not: especially where multiple buses are used, there might be a need for a dedicated I/O MMU.
  - In using memory-mapped I/O, it is obviously important that mapped addresses are protected from use as "normal" memory even though they appear, by design, in the accessible address space. This is easier to solve than it seems: the MMU offers a natural solution via virtual memory.
  - Use of a single bus implies that all devices connected to it must operate at or above the bus frequency: if they didn't (i.e., their clock frequency was *lower* than the bus frequency) they would be unable to access it.
  - Unless a single bus is wide or has a high frequency, it will often provide less communication bandwidth than multiple buses. That is, it provides a less performant solution: less communication can potentially happen per unit of time, in part because more devices share access.
  - Access to memory typically assumes there are no side-effects (bar an update to the content). With memory-mapped I/O, however, an address is mapped to a device register whose content may be updated by the device independently from any loads and stores. This has (at least) two implications, namely a) the mapped address might be marked using the `volatile` keyword (in C) to signal this fact, b) so-called barrier instructions might be required to ensure any pending load or store instructions are completed (in the correct order), and, perhaps most importantly, c) there needs to be a mechanism to bypass or cope with effects of any caches (since cached data may not reflect the actual device register state): the MMU offers a natural solution via virtual memory. With port-mapped I/O, especially when the address space is segregated and there is a dedicated I/O bus, this may not be as problematic. For example, using special-purpose instructions explicitly hints (to the compiler) at `volatile`-esque content.
  - Use of a single bus is simpler, both in terms of the resources needed plus the ease by which *all* attached devices get a consistent view of access activity. This is harder when multiple buses are used, because devices attached to one bus somehow need to see activity on the other. Imagine for example that memory-mapped I/O is used, but there is one I/O bus and a separate memory bus. When the processor accesses an address in memory, the memory bus is used. But if an address mapped to a device register is used, the device has to see this access if it is to act correctly.

  Even then, the terminology used is a little inexact because various hybrid options are possible. For example, x86 opts for port-mapped I/O (by using `inb` and `outb` instructions, plus variants, to either receive and transmit bytes to a given port). However, although there is a separate port address space, this is for control-oriented device registers: data-oriented device registers (e.g., buffers) are memory-mapped instead.
- As a result of the trade-offs involved, one can easily identify examples of any combination of design choices. For instance, the PDP-8 computer used a port-mapped approach while the later PDP-11 was instrumental in introducing memory-mapped I/O.

Concepts (3)

- Idea: device registers are treated as sort of pseudo-memories



yielding a design space wrt. (at least)

1. single or multiple address spaces, and
2. single or multiple buses

and hence either **memory-mapped I/O** or **port-mapped I/O**.

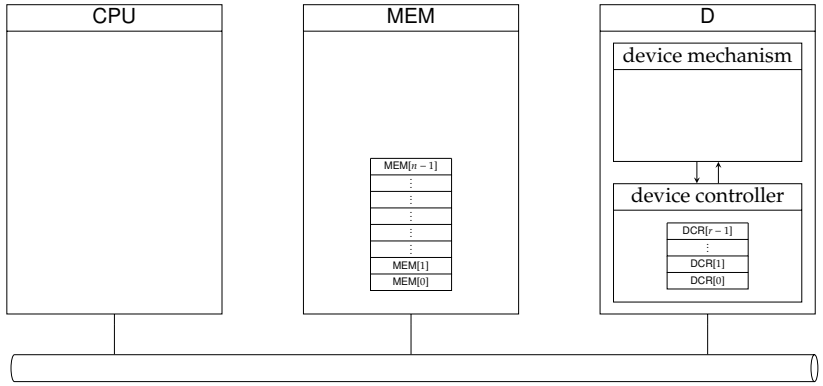
Notes:

- You could argue the device, as illustrated, is missing a component that acts as the **bus interface**. In reality this *is* likely to be a separate component, but we ignore that here and assume the connection to the bus always includes this.
- An obvious feature of this architecture is that the device controller can offer a high-level, fairly generic interface to the underlying mechanism; in contrast, the interface between device controller and mechanism is at least lower-level, and more specific. For example, a disk controller can offer a uniform interface to differing disk mechanisms (e.g., which have different geometries, or capacities), and hide complications such as the use (or not) of error detection and/or correction.
- The choice between memory-mapped and port mapped I/O must be based on a range of trade-offs:
  - Use of port-mapped I/O typically demands special-purpose instructions (to move data between the port and general-purpose registers), whereas memory-mapped I/O can use standard memory access instructions.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms (e.g., the MMU will naturally protect access to a mapped address by an unprivileged instruction), whereas port-mapped I/O may not: especially where multiple buses are used, there might be a need for a dedicated I/O MMU.
  - In using memory-mapped I/O, it is obviously important that mapped addresses are protected from use as "normal" memory even though they appear, by design, in the accessible address space. This is easier to solve than it seems: the MMU offers a natural solution via virtual memory.
  - Use of a single bus implies that all devices connected to it must operate at or above the bus frequency: if they didn't (i.e., their clock frequency was *lower* than the bus frequency) they would be unable to access it.
  - Unless a single bus is wide or has a high frequency, it will often provide less communication bandwidth than multiple buses. That is, it provides a less performant solution: less communication can potentially happen per unit of time, in part because more devices share access.
  - Access to memory typically assumes there are no side-effects (bar an update to the content). With memory-mapped I/O, however, an address is mapped to a device register whose content may be updated by the device independently from any loads and stores. This has (at least) two implications, namely a) the mapped address might be marked using the `volatile` keyword (in C) to signal this fact, b) so-called barrier instructions might be required to ensure any pending load or store instructions are completed (in the correct order), and, perhaps most importantly, c) there needs to be a mechanism to bypass or cope with effects of any caches (since cached data may not reflect the actual device register state): the MMU offers a natural solution via virtual memory. With port-mapped I/O, especially when the address space is segregated and there is a dedicated I/O bus, this may not be as problematic. For example, using special-purpose instructions explicitly hints (to the compiler) at `volatile`-esque content.
  - Use of a single bus is simpler, both in terms of the resources needed plus the ease by which *all* attached devices get a consistent view of access activity. This is harder when multiple buses are used, because devices attached to one bus somehow need to see activity on the other. Imagine for example that memory-mapped I/O is used, but there is one I/O bus and a separate memory bus. When the processor accesses an address in memory, the memory bus is used. But if an address mapped to a device register is used, the device has to see this access if it is to act correctly.

  Even then, the terminology used is a little inexact because various hybrid options are possible. For example, x86 opts for port-mapped I/O (by using `inb` and `outb` instructions, plus variants, to either receive and transmit bytes to a given port). However, although there is a separate port address space, this is for control-oriented device registers: data-oriented device registers (e.g., buffers) are memory-mapped instead.
- As a result of the trade-offs involved, one can easily identify examples of any combination of design choices. For instance, the PDP-8 computer used a port-mapped approach while the later PDP-11 was instrumental in introducing memory-mapped I/O.

Concepts (3)

- Idea: device registers are treated as sort of pseudo-memories



yielding a design space wrt. (at least)

1. single or multiple address spaces, and
2. single or multiple buses

and hence either **memory-mapped I/O** or **port-mapped I/O**.

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # f12b939 @ 2016-02-08 University of BRISTOL

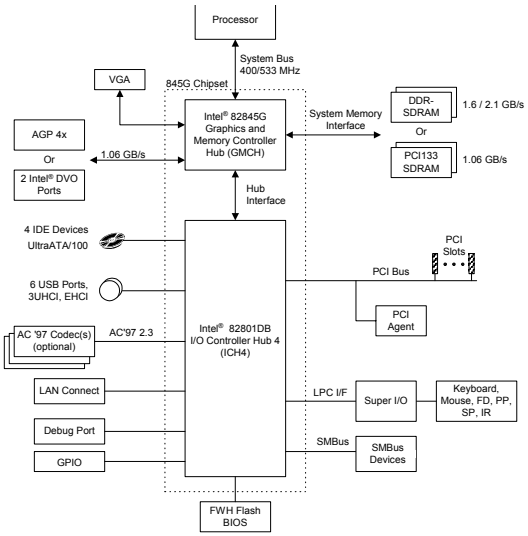
Notes:

- You could argue the device, as illustrated, is missing a component that acts as the **bus interface**. In reality this *is* likely to be a separate component, but we ignore that here and assume the connection to the bus always includes this.
- An obvious feature of this architecture is that the device controller can offer a high-level, fairly generic interface to the underlying mechanism; in contrast, the interface between device controller and mechanism is at least lower-level, and more specific. For example, a disk controller can offer a uniform interface to differing disk mechanisms (e.g., which have different geometries, or capacities), and hide complications such as the use (or not) of error detection and/or correction.
- The choice between memory-mapped and port mapped I/O must be based on a range of trade-offs:
  - Use of port-mapped I/O typically demands special-purpose instructions (to move data between the port and general-purpose registers), whereas memory-mapped I/O can use standard memory access instructions.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms (e.g., the MMU will naturally protect access to a mapped address by an unprivileged instruction), whereas port-mapped I/O may not: especially where multiple buses are used, there might be a need for a dedicated I/O MMU.
  - In using memory-mapped I/O, it is obviously important that mapped addresses are protected from use as "normal" memory even though they appear, by design, in the accessible address space. This is easier to solve than it seems: the MMU offers a natural solution via virtual memory.
  - Use of a single bus implies that all devices connected to it must operate at or above the bus frequency: if they didn't (i.e., their clock frequency was *lower* than the bus frequency) they would be unable to access it.
  - Unless a single bus is wide or has a high frequency, it will often provide less communication bandwidth than multiple buses. That is, it provides a less performant solution: less communication can potentially happen per unit of time, in part because more devices share access.
  - Access to memory typically assumes there are no side-effects (bar an update to the content). With memory-mapped I/O, however, an address is mapped to a device register whose content may be updated by the device independently from any loads and stores. This has (at least) two implications, namely a) the mapped address might be marked using the `volatile` keyword (in C) to signal this fact, b) so-called barrier instructions might be required to ensure any pending load or store instructions are completed (in the correct order), and, perhaps most importantly, c) there needs to be a mechanism to bypass or cope with effects of any caches (since cached data may not reflect the actual device register state): the MMU offers a natural solution via virtual memory. With port-mapped I/O, especially when the address space is segregated and there is a dedicated I/O bus, this may not be as problematic. For example, using special-purpose instructions explicitly hints (to the compiler) at `volatile`-esque content.
  - Use of a single bus is simpler, both in terms of the resources needed plus the ease by which *all* attached devices get a consistent view of access activity. This is harder when multiple buses are used, because devices attached to one bus somehow need to see activity on the other. Imagine for example that memory-mapped I/O is used, but there is one I/O bus and a separate memory bus. When the processor accesses an address in memory, the memory bus is used. But if an address mapped to a device register is used, the device has to see this access if it is to act correctly.

  Even then, the terminology used is a little inexact because various hybrid options are possible. For example, x86 opts for port-mapped I/O (by using `inb` and `outb` instructions, plus variants, to either receive and transmit bytes to a given port). However, although there is a separate port address space, this is for control-oriented device registers: data-oriented device registers (e.g., buffers) are memory-mapped instead.
- As a result of the trade-offs involved, one can easily identify examples of any combination of design choices. For instance, the PDP-8 computer used a port-mapped approach while the later PDP-11 was instrumental in introducing memory-mapped I/O.

An Aside: real bus (and chip set) architectures

Example: Intel 845 "Brookdale" (circa 2002)



http://download.intel.com/design/chipsets/datashts/29074602.pdf

Notes:

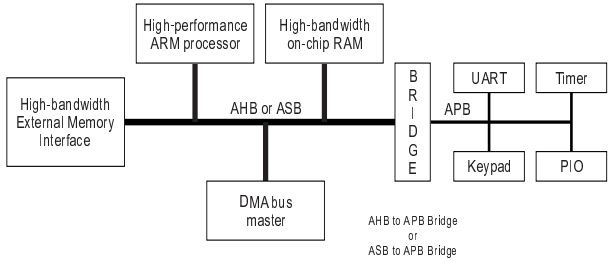
- To quantify "fast" versus slow, you can look at the specification at

  https://en.wikipedia.org/wiki/List_of_Intel_chipsets

  Note the FSB frequency is 400MT s^{-1} (where mega *transfers* per second counts the bus cycles possible per second) vs. the PCI frequency of 33MHz. So-called "conventional" PCI, which this is, had a 32-bit width.
- Normally the diagram is drawn top-to-bottom, so the north-bridge is above the south-bridge! Given there is a FSB, one common question (or joke: it's hard to tell since it's not very funny) is "where/what is the Back-Side Bus?" There *was* such a thing, although somewhat niche: this was an internal bus used to connect the processor to L2 cache etc.
- AGP-type devices (typically GPUs etc.) can be attached to the north-bridge, or Memory Controller Hub (MCH). The south-bridge, or I/O Controller Hub (ICH) is also how *other* "slow" buses are attached: examples include USB, plus legacy devices (e.g., serial and parallel ports, non-USB mice and keyboards) via a so-called Low Pin Count (LPC) bus.

An Aside: real bus (and chip set) architectures

Example: ARM Advanced Micro-controller Bus Architecture (AMBA) 2.0

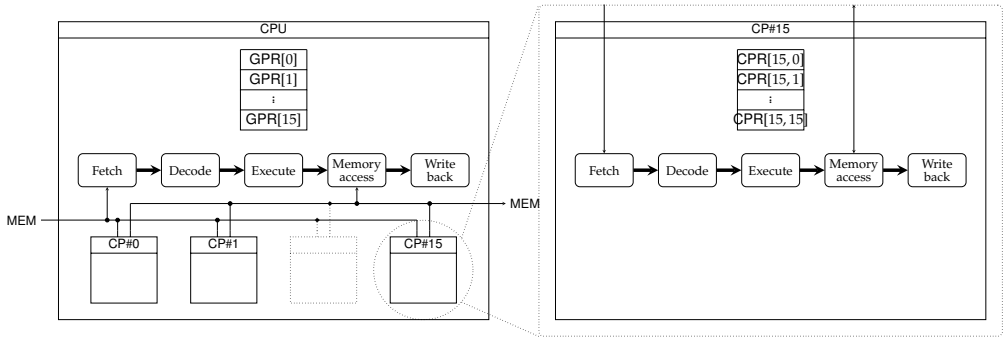


http://infocenter.arm.com/help/topic/com.arm.doc.ihi0011a/index.html

Notes:

- The AMBA specification [**?**] offers a more comprehensive overview.
- The Advanced High-performance Bus (AHB) acts as a replacement for ASB, with performance relationship satisfying AHB > ASB > APB; the Advanced Trace Bus (ATB) was also added as part of the ARM debugging infrastructure.

- The ARMv7-A **co-processor interface** [3, Section A2.9]



allows extension of ISA, where

- there are upto 16 on-chip co-processors (cf. devices) attached,
- each has (upto) 16 registers, which are addressable (cf. ports) by the processor,
- each has an load-store style interface with memory,
- when an instruction for a given co-processor is fetched, *it* executes it rather than the processor.

Notes:

- The notation doesn't matter a lot, but, just to be clear, we've used CPR[i, j] to denote the j-th register of co-processor i.
- When a co-processor instruction is fetched but there is no co-processor attached, this causes an undefined instruction exception: as well as simply making sense, this allows the co-processor function to be emulated in software to allow compatibility.
- The fact that the co-processor interface doesn't *assume* there is a particular co-processor attached means a lot of flexibility in terms of actually doing so: some processor models will have some co-processors, and others lack them.
  - some processors use co-processors #10 and #11 to add (single- and double-precision) floating-point operations to the ISA (i.e., they represent an optional floating-point unit) via the Vector Floating Point (VFP) design,
  - co-processor #14 is sometimes used to support addition of debugging interfaces (plus functionality such as breakpoints, as used by `gdb`),
  - co-processor #15 is normally used as a way to support general system control and configuration,
  - beyond this, non-reserved co-processor numbers can be used to implement any application-specific additions you want: one example is the MOVE [1] co-processor for video encoding.

  Per [3, Chapter 3], the Cortex-A8 uses co-processor #15 as described above: for example, this allows control and configuration of the caches and MMU.

- (Made up) example: consider a floating-point co-processor, where we might have

1. load/store co-processor register from/to memory, e.g.,

$$\begin{array}{lcl} \texttt{ldc p0, cr0, [r1]} & \mapsto & \mathsf{CPR}[0,0] \leftarrow \mathsf{MEM}[\mathsf{GPR}[1]] \\ \texttt{stc p0, cr0, [r1]} & \mapsto & \mathsf{MEM}[\mathsf{GPR}[1]] \leftarrow \mathsf{CPR}[0,0] \end{array}$$

2. move co-processor register from/to register, e.g.,

$$\begin{array}{lcl} \texttt{mrc p0, \#0, r1, cr0, cr1, \#0} & \mapsto & \mathsf{GPR}[1] \leftarrow \mathsf{CPR}[0,0] \\ \texttt{mcr p0, \#0, r1, cr0, cr1, \#0} & \mapsto & \mathsf{CPR}[0,0] \leftarrow \mathsf{GPR}[1] \end{array}$$

3. execute a co-processor operation, e.g.,

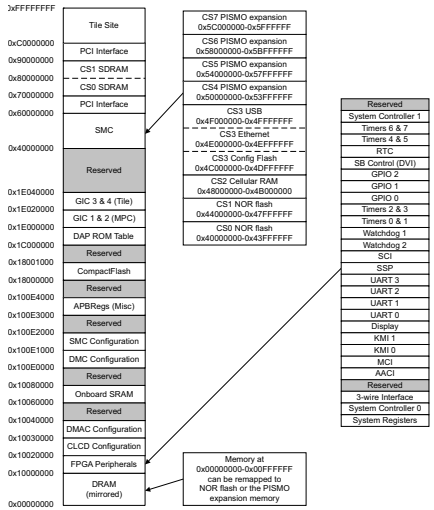
$$\texttt{dcp p0, cr0, cr1, cr2, \#0} \quad \mapsto \quad \mathsf{CPR}[0,0] \leftarrow \mathsf{CPR}[0,1] + \mathsf{CPR}[0,2]$$

Notes:

- The point here is that since there is one interface to any co-processor, the format of instructions is necessarily general-purpose. This makes the example odd in some respects: note that
  - each instruction type identifies the co-processor number using the first operand, e.g., `p0` is co-processor 0,
  - for `ldc` and `stc`, the address is supplied by the processor using standard addressing modes (based on use of the standard general-purpose register file), and
  - `mrc`, `mcr` and `dcp` instructions include immediate an (or two, in the former cases) operand for use as an opcode: this is obvious for `dcp`, but also allows extra operations to occur in support of a transfer (e.g., allowing an addressing mode).

Implementation (3) – memory-mapped I/O on Cortex-A8 Platform Baseboard

Example: Cortex-A8 Platform Baseboard memory map [2, Figure 4-1]

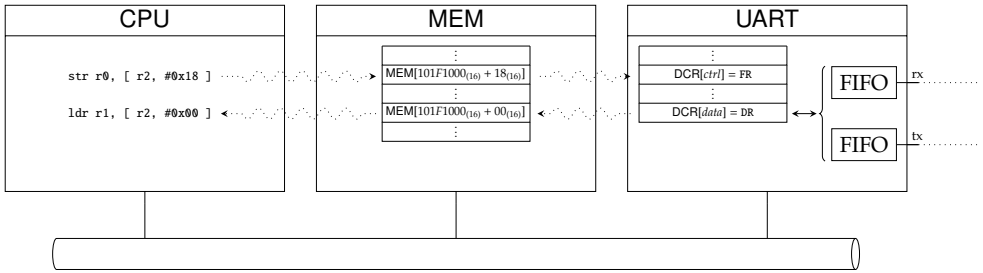


http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/index.html

Notes:

Implementation (4) – memory-mapped I/O on Cortex-A8 Platform Baseboard

- Translation: we have, for example,



st. in C, we'd

1. define a pointer to the base address, i.e.,

```
volatile uint32_t* const UART0 = ( uint32_t* )( 0x10009000 );
```

   then
2. access device registers via offsets from this

```
*( UART0 + 0x18 ) = x;
```


Notes:

- The term **base address** is used, within the context of memory-mapped I/O, to mean the address where the device register mapping starts; each device register is at an offset from this base address. This approach means if/when the mapping changes, instructions performing memory-mapped I/O *only* need alter the base address they use (rather than the offsets, which remain fixed).
- There are various way to improve the quality of this source code:
  1. One could define a specific macros for each device register, e.g.,

```
volatile uint32_t* const UART0_DR = ( uint32_t* )( 0x10009000 );
```

     and thereby remove the need for any magic constant style offsets: we just do

```
*UART0_DR = x;
```

     instead.
  2. A neater way still would be to define a structure that fits over the mapped region, allowing access to device registers via the structure field identifiers. If we had such a structure, `uart_t` say, then we could first define

```
volatile uart_t* const UART0 = ( uart_t* )( 0x10009000 );
```

     and then perform access via

```
UART0->DR = x;
```

I/O Programming Models (1)

- Problem(s):
  1. there is a limited amount of I/O bandwidth available, so using it effectively is important, and
  2. ideally, we'd like the processor to be able to
     - avoid having to check if I/O *can* occur and
     - avoid having to wait for I/O *to* occur.
- Solution(s): efficient approaches st. we know
  - *when* to communicate (polling vs. interrupts), *and*
  - *how* to communicate (DMA vs. programmed I/O).

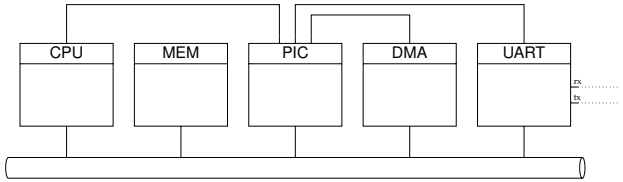
Notes:

- The idea of DMA is conceptually simple: the DMA engine is really just a special-purpose processor that can act as a bus master (i.e., perform a write or read cycle in the same way that the processor itself can). As such, the processor can configure the DMA engine to then autonomously transfer data (between bus slaves) rather then do it itself.
- Although the term DMA engine is common, you may also see DMA controller used to refer to the same thing (i.e., the device that does DMA). The name suggests that transfers are to and from memory, but typically more general x-to-x transfers for $x \in \{\text{memory}, \text{device}\}$ are possible: this means, for example, that DMA can be used to accelerate `memcpy` (or at least a version of it) as easily as transferring the contents of memory to or from disk.
- With more than one bus master (*and* more than one DMA channel), it is clear the total bus bandwidth will need to be shared: shared access implies the need for a **bus arbiter**, but can be realised in various ways. Two examples are for the DMA engine to operate in
  - **burst mode**, where it performs the transfer in one burst requesting then releasing access only after it completes (st. other access during the transfer is prevented), or
  - **cycle stealing mode** where it performs the transfer in many bursts, requesting then releasing access before and after each one (st. other access can be interleaved): as the name suggests, the idea is that the DMA engine "steals" bus cycles that would otherwise be unused by other bus masters.

  This issue is important: if there is no dedicated interface between the processor and memory, use of burst mode transfer may block instruction fetches by said processor and therefore effectively block execution for the transfer duration.

I/O Programming Models (2)

- Example: consider



and a goal of transmitting some data in memory via the UART, i.e.,

Listing (C)

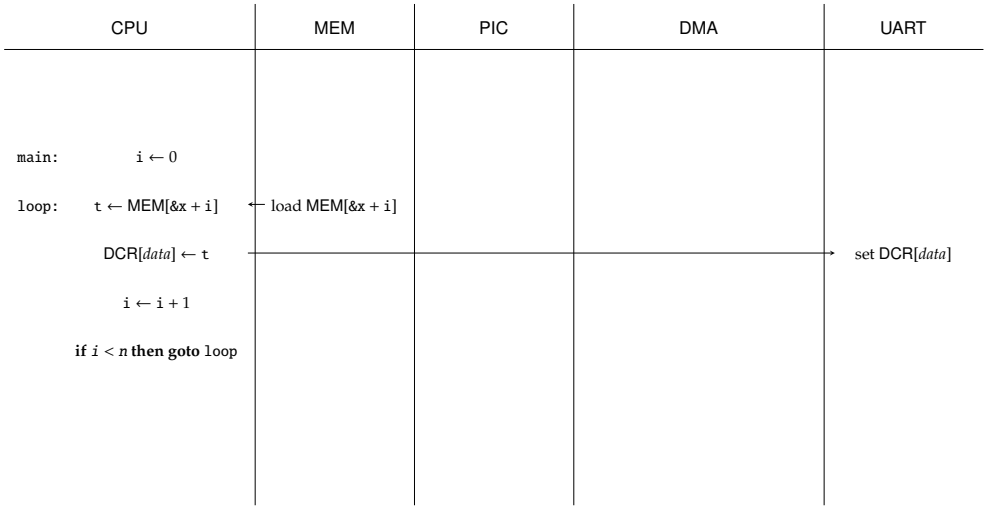
```
1 for( int i = 0; i < n; i++ ) {
2   while( *( UART0 + 0x18 ) & 0x20 ) {
3     /* wait while transmit FIFO is full */
4   }
5
6   *( UART0 + 0x00 ) = x[ i ];
7 }
```


Notes:

- Where in the system the DMA engine is placed is implementation dependent; it could be a dedicated component as shown (cf. AMBA or PCI), or could be embedded in another component (cf. I/OAT, where the processor itself has an embedded DMA engine). However, the location can have implications for how other elements of the system should or must operate. As an example, imagine the processor has an on-chip L1 cache: noting that accesses made by the DMA engine will not go through the cache to memory,
  - if the DMA engine loads directly from address x, it needs to snoop the cache (or the cache should signal to it) st. it does not mistakenly ignore a cached, dirty version of the same address, and
  - if the DMA engine stores directly to address x, it needs to signal to the cache st. a cached version of the same address is invalidated.

I/O Programming Models (3)

- ... where we can use (at least) three I/O strategies:
  1. CPU-driven (or programmed) I/O,

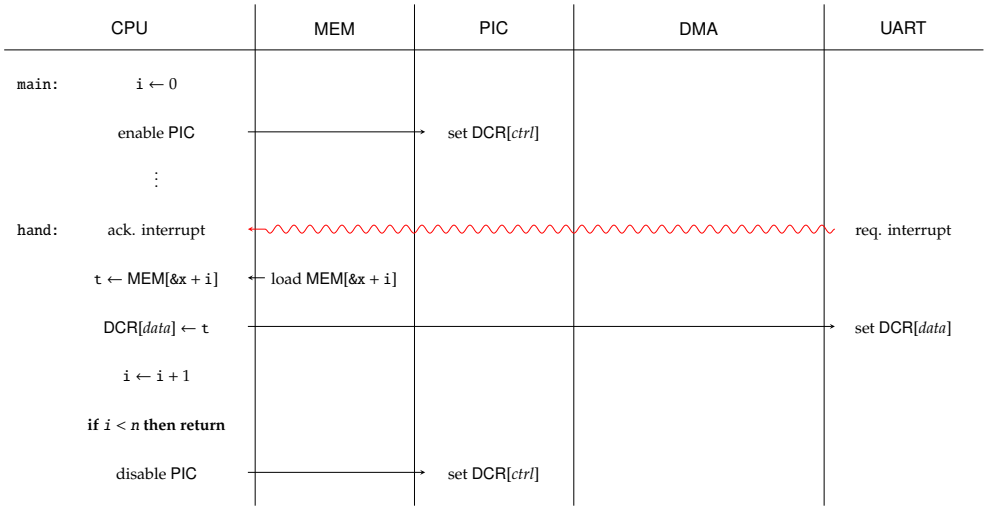


Notes:

- You may see the acronym Programmed I/O (PIO) used in some places.
- – With CPU-driven I/O, there is no extra overhead in the sense the processor does not need to communicate with or depend on any other devices (memory and UART aside of course). However, the processor is clearly busy when executing the transfer: it cannot do anything else during this period. This problem is exacerbated by the need for **polling**, e.g., checking whether or not the device is ready to accept the next byte. This is omitted from the illustration, but clearly might represent a high overhead depending on the relative speeds of processor and device.
  – With interrupt-driven I/O the processor might be able to do something else in the period between interrupts from the UART signalling it is free to accept the next byte of data; on the other hand, it demands more careful control by the processor (wrt. synchronisation between the main flow of control and the interrupt handler, which is invoked asynchronously, and of the interrupt controller). The ability for the processor to do something else is limited by how often the UART interrupts it: if this is very often, the extra overhead probably makes the solution slower than use of CPU-driven I/O, but if this is very seldom them the advantage is more obvious.
  – With DMA-driven I/O, the processor essentially offloads *everything* bar some initial configuration to the DMA engine. In a sense, the processor is using it like a special-purpose transfer co-processor. This is efficient, particularly for large transfers since the overhead of configuration is amortised, since the processor can do something else in parallel; it also reduces the number of interrupts from one per byte in the transfer (i.e., n), to one per transfer. Beyond the fact it is more complex, the only clear disadvantage is the need for bespoke DMA engine hardware.

I/O Programming Models (3)

- ... where we can use (at least) three I/O strategies:
  1. CPU-driven (or programmed) I/O,
  2. interrupt-driven I/O, and

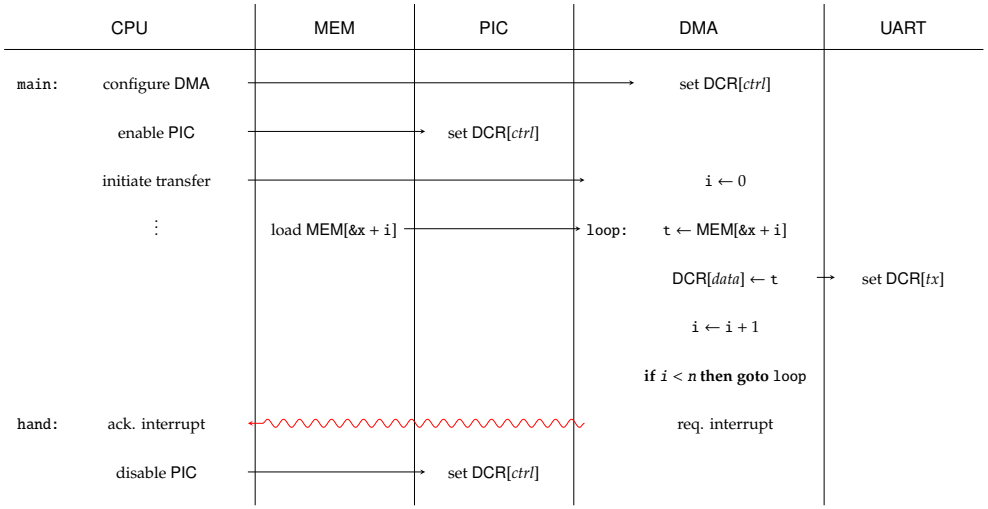


Notes:

- You may see the acronym Programmed I/O (PIO) used in some places.
- – With CPU-driven I/O, there is no extra overhead in the sense the processor does not need to communicate with or depend on any other devices (memory and UART aside of course). However, the processor is clearly busy when executing the transfer: it cannot do anything else during this period. This problem is exacerbated by the need for **polling**, e.g., checking whether or not the device is ready to accept the next byte. This is omitted from the illustration, but clearly might represent a high overhead depending on the relative speeds of processor and device.
  – With interrupt-driven I/O the processor might be able to do something else in the period between interrupts from the UART signalling it is free to accept the next byte of data; on the other hand, it demands more careful control by the processor (wrt. synchronisation between the main flow of control and the interrupt handler, which is invoked asynchronously, and of the interrupt controller). The ability for the processor to do something else is limited by how often the UART interrupts it: if this is very often, the extra overhead probably makes the solution slower than use of CPU-driven I/O, but if this is very seldom them the advantage is more obvious.
  – With DMA-driven I/O, the processor essentially offloads *everything* bar some initial configuration to the DMA engine. In a sense, the processor is using it like a special-purpose transfer co-processor. This is efficient, particularly for large transfers since the overhead of configuration is amortised, since the processor can do something else in parallel; it also reduces the number of interrupts from one per byte in the transfer (i.e., n), to one per transfer. Beyond the fact it is more complex, the only clear disadvantage is the need for bespoke DMA engine hardware.

I/O Programming Models (3)

- ... where we can use (at least) three I/O strategies:
  1. CPU-driven (or programmed) I/O,
  2. interrupt-driven I/O, and
  3. DMA-driven I/O.



Notes:

- You may see the acronym Programmed I/O (PIO) used in some places.
- – With CPU-driven I/O, there is no extra overhead in the sense the processor does not need to communicate with or depend on any other devices (memory and UART aside of course). However, the processor is clearly busy when executing the transfer: it cannot do anything else during this period. This problem is exacerbated by the need for **polling**, e.g., checking whether or not the device is ready to accept the next byte. This is omitted from the illustration, but clearly might represent a high overhead depending on the relative speeds of processor and device.
  – With interrupt-driven I/O the processor might be able to do something else in the period between interrupts from the UART signalling it is free to accept the next byte of data; on the other hand, it demands more careful control by the processor (wrt. synchronisation between the main flow of control and the interrupt handler, which is invoked asynchronously, and of the interrupt controller). The ability for the processor to do something else is limited by how often the UART interrupts it: if this is very often, the extra overhead probably makes the solution slower than use of CPU-driven I/O, but if this is very seldom them the advantage is more obvious.
  – With DMA-driven I/O, the processor essentially offloads *everything* bar some initial configuration to the DMA engine. In a sense, the processor is using it like a special-purpose transfer co-processor. This is efficient, particularly for large transfers since the overhead of configuration is amortised, since the processor can do something else in parallel; it also reduces the number of interrupts from one per byte in the transfer (i.e., n), to one per transfer. Beyond the fact it is more complex, the only clear disadvantage is the need for bespoke DMA engine hardware.

Conclusions

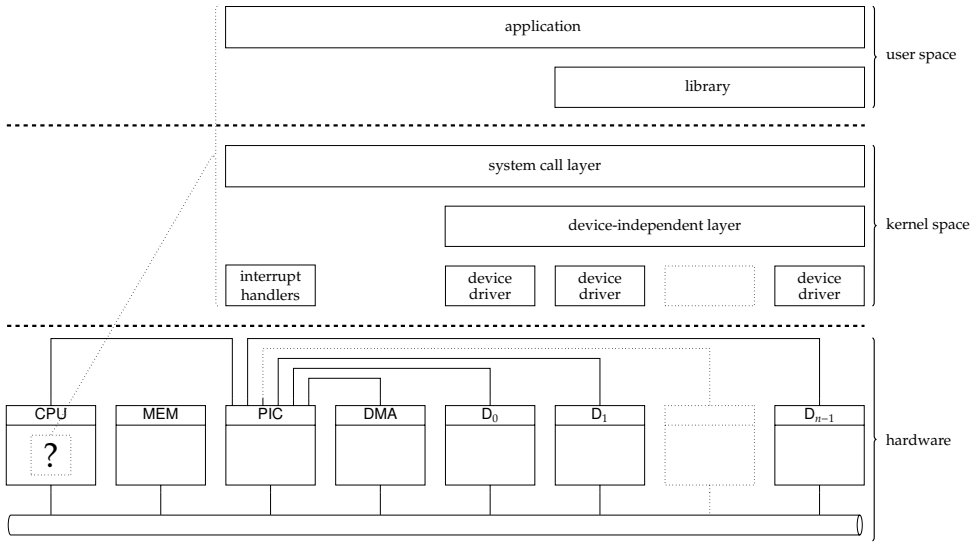
- Take away points:
  1. I/O is hard!
  2. The I/O sub-system must support
     - privilege management via invocation of **mode switches**,
     - the **system call interface**, allowing the kernel and user space software to interact, *and*
     - a suite of device-specific **device drivers**, allowing the kernel and hardware to interact,

     plus deal with some (significant) engineering challenges, e.g.,
     - unreliable and unpredictable devices and communication,
     - large, diverse and (relatively) fast-changing space of device types,
     - (non-)uniformity of kernel and hardware interfaces *and*
     - requirement for efficiency (cf. **programming models**)

     suggesting an organisation something like ...

Notes:

Conclusions



Notes:

- The software engineering aspect of the kernel space I/O sub-system is an involved topic in itself, with various trade-offs implying a range of reasonable strategies and organisations. Comprehensive overviews by [11, Section 5.3] and [9, Section 13.4] do this justice, but a few of the issues are outlined below:
  - There needs to be some way of naming, or identifying each of the attached devices. An example is the Linux approach of a major/minor numbering, whereby the major number specifies the device driver (implying the type or family), and the minor number relates to the attached device itself.
  - The device-independent layer performs various general-purpose roles extracted from the special-purpose, device-dependant device drivers. A significant example is buffering, for which there are a surprisingly large number of strategies.
  - Ideally, the interfaces between layers will be uniform in the sense that lower layers are perfectly abstracted. This turns out to be quite a challenge at the device driver layer. We'd prefer to have all devices present the same interface, since this makes the device independent layer simpler and supports a more modular approach to the drivers themselves; this is hard to realise, however, because there are so many devices which may be similar but, equally, different in subtle ways.
    In reality, one device type might imply use of a driver that is, in reality, a driver *stack*. USB is a good example: USB devices are similar in the sense they all communicate via the USB protocol, but the USB driver stack is organised into sub-layers that deal with device-dependent and -independent aspects of a given device.
- As a result of this being *an* organisation rather than *the* organisation of the I/O sub-system, some of the relationships between portions of it are inexact.
  For example, there is an implication that access to the device driver, from user space, must go through other layers; in reality, there is often a way to bypass these to pass configuration information. Linux has the `ioctl` [**?**] system call, and Windows has `DeviceIoControl` which is basically the same.

References

[1] ARM Limited.
MOVE co-processor Technical Reference Manual.
Technical Report DDI-0235, 2004.
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0235c/index.html.

[2] ARM Limited.
RealView Platform Baseboard for Cortex-A8.
Technical Report HBI-0178, 2011.
http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/index.html.

[3] ARM Limited.
ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition.
Technical Report DDI-0406C, 2014.
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html.

[4] J. Corbet, G. Kroah-Hartman, and A. Rubini.
Chapter 14: The Linux device model.
In *Linux Device Drivers* [7].
http://www.makelinux.net/ldd3/.

[5] J. Corbet, G. Kroah-Hartman, and A. Rubini.
Chapter 15: Memory mapping and DMA.
In *Linux Device Drivers* [7].
http://www.makelinux.net/ldd3/.

Notes:

[6] J. Corbet, G. Kroah-Hartman, and A. Rubini.
Chapter 9: Communicting with hardware.
In *Linux Device Drivers* [7].
http://www.makelinux.net/ldd3/.

[7] J. Corbet, G. Kroah-Hartman, and A. Rubini.
Linux Device Drivers.
O'Reilly, 3rd edition, 2005.
http://www.makelinux.net/ldd3/.

[8] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 13: I/O systems.
In *Operating System Concepts* [9].

[9] A. Silberschatz, P.B. Galvin, and G. Gagne.
Operating System Concepts.
Wiley, 9th edition, 2014.

[10] A.S. Tanenbaum and H. Bos.
Chapter 5: Input/output.
In *Modern Operating Systems* [11].

[11] A.S. Tanenbaum and H. Bos.
Modern Operating Systems.
Pearson, 4th edition, 2015.

Notes: