Concurrent Computing (Operating Systems)

Daniel Page

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

February 9, 2018

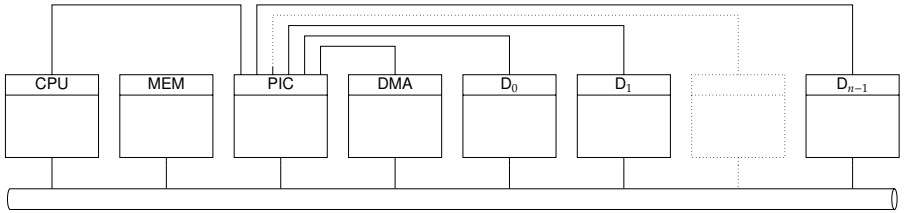
Keep in mind there are *two* PDFs available (of which this is the latter):

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:

- Problem: our example computer system looks like this



but (so far) we only really know about one component, i.e., the processor.

- Goals: explain
  1. what some of the *other* components are, plus
  2. how the processor communicates with and hence uses them

  and thus how **Input/Output (I/O)** functionality is realised.

Notes:

- In some more detail, the goals are to answer various questions such as
  - what "other components" are there?
  - how do components asynchronously signal their need for attention to the processor?
  - how is data transferred (a)synchronously between a component and the processor?
  - what API(s) should a programmer be offered so they can make use of the above?

  Providing answers depends on our addressing two features illustrated in the diagram, namely
  1. the top-most connections model an **interrupt** delivery mechanism, and
  2. the bottom-most connections model a communication **bus**.
- As an aside, or to add some detail, note that we already know the processor is, internally, a concurrent system: the fact it uses a pipeline satisfies this criteria. So now adding n other hardware devices, we have a more obvious, externally concurrent system (and the challenges that go with it).

Concept (1)

Definition (Walker and Cragon [9])

An **interrupt** is an event (or condition) where normal execution of instructions is halted: two major classes exist, namely

- **hardware interrupt** are (typically) generated asynchronously, by an external source, and intentionally (e.g., by a hardware device), while
- **software interrupt** are (typically) generated synchronously, by an internal source, and either intentionally (e.g., system call, cf. **trap**), *or* unintentionally (e.g., divide-by-zero, cf. **exception**).

Definition (Walker and Cragon [9])

Each **Interrupt ReQuest (IRQ)** causes a software **interrupt handler** to be invoked, whose task is to respond. Note that

- **interrupt latency** measures the time between an interrupt being requested and handled, and
- an **interrupt vector table** (located at a known address) allows a specific handler to be invoked for each interrupt type.

Notes:

- It's *vital* to understand that the terminology around interrupts is totally inconsistent! In the general context of ARM, and ARMv7-A specifically, the term exception [11, Section A2.12] is a catch-all for *all* interrupt types.
  There is no perfect solution to this inconsistency, but in an effort to be generally applicable (and avoid issues such as the implication that exception means "there was an error"), we will stick with more traditional use of the term interrupt. As such, we can taxonomise the various cases wrt.
  - *when* the interrupt is requested (e.g., synchronous or asynchronous),
  - *what* requests the interrupt (e.g., internal or external to the processor),
  - *why* it does so (e.g., intentionally vs. unintentionally).
- In some cases, the term interrupt request is specifically associated with hardware (and hence physical signals); in the case of ARMv7-A, it is also unfortunate in the sense it overloads the same acronym as used for IRQ mode etc. Either way, keep in mind that an alternative, namely to *request* an interrupt, basically means the same thing.
- It is tempting to say an interrupt is *triggered*, and this is a term used sometimes. We avoid this here, however, since it could be read as implying the interrupt is handled instantly (which is not necessarily the case).

Concept (2)

- Conceptually, the process of handling an interrupt is:

1. detect the interrupt,
2. update the processor mode,
3. preserve the processor state,
4. execute interrupt handler,
5. restore the processor state,
6. update the processor mode,
7. restart (an) instruction.

Notes:

- As such, interrupts and interrupt handling are somewhat similar, in a *conceptual* sense, to how function calls work: similarities can be drawn between function caller/callee and interrupt source/handler. But although it is reasonable to characterise some of the above as a soft of interrupt handling (resp. function call) prologue and epilogue, it is also true that some important differences exist, e.g.,
  1. interrupts can occur asynchronously (i.e., at *any* time rather than as the result of synchronous instruction execution), and
  2. various special operations are performed as part of the prologue and epilogue (e.g., updating processor mode) which can not (and should not) be realised by the interrupt source.
- A typical example would be that, having detected the interrupt, we update the processor mode from user to kernel mode; once interrupt handling is complete in kernel mode, the processor mode is updated back again so that the restarted instruction executes in user mode.
- There are several viable options for preserving the processor state, which will include, for example, the general-purpose registers: they need to be preserved, because their content relates to the process that was interrupted. Two examples are termed **shadow registers** (essentially another register file that exists specifically for the purpose of handling interrupts), and **shadow stacks** (which is a stack "owned" by the kernel, onto which the processor state is pushed).
- The careful wording wrt. restarting *an* instruction rather than *the* instruction is intentional. There are sort of two cases:
  1. Restart the instruction that was interrupted. Imagine the interrupt is requested by a hardware device demanding attention from the kernel. In this case the executing process is interrupted st. the kernel can handle the interrupt, but once this is complete the process (i.e., the interrupted instruction) should be resumed: it should not even be aware anything happened.
  2. Restart some other instruction. Imagine the interrupt is requested by an exception stemming from the executing process. In this case, we potentially cannot resume the process: if the exception is severe enough that we terminate the process (e.g., a segmentation fault), there is no instruction in it to restart! So in this case, we would need to restart some other instruction (e.g., from another process).
- [11, Sections B1.8 + B1.9] offer a detailed overview of the exception model implemented by ARMv7-A compliant processors: we look at (an example of) this in more detail later.
  Although we ignore them on the whole, keep in mind various differences exist between what is often termed the "classic"exception model of ARMv7-A/R, ARMv6 and previous ISAs vs. the "new" ARMv7-M model. An example is the type of interrupt vector table entries: in the former each entry is a branch instruction (i.e., a branch to the interrupt handler), whereas the latter they are addresses (i.e., pointers to the interrupt handler).

Concept (2)

- Conceptually, the process of handling an interrupt is:

interrupt handling (reality)		function calling (analogy)
1. detect the interrupt,		
2. update the processor mode,		
3. preserve the processor state,		3. preserve caller-save registers,
4. execute interrupt handler,	$\simeq$	4. invoke callee function,
5. restore the processor state,		5. restore caller-save registers,
6. update the processor mode,		
7. restart (an) instruction.		7. resume caller function.

© Daniel Page (<redacted>)
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes:

- As such, interrupts and interrupt handling are somewhat similar, in a *conceptual* sense, to how function calls work: similarities can be drawn between function caller/callee and interrupt source/handler. But although it is reasonable to characterise some of the above as a soft of interrupt handling (resp. function call) prologue and epilogue, it is also true that some important differences exist, e.g.,
  1. interrupts can occur asynchronously (i.e., at *any* time rather than as the result of synchronous instruction execution), and
  2. various special operations are performed as part of the prologue and epilogue (e.g., updating processor mode) which can not (and should not) be realised by the interrupt source.
- A typical example would be that, having detected the interrupt, we update the processor mode from user to kernel mode; once interrupt handling is complete in kernel mode, the processor mode is updated back again so that the restarted instruction executes in user mode.
- There are several viable options for preserving the processor state, which will include, for example, the general-purpose registers: they need to be preserved, because their content relates to the process that was interrupted. Two examples are termed **shadow registers** (essentially another register file that exists specifically for the purpose of handling interrupts), and **shadow stacks** (which is a stack "owned" by the kernel, onto which the processor state is pushed).
- The careful wording wrt. restarting *an* instruction rather than *the* instruction is intentional. There are sort of two cases:
  1. Restart the instruction that was interrupted. Imagine the interrupt is requested by a hardware device demanding attention from the kernel. In this case the executing process is interrupted st. the kernel can handle the interrupt, but once this is complete the process (i.e., the interrupted instruction) should be resumed: it should not even be aware anything happened.
  2. Restart some other instruction. Imagine the interrupt is requested by an exception stemming from the executing process. In this case, we potentially cannot resume the process: if the exception is severe enough that we terminate the process (e.g., a segmentation fault), there is no instruction in it to restart! So in this case, we would need to restart some other instruction (e.g., from another process).
- [11, Sections B1.8 + B1.9] offer a detailed overview of the exception model implemented by ARMv7-A compliant processors: we look at (an example of) this in more detail later.
  Although we ignore them on the whole, keep in mind various differences exist between what is often termed the "classic"exception model of ARMv7-A/R, ARMv6 and previous ISAs vs. the "new" ARMv7-M model. An example is the type of interrupt vector table entries: in the former each entry is a branch instruction (i.e., a branch to the interrupt handler), whereas the latter they are addresses (i.e., pointers to the interrupt handler).

Concept (3)

Definition (Walker and Cragon [9])

External hardware devices are interfaced with the processor via an **interrupt controller**, which

- multiplexes a large(r) number of devices to a small(er) number of interrupt signals (into the processor), and
- offer extended functionality, such as priority levels.

Notes:

Concept (4)

Definition (Walker and Cragon [9])

An interrupt may be

- **maskable** if it can be ignored (or disabled) by setting an **interrupt mask** (e.g., within a control register), or
- **non-maskable** otherwise.

Definition (Walker and Cragon [9])

An interrupt is deemed

- **precise** if it leaves the processor in a well-defined state, or
- **imprecise** otherwise.

Per [9, 8], well-defined is taken to mean

1. the program counter is retained somehow,
2. all instructions before current one have completed,
3. no instructions after current one have completed, and
4. the execution state of current instruction is known.

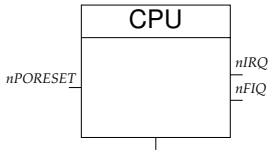
Notes:

- The following represents a non-exhaustive list of other interrupt-related terms that you may encounter: the taxonomy in [9] offers a good overview.
  - An non-maskable interrupt is often acronym'ised as **Non-Maskable Interrupt (NMI)**.
  - Interrupt handling may be
    - **nested** if instant handling of one interrupt while handling another is possible (implying the handler must be **reentrant**), or
    - **non-nested** otherwise,

    and
    - **queued** if deferred handling of one interrupt while handling another is possible (st. interrupts are handled to completion, one at a time), or
    - **non-queued** otherwise.
  - An interrupt may be termed **spurious** if, at the point when it is handled, there is no longer a need to handle it; this suggests the interrupt latency was long enough that the interrupt signal changed (e.g., the hardware device waited, and timed-out so no longer needs the attention it originally demanded).
- Bar the high-level definition, we'll ignore the problems associated with imprecise interrupts and assume they are precise. The main reasons stem from a need to investigate advanced topics in computer architecture (e.g., superscalar execution) first, *if* more complete coverage were then attempted.

Implementation: Cortex-A8 (1) – step #1 detect the interrupt

- Interrupt detection is managed automatically by the processor



st.

- an interrupt can be requested, e.g., by
  - a software system call,
  - a software exception, or
  - a hardware signal,
- CPSR[F] and CPSR[I] mask FIQ- and IRQ-based interrupts respectively.

Notes:

- [12, Appendix A] lists the Cortex-A8 signals, including *nIRQ* and *nFIQ* which are obviously relevant here.
- Originally, the `swi` (or "software interrupt") instruction was used to request a software-based interrupt; this was later replaced by the `svc` (or "supervisor call") instruction. Keep in mind the two are identical wrt. encoding (i.e., bar the mnemonics), but depending on the context you may see the former rather than latter used: for example `gdb` *may* show the disassembly of such an instruction using an `swi` mnemonic.
- Keeping in mind that `cpsr` can be used with various suffixes per [11, Section B9.3.129], for example, the following fragments act to enable

```
1                   ; enable  FIQ interrupts
2 mrs r0,   cpsr    ; load      CPSR
3 bic r0, r0, #0x40 ; clear bit 6 of CPSR, i.e., F flag
4 msr cpsr_c, r0    ; store     CPSR (control bits only)
5                   ; enable  IRQ interrupts
6 mrs r0,   cpsr    ; load      CPSR
7 bic r0, r0, #0x80 ; clear bit 7 of CPSR, i.e., I flag
8 msr cpsr_c, r0    ; store     CPSR (control bits only)
```

and disable (or mask)

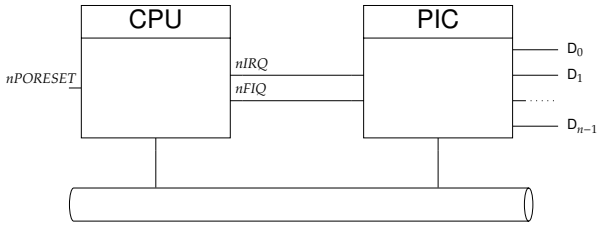
```
1                   ; disable FIQ interrupts
2 mrs r0,   cpsr    ; load      CPSR (control bits only)
3 orr r0, r0, #0x40 ; set   bit 6 of CPSR, i.e., F flag
4 msr cpsr_c, r0    ; store new CPSR
5                   ; disable IRQ interrupts
6 mrs r0,   cpsr    ; load      CPSR (control bits only)
7 orr r0, r0, #0x80 ; set   bit 7 of CPSR, i.e., I flag
8 msr cpsr_c, r0    ; store new CPSR
```

FIQ- and IRQ-based interrupts respectively.

- We'll use the acronym Programmable Interrupt Controller (PIC) as a way of referring to a generic component of this type; various specific decisions and functionalities are evident in specific instances.
  Examples of early PIC include the Intel 8259, and the more modern Advanced Programmable Interrupt Controller (APIC) family; in the context of ARM, it is common to see one or other of their Vectored Interrupt Controller (VIC) and Generic Interrupt Controller (GIC) designs used.

Implementation: Cortex-A8 (1) – step #1 detect the interrupt

- Interrupt detection is managed automatically by the processor



st.

- an interrupt can be requested, e.g., by
  - a software system call,
  - a software exception, or
  - a hardware signal,
- CPSR[F] and CPSR[I] mask FIQ- and IRQ-based interrupts respectively, and
- features are (optionally) added via a *programmable* interrupt controller (e.g., PL190 [14]).

© Daniel Page ⟨dan@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes:

- [12, Appendix A] lists the Cortex-A8 signals, including *nIRQ* and *nFIQ* which are obviously relevant here.
- Originally, the `swi` (or "software interrupt") instruction was used to request a software-based interrupt; this was later replaced by the `svc` (or "supervisor call") instruction. Keep in mind the two are identical wrt. encoding (i.e., bar the mnemonics), but depending on the context you may see the former rather than latter used: for example `gdb` *may* show the disassembly of such an instruction using an `swi` mnemonic.
- Keeping in mind that `cpsr` can be used with various suffixes per [11, Section B9.3.129], for example, the following fragments act to enable

```
1                   ; enable  FIQ interrupts
2 mrs r0,   cpsr    ; load      CPSR
3 bic r0, r0, #0x40 ; clear bit 6 of CPSR, i.e., F flag
4 msr cpsr_c, r0    ; store     CPSR (control bits only)
5                   ; enable  IRQ interrupts
6 mrs r0,   cpsr    ; load      CPSR
7 bic r0, r0, #0x80 ; clear bit 7 of CPSR, i.e., I flag
8 msr cpsr_c, r0    ; store     CPSR (control bits only)
```

and disable (or mask)

```
1                   ; disable FIQ interrupts
2 mrs r0,   cpsr    ; load      CPSR (control bits only)
3 orr r0, r0, #0x40 ; set   bit 6 of CPSR, i.e., F flag
4 msr cpsr_c, r0    ; store new CPSR
5                   ; disable IRQ interrupts
6 mrs r0,   cpsr    ; load      CPSR (control bits only)
7 orr r0, r0, #0x80 ; set   bit 7 of CPSR, i.e., I flag
8 msr cpsr_c, r0    ; store new CPSR
```

FIQ- and IRQ-based interrupts respectively.

- We'll use the acronym Programmable Interrupt Controller (PIC) as a way of referring to a generic component of this type; various specific decisions and functionalities are evident in specific instances.
  Examples of early PIC include the Intel 8259, and the more modern Advanced Programmable Interrupt Controller (APIC) family; in the context of ARM, it is common to see one or other of their Vectored Interrupt Controller (VIC) and Generic Interrupt Controller (GIC) designs used.

Implementation: Cortex-A8 (2) – step #2 update the processor mode

ARMv7-A processor modes [11, Table B1-1]

Name	Mnemonic	CPSR[M]	Privilege level	Security state
User	USR	$10000_{(2)}$	PL0	Either
Fast interrupt (FIQ)	FIQ	$10001_{(2)}$	PL1	Either
Interrupt (IRQ)	IRQ	$10010_{(2)}$	PL1	Either
Supervisor	SVC	$10011_{(2)}$	PL1	Either
Monitor	MON	$10110_{(2)}$	PL1	Secure
Abort	ABT	$10111_{(2)}$	PL1	Either
Hypervisor	HYP	$11010_{(2)}$	PL2	Non-secure
Undefined	UND	$11011_{(2)}$	PL1	Either
System	SYS	$11111_{(2)}$	PL1	Either

Notes:

- Although [11, Section B1.2] carefully outlines the meaning of selected terms wrt. their interpretation for ARMv7-A, we ignore this on the whole. More specifically, notice that *several* privilege levels exist; we ignore these, and assume the processor mode, namely not USR mode or USR mode, implies privilege level, i.e, privileged or not privileged. Put another way, we adopt a simplification by assuming a less granular model with two non-secure privilege levels PL1 and PL0.
- [11, Section B1.3] describes each processor mode in terms of an associated purpose or role. Although several are out-of-scope wrt. the goals here (e.g., HYP and MON modes), one other, namely SYS mode, may see odd: it basically has access to USR mode registers, but operates at a higher privilege level. The goal of supporting *both* supervisor and system modes is that the assists in writing interrupt handlers.

Implementation: Cortex-A8 (3) – step #3 preserve the processor state

	privileged modes							
USR mode	FIQ mode	IRQ mode	SVC mode	MON mode	ABT mode	HYP mode	UND mode	SYS mode
r0	r0	r0	r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7	r7	r7	r7
r8	r8	r8	r8	r8	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12	r12	r12	r12
r13	r13_fiq	r13_irq	r13_svc	r13_mon	r13_abt	r13_hyp	r13_und	r13
r14	r14_fiq	r14_irq	r14_svc	r14_mon	r14_abt	r14_hyp	r14_und	r14
r15	r15	r15	r15	r15	r15	r15	r15	r15
cpsr	cpsr	cpsr	cpsr	cpsr	cpsr	cpsr	cpsr	cpsr
	spsr_fiq	spsr_irq	spsr_svc	spsr_mon	spsr_abt	spsr_hyp	spsr_und	

Notes:

- As the result of providing more banked registers in FIQ mode, there is less need for an interrupt handler to save the user mode equivalents (conversely, there is more chance the interrupt handler can execute within the FIQ mode register set alone). This can potentially reduce interrupt latency, since a non-trivial proportion of it will relate to saving (user) state to memory.

Implementation: Cortex-A8 (4) – step #4 execute an interrupt handler

ARMv7-A interrupt handling [11, Tables B1-3 + B1-4]

Type	Entry mode	Entry low address	Entry high address
Reset	SVC	$00000000_{(16)}$	$FFFF0000_{(16)}$
Undefined instruction	UND	$00000004_{(16)}$	$FFFF0004_{(16)}$
Software interrupt	SVC	$00000008_{(16)}$	$FFFF0008_{(16)}$
(Pre-)fetch abort	ABT	$0000000C_{(16)}$	$FFFF000C_{(16)}$
Data abort	ABT	$00000010_{(16)}$	$FFFF0010_{(16)}$
		$00000014_{(16)}$	$FFFF0014_{(16)}$
IRQ	IRQ	$00000018_{(16)}$	$FFFF0018_{(16)}$
FIQ	FIQ	$0000001C_{(16)}$	$FFFF001C_{(16)}$

ARMv7-A interrupt handling [12, Table 2-12]

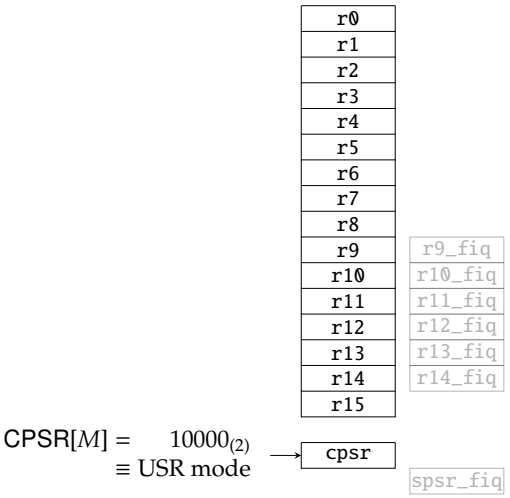
Type	Return offset	Return instruction
Reset	$-0_{(10)}$	
Undefined instruction	$-0_{(10)}$	`movs pc, lr`
Software interrupt	$-0_{(10)}$	`movs pc, lr`
(Pre-)fetch abort	$-4_{(10)}$	`subs pc, lr, #4`
Data abort	$-8_{(10)}$	`subs pc, lr, #8`
IRQ	$-4_{(16)}$	`subs pc, lr, #4`
FIQ	$-4_{(16)}$	`subs pc, lr, #4`

Notes:

- Keep in mind that as a result of our simplification wrt. privilege level, we have *one* interrupt vector table: in reality *several* such tables exist (with difference base addresses) to support various cases we ignored, which are explained in more detail by [11, Section B1.8].
- The system control register, namely SCLR[V] dictates whether the default, low vector interrupt table base address *or* alternative, high based address will be used: where not otherwise stated, we assume the former.
- The interrupt vector table deliberately uses the *last* entry to point at the FIQ handler: this means the handler instructions can be placed at address $0000001C_{(16)}$, eliminating the need for a branch to them and hence reducing the interrupt latency as a result.

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



Notes:

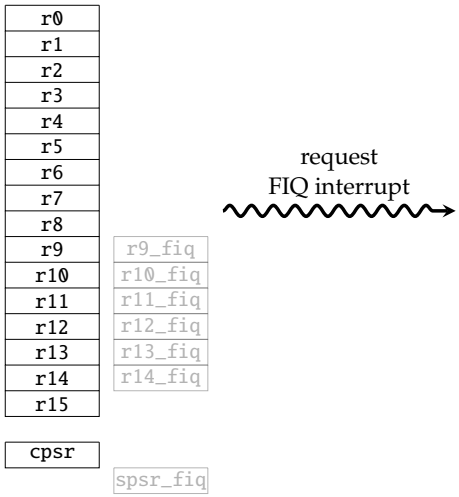
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

  ```
  subs pc, lr, #4
  ```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



Notes:

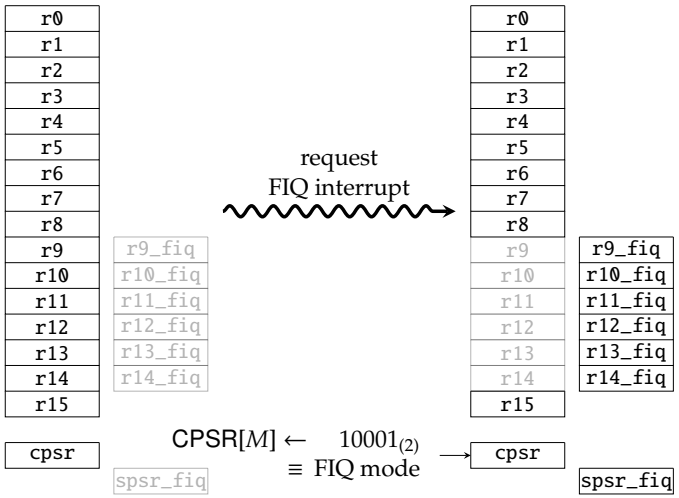
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

```
subs pc, lr, #4
```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



Notes:

- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

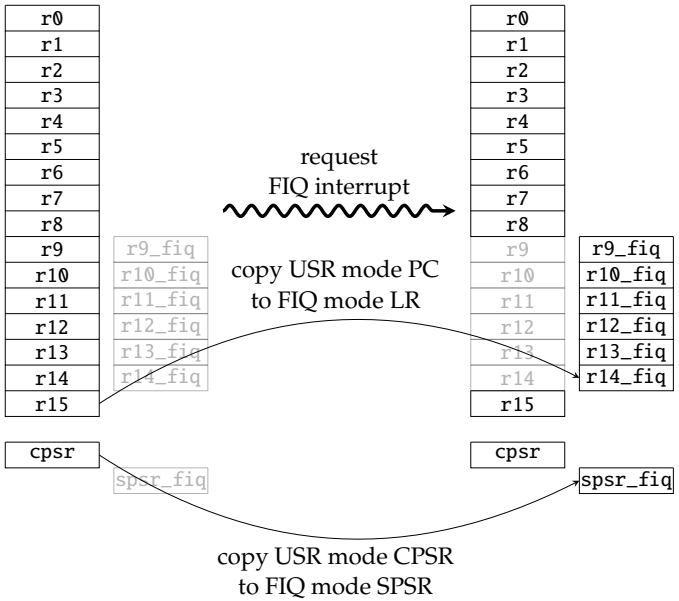
```
subs pc, lr, #4
```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

© Daniel Page ⟨cdp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



Notes:

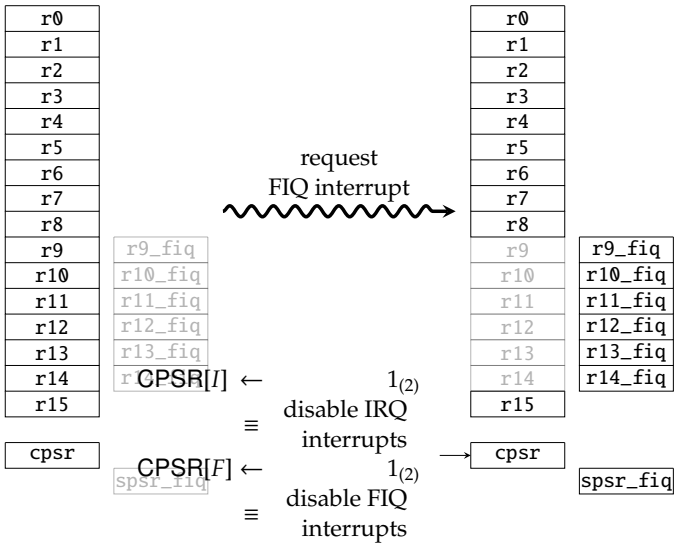
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

```
subs pc, lr, #4
```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



© Daniel Page ⟨dsp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes:

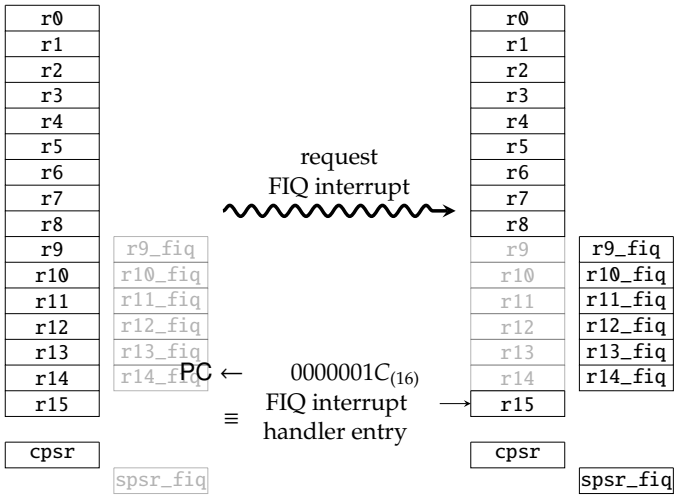
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

```
subs pc, lr, #4
```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



Notes:

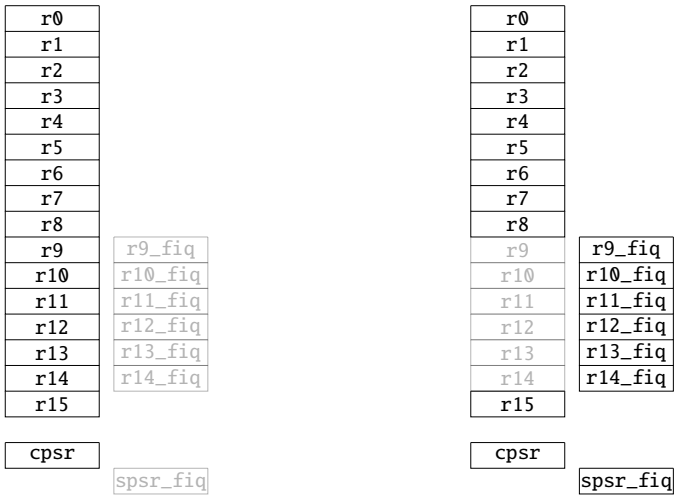
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

```
subs pc, lr, #4
```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



Notes:

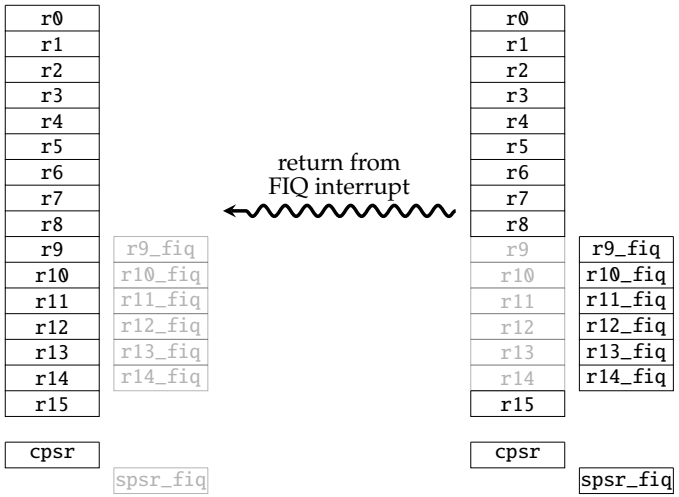
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

```
subs pc, lr, #4
```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



Notes:

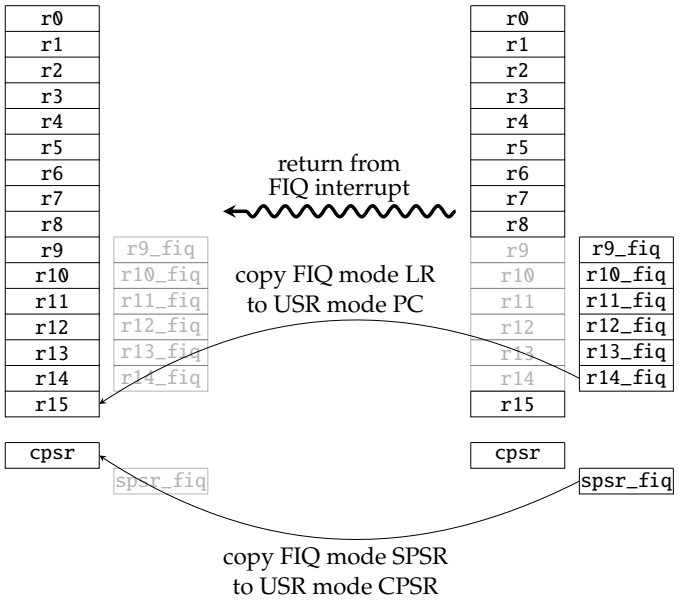
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

```
subs pc, lr, #4
```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



Notes:

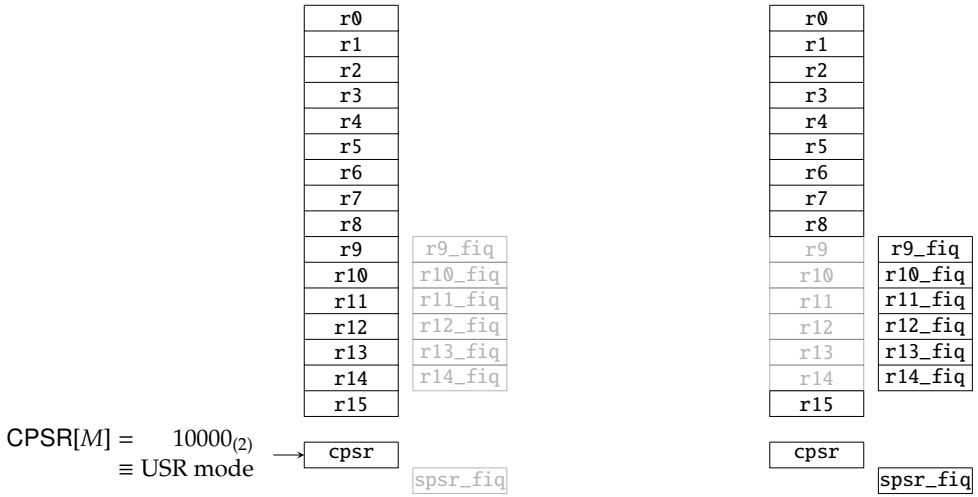
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

```
subs pc, lr, #4
```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



Notes:

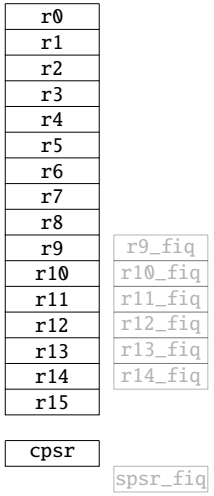
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

```
subs pc, lr, #4
```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

Implementation: Cortex-A8 (5) – putting it all together [11, Section B1.9.12]

- Example:



Notes:

- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed by the processor (e.g., saving the USR mode program counter and CPSR registers), and some will need to be performed by the interrupt handler: the main step is to *atomically* copy the FIQ mode link register and SPSR into the USR mode PC and CPSR registers so the interrupted instruction is restarted in the same state. The description of each interrupt type recommends how to achieve this. [12, Section 2.15.4] says we could execute

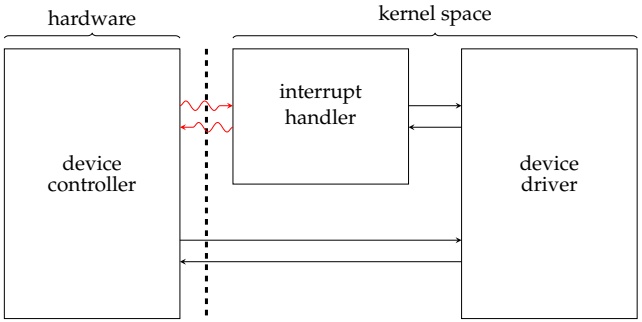
```
subs pc, lr, #4
```

  to copy the FIQ mode PC and CPSR from the (corrected) FIQ LR and SPSR, for example.
- It might not be obvious, but copying the FIQ mode SPSR into USR mode CPSR serves various purposes in one step: it a) set the processor mode, plus b) (re)enables any interrupts that were enabled before. This also shows why it is important to atomically copy the FIQ mode SPSR and LR in one atomic step. That is, once we set the processor more we are no longer in FIQ mode so *can't* then copy the FIQ mode LR even if we wanted to!

Implementation: Cortex-A8 (6) – putting it all together [11, Section B1.9.12]

- ... or, more abstractly,
  - a hardware device

can get attention from (i.e., invoke functionality in) the interrupt-aware kernel, e.g.,



Notes:

- To user space, the system call interface is like an API into the kernel. The manual page accessible via

  ```
  man syscall
  ```

  is for a generic wrapper function for making system calls: it basically does the same as the assembly language version shown here, but offers an abstraction wrt. rules of the API. It *also* acts as a guide to the rules: for ARM (assuming use of EABI), these include

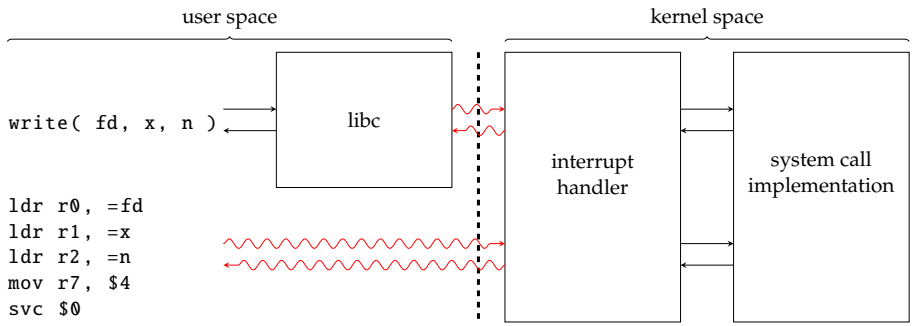
  - the system call identifier goes in `r7`,
  - any arguments to the system call go in `r0` through `r6`,
  - the system call is invoked by executing `svc 0x0`
  - any return value from the system call is in `r0`

  Not that bar the system call identifier (which you could still think of as analogous to the callee address) and that `svc` replaces a branch, this is much like the AAPCS function calling convention.
- Minimising the various overheads associated with making system calls is a challenge common to most kernels. Linux supports an approach based on a **virtual Dynamic Shared Object (vDSO)**. As the name suggests, this is similar to a standard shared object (e.g., library) linked at load- or run-time; the difference is that *this* object is supplied by the kernel: it contains user-space versions of certain performance-critical system call implementations. By placing them in user-space the need for a (full) system call is removed (for example, there is no longer a need to context switch st. the kernel executes), with a light-weight analogue normally executed transparently by the C standard library.
- Two non-blocking behaviours are carefully distinguished here. The early abort case avoids blocking by simply returning without *necessarily* completing the required operation if doing so would block: `read` offers an example, where the number of bytes read (provided by the return value) may not necessarily match that requested (provided as an argument). The asynchronicity case avoids blocking by returning immediately, but organising for the required operation to be completed concurrently; any output may be written into a buffer (provided as an argument), or via a **call-back function** (provided as an argument) which may also be used as notification of completion.

Implementation: Cortex-A8 (6) – putting it all together [11, Section B1.9.12]

- ... or, more abstractly,
  - a hardware device, or
  - a user space instruction

can get attention from (i.e., invoke functionality in) the interrupt-aware kernel, e.g.,



the latter case representing a **system call**, which may be

- blocking,
- non-blocking via early abort, or
- non-blocking via asynchronicity.

© Daniel Page (
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes:

- To user space, the system call interface is like an API into the kernel. The manual page accessible via

  ```
  man syscall
  ```

  is for a generic wrapper function for making system calls: it basically does the same as the assembly language version shown here, but offers an abstraction wrt. rules of the API. It *also* acts as a guide to the rules: for ARM (assuming use of EABI), these include

  - the system call identifier goes in `r7`,
  - any arguments to the system call go in `r0` through `r6`,
  - the system call is invoked by executing `svc 0x0`
  - any return value from the system call is in `r0`

  Not that bar the system call identifier (which you could still think of as analogous to the callee address) and that `svc` replaces a branch, this is much like the AAPCS function calling convention.
- Minimising the various overheads associated with making system calls is a challenge common to most kernels. Linux supports an approach based on a **virtual Dynamic Shared Object (vDSO)**. As the name suggests, this is similar to a standard shared object (e.g., library) linked at load- or run-time; the difference is that *this* object is supplied by the kernel: it contains user-space versions of certain performance-critical system call implementations. By placing them in user-space the need for a (full) system call is removed (for example, there is no longer a need to context switch st. the kernel executes), with a light-weight analogue normally executed transparently by the C standard library.
- Two non-blocking behaviours are carefully distinguished here. The early abort case avoids blocking by simply returning without *necessarily* completing the required operation if doing so would block: `read` offers an example, where the number of bytes read (provided by the return value) may not necessarily match that requested (provided as an argument). The asynchronicity case avoids blocking by returning immediately, but organising for the required operation to be completed concurrently; any output may be written into a buffer (provided as an argument), or via a **call-back function** (provided as an argument) which may also be used as notification of completion.

Concept (1)

Definition

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.

Concept (2)

Definition

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

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

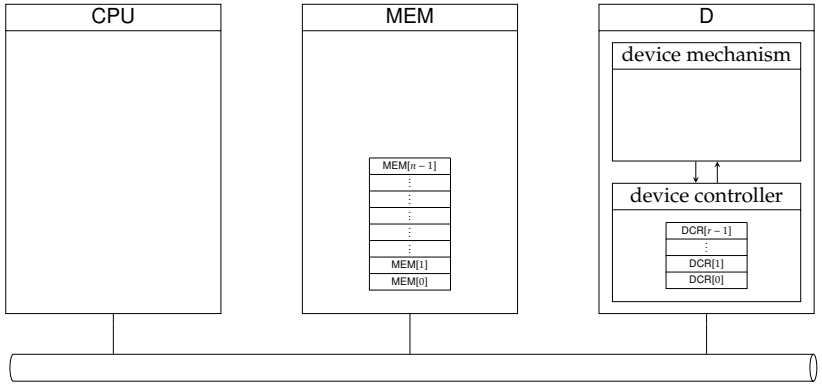
   or
3. a **network device**.

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.

Concept (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**.

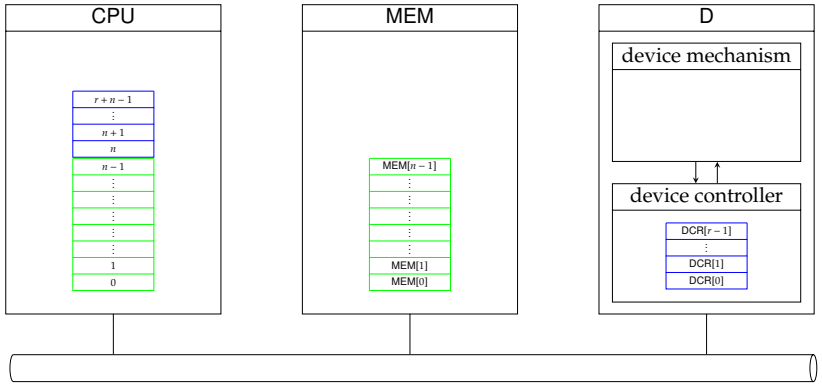
Notes:

- 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 naturally with programming languages, in the sense they will include constructs to access memory anyway; port-mapped I/O demands use of dedicated instructions, which are less often (i.e., not) exposed in such languages and so dictate the use of (inline) assembly language.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms; for example, the MMU will naturally protect access to a mapped address (via flag in the associated PTE) by an unprivileged instruction. In contrast, port-mapped I/O may not: especially where multiple buses are used, there may be a need for a dedicated I/O MMU, or otherwise protection is a matter of privilege (i.e., user or kernel mode).
  - 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.

Concept (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 ⟨cdp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

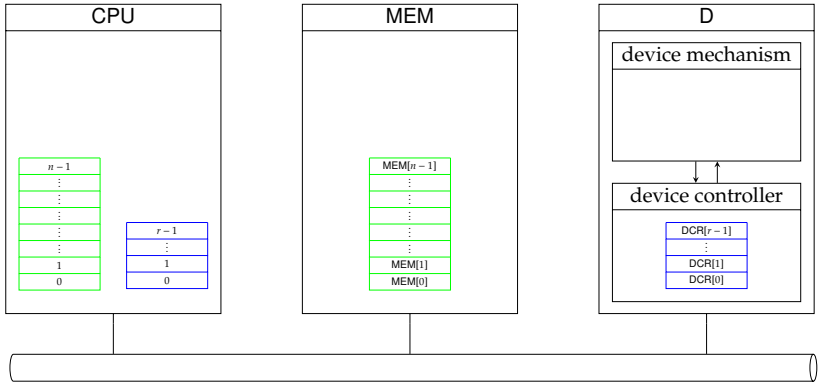
Notes:

- 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 naturally with programming languages, in the sense they will include constructs to access memory anyway; port-mapped I/O demands use of dedicated instructions, which are less often (i.e., not) exposed in such languages and so dictate the use of (inline) assembly language.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms; for example, the MMU will naturally protect access to a mapped address (via flag in the associated PTE) by an unprivileged instruction. In contrast, port-mapped I/O may not: especially where multiple buses are used, there may be a need for a dedicated I/O MMU, or otherwise protection is a matter of privilege (i.e., user or kernel mode).
  - 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.

Concept (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**.

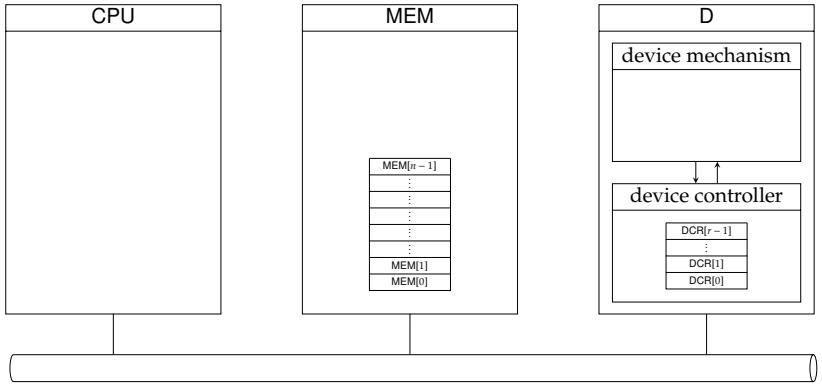
Notes:

- 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 naturally with programming languages, in the sense they will include constructs to access memory anyway; port-mapped I/O demands use of dedicated instructions, which are less often (i.e., not) exposed in such languages and so dictate the use of (inline) assembly language.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms; for example, the MMU will naturally protect access to a mapped address (via flag in the associated PTE) by an unprivileged instruction. In contrast, port-mapped I/O may not: especially where multiple buses are used, there may be a need for a dedicated I/O MMU, or otherwise protection is a matter of privilege (i.e., user or kernel mode).
  - 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.

Concept (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 ⟨cdp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

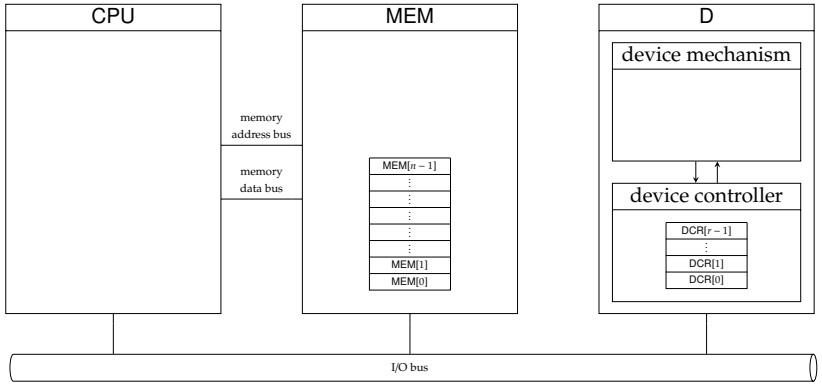
Notes:

- 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 naturally with programming languages, in the sense they will include constructs to access memory anyway; port-mapped I/O demands use of dedicated instructions, which are less often (i.e., not) exposed in such languages and so dictate the use of (inline) assembly language.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms; for example, the MMU will naturally protect access to a mapped address (via flag in the associated PTE) by an unprivileged instruction. In contrast, port-mapped I/O may not: especially where multiple buses are used, there may be a need for a dedicated I/O MMU, or otherwise protection is a matter of privilege (i.e., user or kernel mode).
  - 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.

Concept (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**.

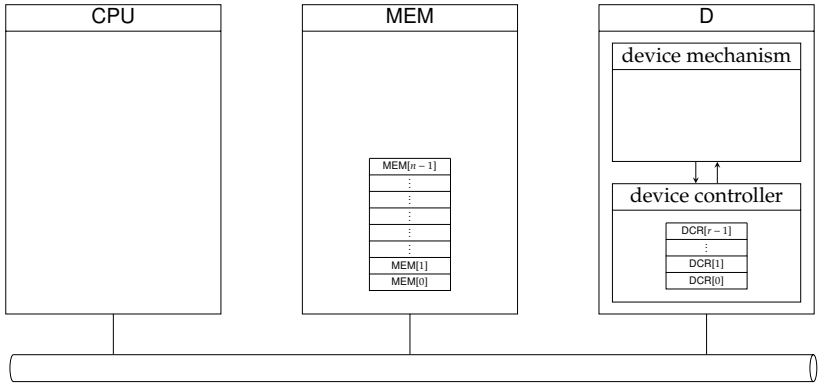
Notes:

- 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 naturally with programming languages, in the sense they will include constructs to access memory anyway; port-mapped I/O demands use of dedicated instructions, which are less often (i.e., not) exposed in such languages and so dictate the use of (inline) assembly language.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms; for example, the MMU will naturally protect access to a mapped address (via flag in the associated PTE) by an unprivileged instruction. In contrast, port-mapped I/O may not: especially where multiple buses are used, there may be a need for a dedicated I/O MMU, or otherwise protection is a matter of privilege (i.e., user or kernel mode).
  - 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.

Concept (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**.

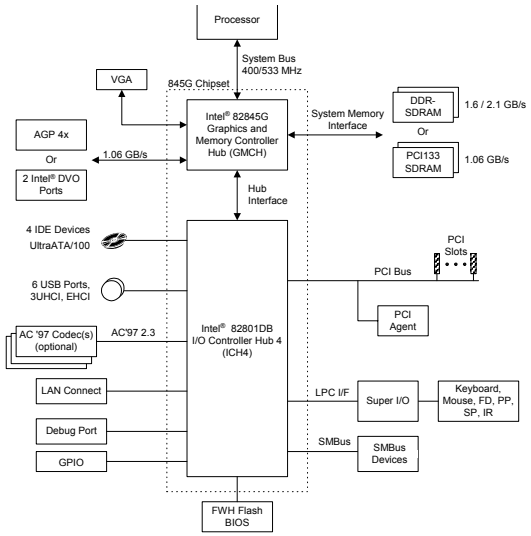
© Daniel Page ⟨cdp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

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 naturally with programming languages, in the sense they will include constructs to access memory anyway; port-mapped I/O demands use of dedicated instructions, which are less often (i.e., not) exposed in such languages and so dictate the use of (inline) assembly language.
  - Use of memory-mapped I/O integrates with existing memory management mechanisms; for example, the MMU will naturally protect access to a mapped address (via flag in the associated PTE) by an unprivileged instruction. In contrast, port-mapped I/O may not: especially where multiple buses are used, there may be a need for a dedicated I/O MMU, or otherwise protection is a matter of privilege (i.e., user or kernel mode).
  - 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.

Example (Intel 845 "Brookdale", circa 2002)



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

Notes:

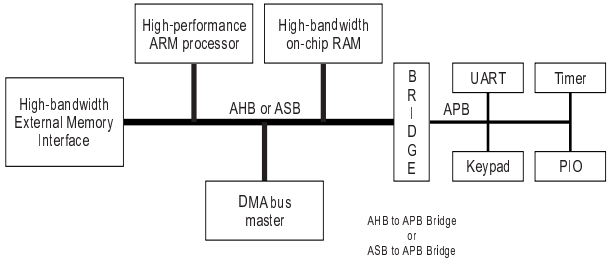
- To quantify "fast" versus slow, you can look at the specification at

  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)

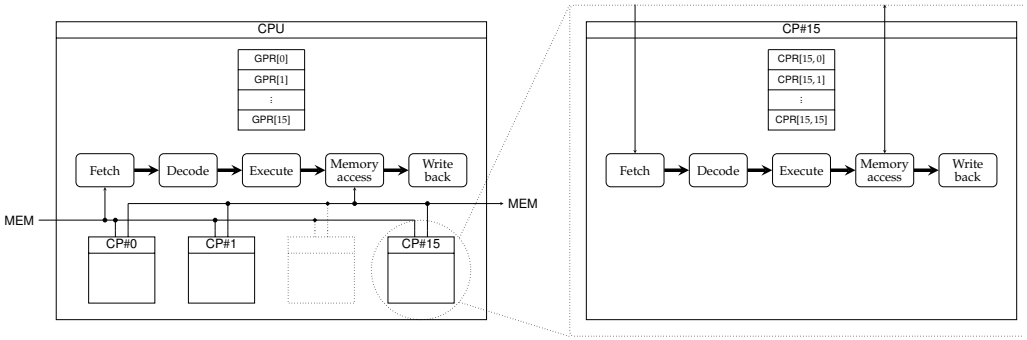


http://infocenter.arm.com/help/topic/com.arm.doc.ihi0011a/index.html

Notes:

- The AMBA specification [10] offers a 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** [11, 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 [13] co-processor for video encoding.

  Per [11, 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.,

     `ldc p0, cr0, [ r1 ]` $\mapsto$ CPR[0, 0] $\leftarrow$ MEM[GPR[1]]
     `stc p0, cr0, [ r1 ]` $\mapsto$ MEM[GPR[1]] $\leftarrow$ CPR[0, 0]

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

     `mrc p0, #0, r1, cr0, cr1, #0` $\mapsto$ GPR[1] $\leftarrow$ CPR[0, 0]
     `mcr p0, #0, r1, cr0, cr1, #0` $\mapsto$ CPR[0, 0] $\leftarrow$ GPR[1]

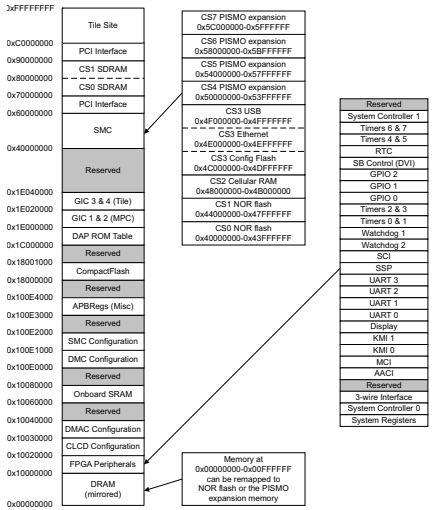
  3. execute a co-processor operation, e.g.,

     `dcp p0, cr0, cr1, cr2, #0` $\mapsto$ CPR[0, 0] $\leftarrow$ CPR[0, 1] + 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).

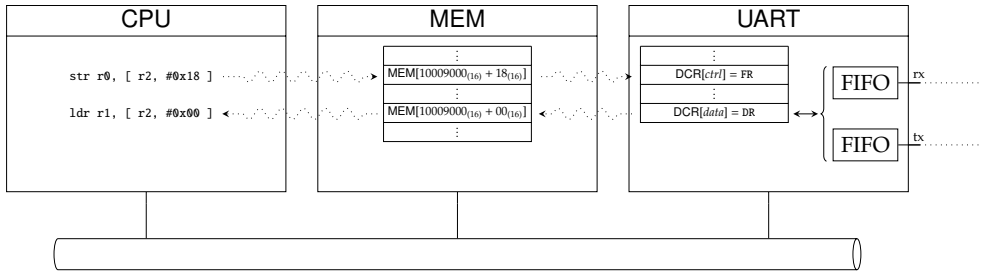
Example (Cortex-A8 Platform Baseboard memory map [15, Figure 4-1])



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

Notes:

- 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;
```

Concept: 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).

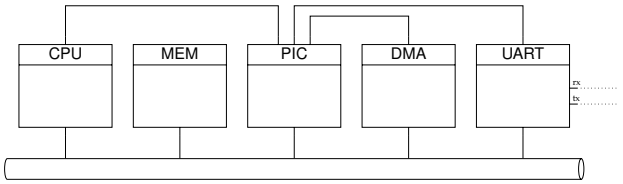
Notes:

- 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.

Concept: I/O programming models (2)

- Example: consider



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

Listing

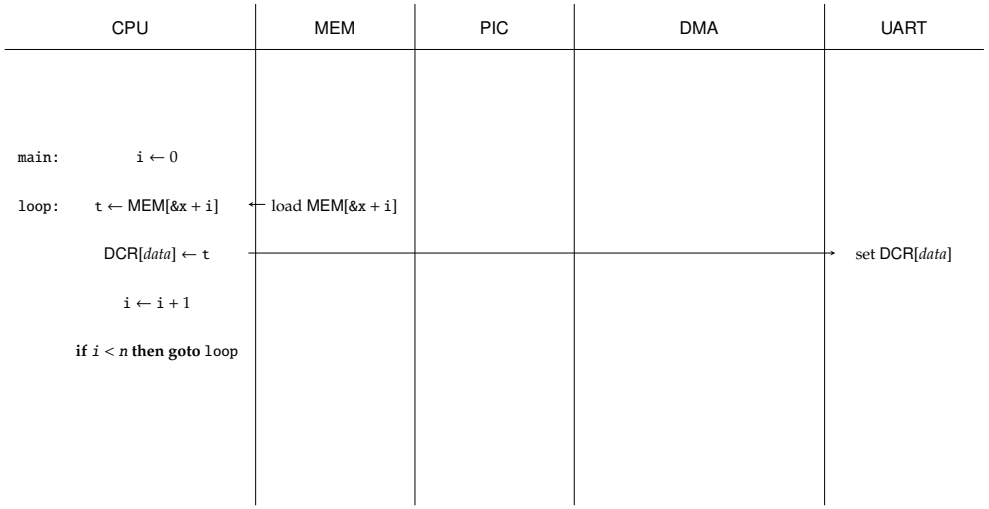
```
1 for( int i = 0; i < n; i++ ) {
2   // wait while transmit FIFO is full
3   while( *( UART0 + 0x18 ) & 0x20 );
4   //             transmit x[ i ]
5   *( UART0 + 0x00 ) = x[ i ];
6 }
```

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.

Concept: I/O programming models (3)

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

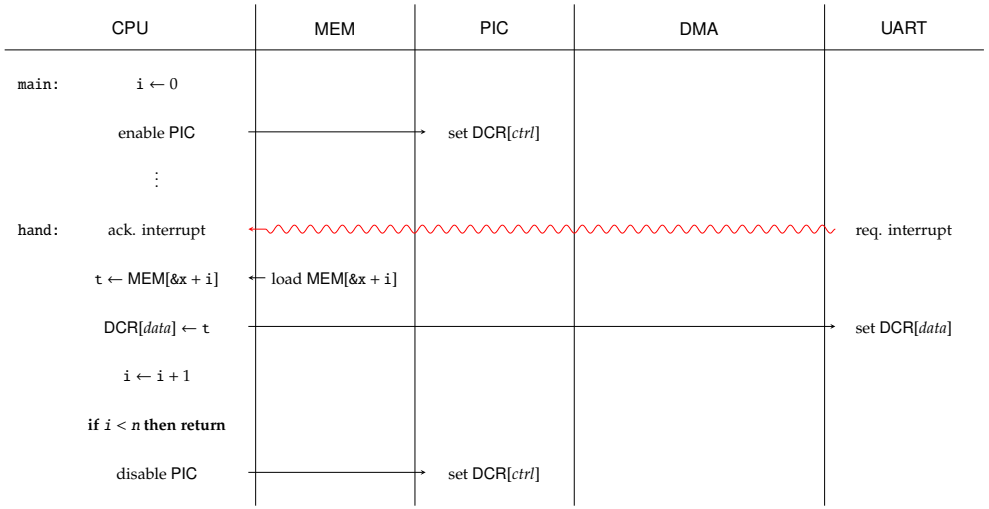


Notes:

- 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.

Concept: 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

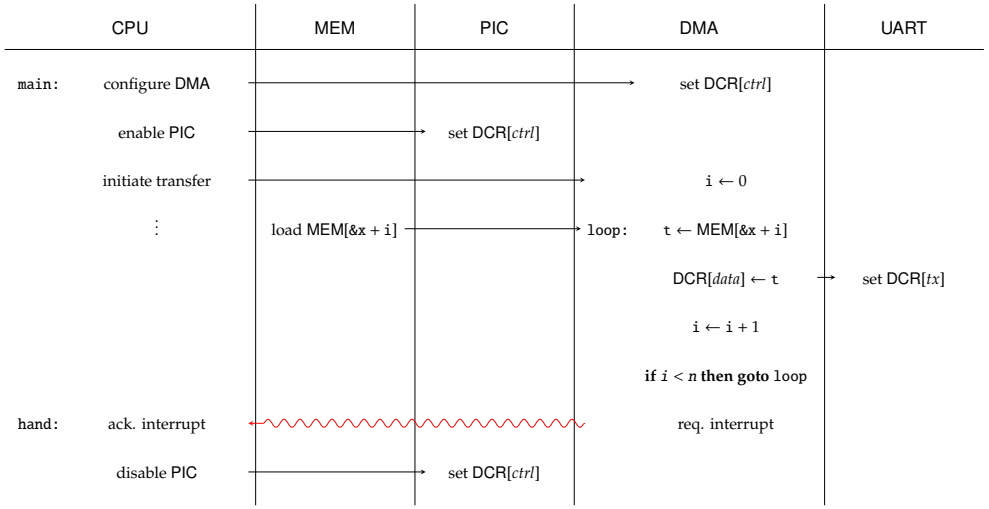


Notes:

- 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.

Concept: 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

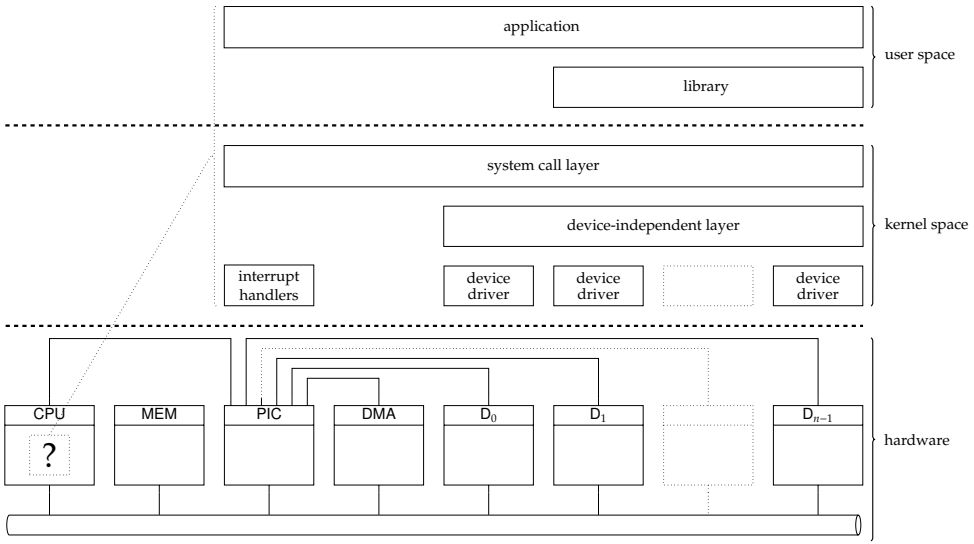
- Take away points:
  1. I/O is hard!
  2. The I/O sub-system must support
     - privilege management via invocation of **mode switches**,
     - the **system call interface**, allowing the kernel and user space software to interact, *and*
     - a suite of device-specific **device drivers**, allowing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

Additional Reading

- *Wikipedia: Interrupt*. URL: http://en.wikipedia.org/wiki/Interrupt.
- J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 9: Communicting with hardware". In: *Linux Device Drivers*. 3rd ed. http://www.makelinux.net/ldd3/. O'Reilly, 2005.
- J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 10: Interrupt handling". In: *Linux Device Drivers*. 3rd ed. http://www.makelinux.net/ldd3/. O'Reilly, 2005.
- J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 15: Memory mapping and DMA". . In: *Linux Device Drivers*. 3rd ed. http://www.makelinux.net/ldd3/. O'Reilly, 2005.
- A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 13: I/O systems". In: *Operating System Concepts*. 9th ed. Wiley, 2014.
- A.S. Tanenbaum and H. Bos. "Chapter 5: Input/output". In: *Modern Operating Systems*. 4th ed. Pearson, 2015.
- A. N. Sloss, D. Symes, and C. Wright. "Chapter 9: Exception and interrupt handling". In: *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier, 2004.

Notes:

References

[1] *Wikipedia: Interrupt*. URL: http://en.wikipedia.org/wiki/Interrupt (see p. 95).

[2] J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 10: Interrupt handling". In: *Linux Device Drivers*. 3rd ed. http://www.makelinux.net/ldd3/. O'Reilly, 2005 (see p. 95).

[3] J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 15: Memory mapping and DMA". In: *Linux Device Drivers*. 3rd ed. http://www.makelinux.net/ldd3/. O'Reilly, 2005 (see p. 95).

[4] J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 9: Communicting with hardware". In: *Linux Device Drivers*. 3rd ed. http://www.makelinux.net/ldd3/. O'Reilly, 2005 (see p. 95).

[5] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 13: I/O systems". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 95).

[6] A. N. Sloss, D. Symes, and C. Wright. "Chapter 9: Exception and interrupt handling". In: *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier, 2004 (see p. 95).

[7] A.S. Tanenbaum and H. Bos. "Chapter 5: Input/output". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see p. 95).

[8] J.E. Smith and A.R. Pleszkun. "Implementing Precise Interrupts in Pipelined Processors". In: *IEEE Transactions On Computers* 37.5 (1998), pp. 562–573 (see p. 15).

[9] W. Walker and H.G. Cragon. "Interrupt Processing in Concurrent Processors". In: *IEEE Computer* 28.6 (1995), pp. 36–46 (see pp. 7, 13, 15, 16).

[10] ARM Limited. *AMBA Specification*. Tech. rep. IHI-0011A. http://infocenter.arm.com/help/topic/com.arm.doc.set.amba/index.html. 1999 (see p. 72).

[11] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. Tech. rep. DDI-0406C. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html. 2014 (see pp. 8, 10, 12, 18, 20–22, 25–27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 73, 74).

[12] ARM Limited. *Cortex-A8 Technical Reference Manual*. Tech. rep. DDI-0344K. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html. 2010 (see pp. 18, 20, 25, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48).

Notes:

References

[13] ARM Limited. *MOVE Co-processor Technical Reference Manual*. Tech. rep. DDI-0235. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0235c/index.html. 2004 (see p. 74).

[14] ARM Limited. *PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual*. Tech. rep. DDI-0181E. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0181e/index.html. 2004 (see pp. 17, 19).

[15] ARM Limited. *RealView Platform Baseboard for Cortex-A8*. Tech. rep. HBI-0178. http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/index.html. 2011 (see p. 77).

© Daniel Page (<redacted>)
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes: