Concurrent Computing (Operating Systems)

Daniel Page

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

January 28, 2016

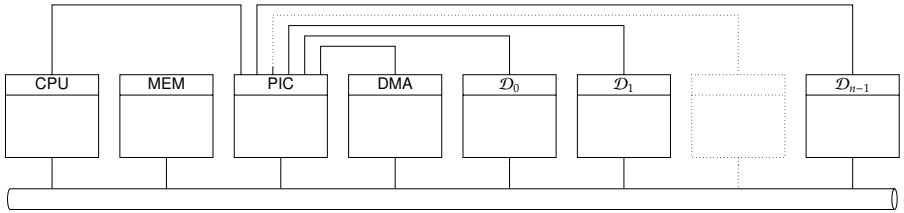
Keep in mind there are *two* PDFs available (of which this is the latter):

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

Notes:

Notes:

- 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?
- 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).

Concepts (1)

Definition (interrupt [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., 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 (interrupt handler [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 [3, 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.
- 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 *raise* 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).

Concepts (2)

- The overall process of handling an interrupt is
  1. detect the interrupt,

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.
- [3, 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).

Concepts (2)

- The overall process of handling an interrupt is
  1. detect the interrupt,
  2. update the processor mode (e.g., to kernel mode),

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.
- [3, 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).

Concepts (2)

- The overall process of handling an interrupt is
  1. detect the interrupt,
  2. update the processor mode (e.g., to kernel mode),
  3. save the processor state, e.g., using
     - **shadow registers**, or
     - **shadow stacks**

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.
- [3, 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).

Concepts (2)

- The overall process of handling an interrupt is
  1. detect the interrupt,
  2. update the processor mode (e.g., to kernel mode),
  3. save the processor state, e.g., using
     - **shadow registers**, or
     - **shadow stacks**
  4. execute an (or the) interrupt handler,

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.
- [3, 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).

Concepts (2)

- The overall process of handling an interrupt is
  1. detect the interrupt,
  2. update the processor mode (e.g., to kernel mode),
  3. save the processor state, e.g., using
     - **shadow registers**, or
     - **shadow stacks**
  4. execute an (or the) interrupt handler,
  5. update the processor mode (e.g., to user mode),

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.
- [3, 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).

Concepts (2)

- The overall process of handling an interrupt is
  1. detect the interrupt,
  2. update the processor mode (e.g., to kernel mode),
  3. save the processor state, e.g., using
     - **shadow registers**, or
     - **shadow stacks**
  4. execute an (or the) interrupt handler,
  5. update the processor mode (e.g., to user mode),
  6. restart the interrupted 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.
- [3, 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).

Concepts (3)

Definition (**interrupt controller** [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:

Concepts (4)

Definition (**(non-)maskable interrupt** [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 (**(im)precise interrupt** [9, 7])

An interrupt is deemed

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

Per [9, 7], well-defined is taken to mean

1. the value of PC is retained somehow,
2. all instructions before PC have completed,
3. no instructions after PC have completed, and
4. the execution state of instruction at PC is known.

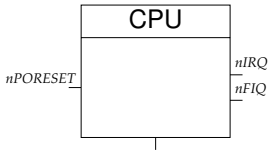
Notes:

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

- Interrupt detection is managed automatically by the processor



st.

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

Notes:

- [?, 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 raise a software-based trap; this was latterly replaced by the `svc` (or "supervisor call") instruction. Keep in mind the two are identical wrt. encoding (i.e., bar the mnemonic), *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 [3, 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)

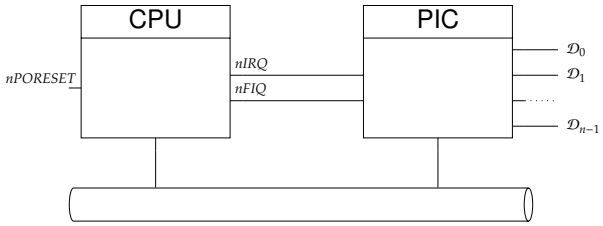
```
1                 ; disable FIQ interrupts
2 mrs r0,   cpsr  ; load      CPSR (control bits only)
3 orr r0, r0, #0x40 ; set   bit 6 of CPSR, i.e., F flag
4 msr cpsr_c, r0  ; store new CPSR
5                 ; disable IRQ interrupts
6 mrs r0,   cpsr  ; load      CPSR (control bits only)
7 orr r0, 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 (1) – Cortex-A8: step #1 detect the interrupt

- Interrupt detection is managed automatically by the processor



st.

- an interrupt can be requested, e.g., by
  - a software trap,
  - 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 [1]).

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # def8a0f @ 2016-01-26 University of BRISTOL

Notes:

- [?, 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 raise a software-based trap; this was latterly replaced by the `svc` (or "supervisor call") instruction. Keep in mind the two are identical wrt. encoding (i.e., bar the mnemonic), *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 [3, 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 (2) – Cortex-A8: step #2 update the processor mode

ARMv7-A processor modes [3, 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 [3, 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.
- [3, 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 (3) – Cortex-A8: step #3 save the processor state

USR mode	privileged modes							
	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 (4) – Cortex-A8: step #4 execute an interrupt handler

ARMv7-A interrupt handling [3, 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 [2, Table 2-12]

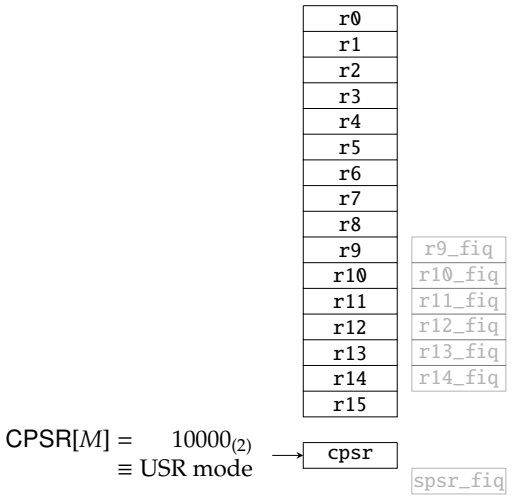
Type	Return offset	Return instruction
Reset	$-0_{(10)}$	
Undefined instruction	$-0_{(10)}$	`movs pc, lr`
Software interrupt	$-0_{(10)}$	`movs pc, lr`
(Pre-)fetch abort	$-4_{(10)}$	`subs pc, lr, #4`
Data abort	$-8_{(10)}$	`subs pc, lr, #8`
IRQ	$-4_{(16)}$	`subs pc, lr, #4`
FIQ	$-4_{(16)}$	`subs pc, lr, #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 [3, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Notes:

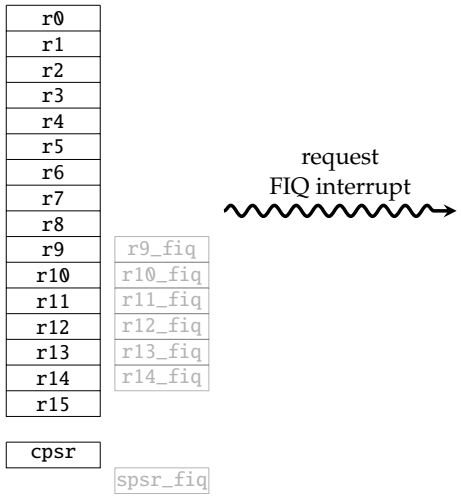
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed 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. [2, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Notes:

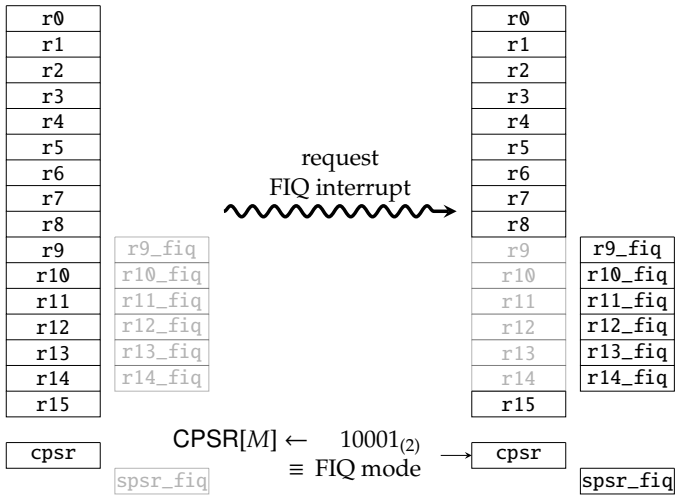
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed 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. [2, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # def8a0f @ 2016-01-26 University of BRISTOL

Notes:

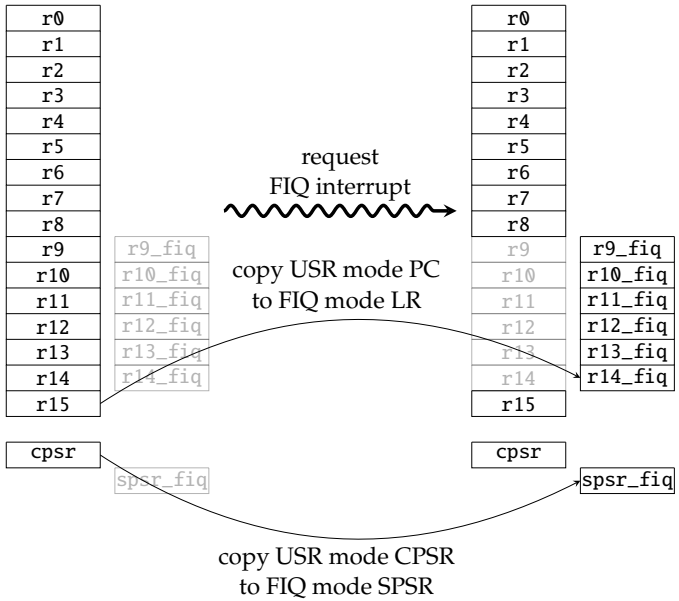
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed 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. [2, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Notes:

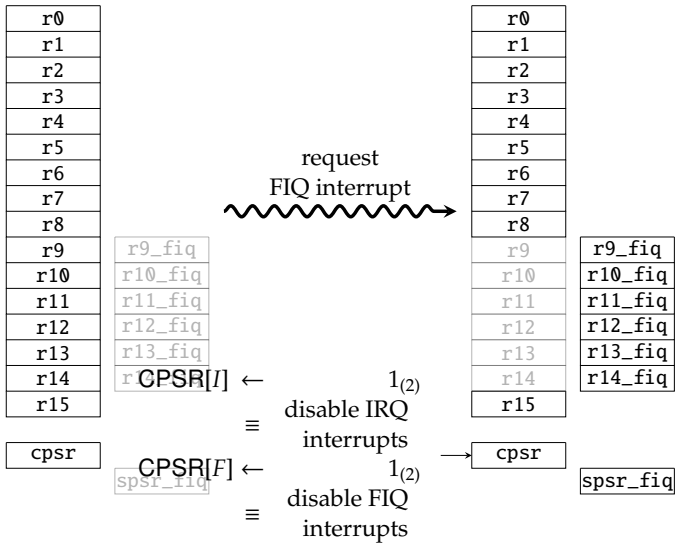
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed 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. [2, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # def8a0f @ 2016-01-26 University of BRISTOL

Notes:

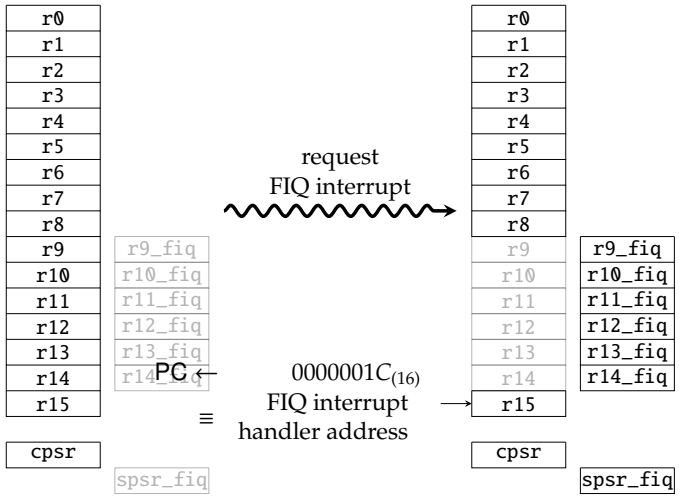
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed 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. [2, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Notes:

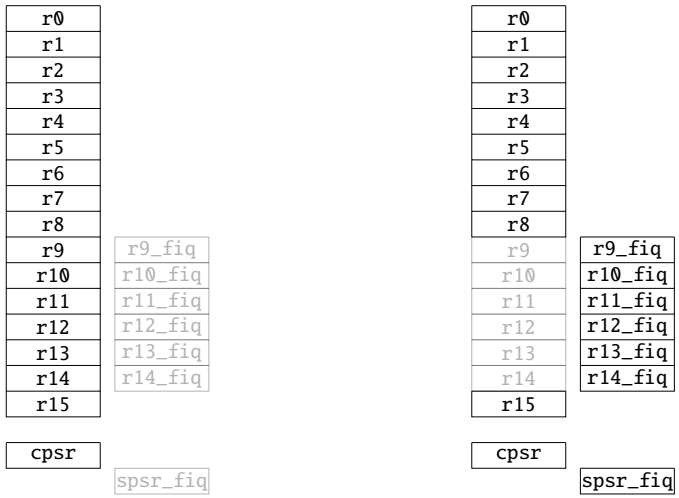
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed 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. [2, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Notes:

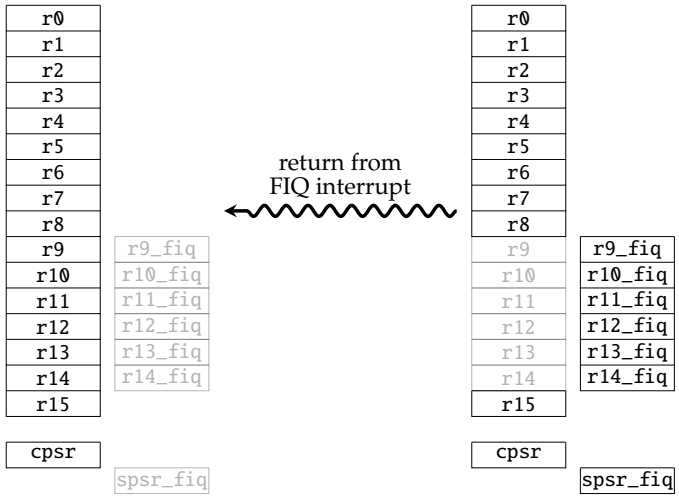
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed 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. [2, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Notes:

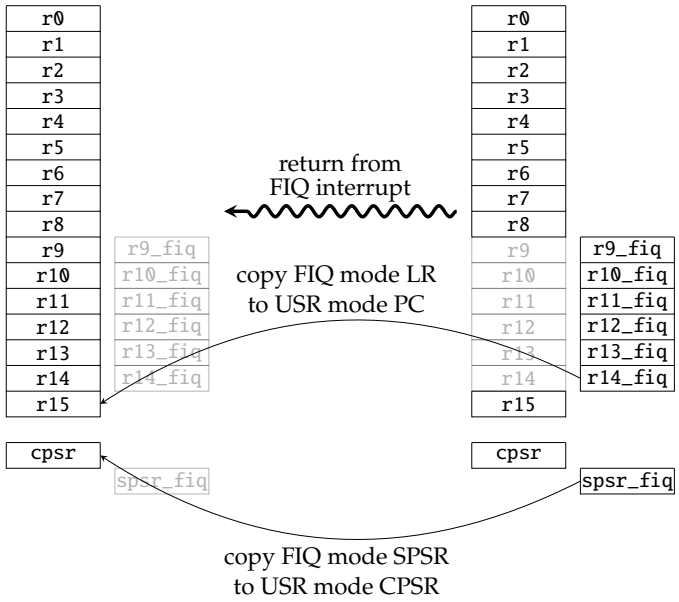
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed 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. [2, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Notes:

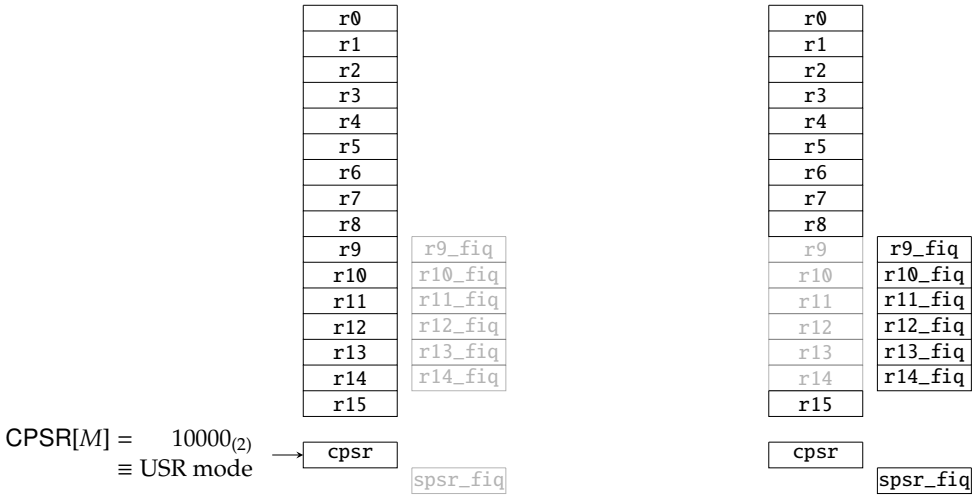
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed 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. [2, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Notes:

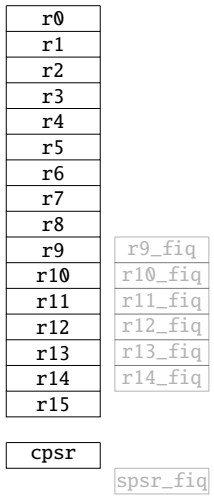
- This is an example for an FIQ interrupt: the same idea is true for other interrupt types, but clearly some more specific details may differ. For instance, for an IRQ interrupt it is *not* true that FIQ interrupts are disabled on entry.
- It's important to keep in mind that some of the steps are performed 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. [2, 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 (5) – Cortex-A8: putting it all together [3, Section B1.9.12]

- Example:



Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # def8a0f @ 2016-01-26 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. [2, Section 2.15.4] says we could execute

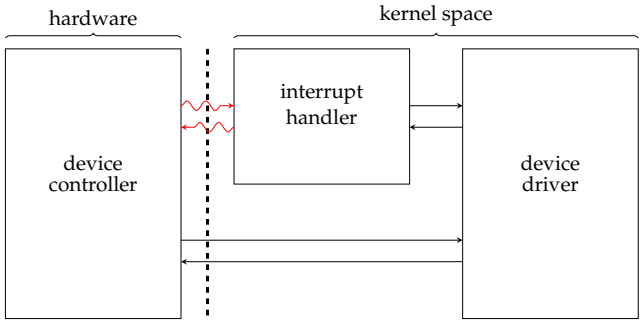
```
subs pc, lr, #4
```

  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 (6) – Cortex-A8: putting it all together [3, 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 `swi 0x0`
  - any return value from the system call is in `r0`

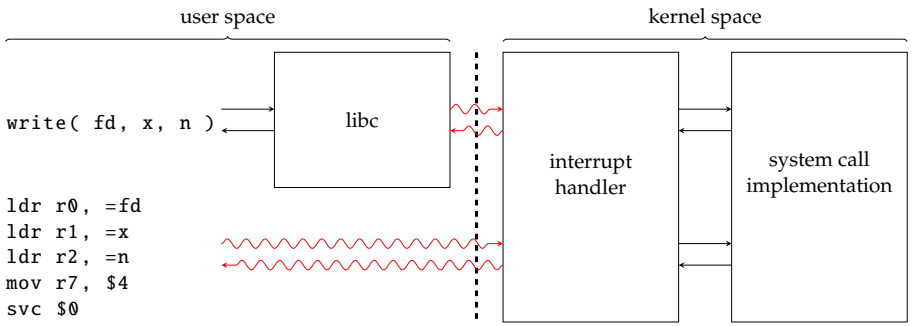
  Not that bar the system call identifier (which you could still think of as analogous to the callee address) and that `swi` replaces a branch, this is much like the AAPCS function calling convention.

- 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 (6) – Cortex-A8: putting it all together [3, 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 ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # def8a0f @ 2016-01-26 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 `swi 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 `swi` replaces a branch, this is much like the AAPCS function calling convention.

- 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.

Continued in next lecture ...

Notes:

References

[1] ARM Limited.
PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual.
Technical Report DDI-0181E, 2004.
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0181e/index.html.

[2] ARM Limited.
Cortex-A8 Technical Reference Manual.
Technical Report DDI-0344K, 2010.
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html.

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

[4] J. Corbet, G. Kroah-Hartman, and A. Rubini.
Chapter 10: Interrupt handling.
In *Linux Device Drivers*. O'Reilly, 3rd edition, 2005.
http://www.makelinux.net/ldd3/.

[5] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 13: I/O systems.
In *Operating System Concepts*. Wiley, 9th edition, 2014.

Notes:

[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.

[7] J.E. Smith and A.R. Pleszkun.
Implementing precise interrupts in pipelined processors.
IEEE Transactions On Computers, 37(5):562–573, 1998.

[8] A.S. Tanenbaum and H. Bos.
Chapter 5: Input/output.
In *Modern Operating Systems*. Pearson, 4th edition, 2015.

[9] W. Walker and H.G. Cragon.
Interrupt processing in concurrent processors.
IEEE Computer Journal, 28(6):36–46, 1995.

Notes: