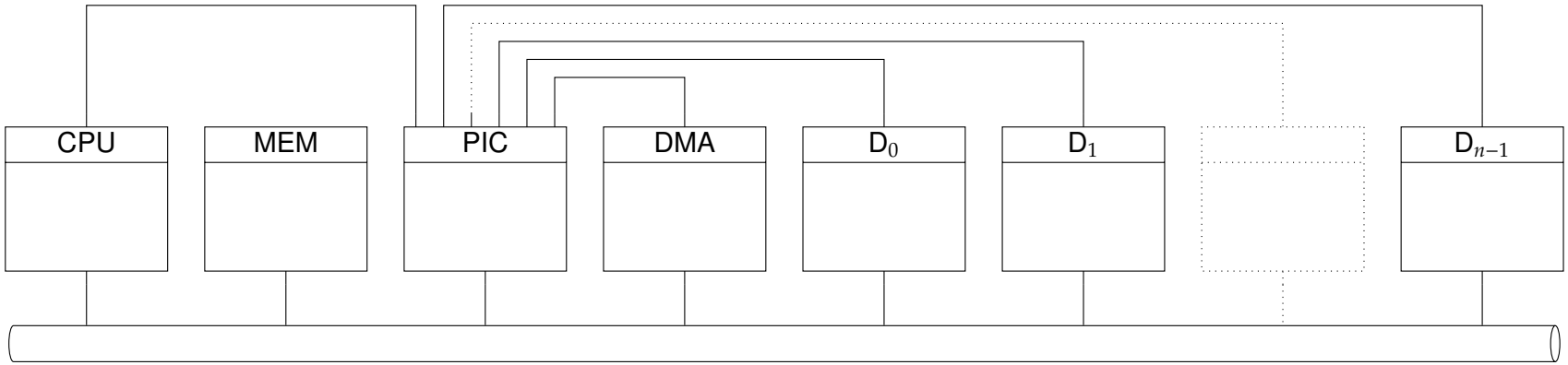
- Problem: our example computer system looks like this



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.

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.

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.

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.

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.

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

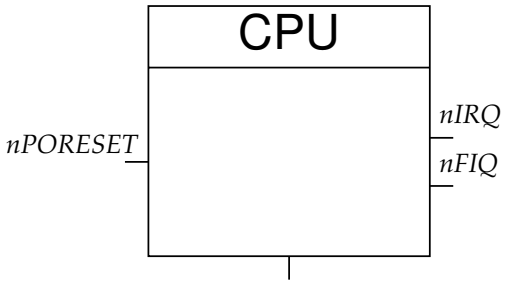
- **precise** if it leaves the processor in a well-defined state, or
- **imprecise** otherwise.

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.

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

- Interrupt detection is managed automatically by the processor

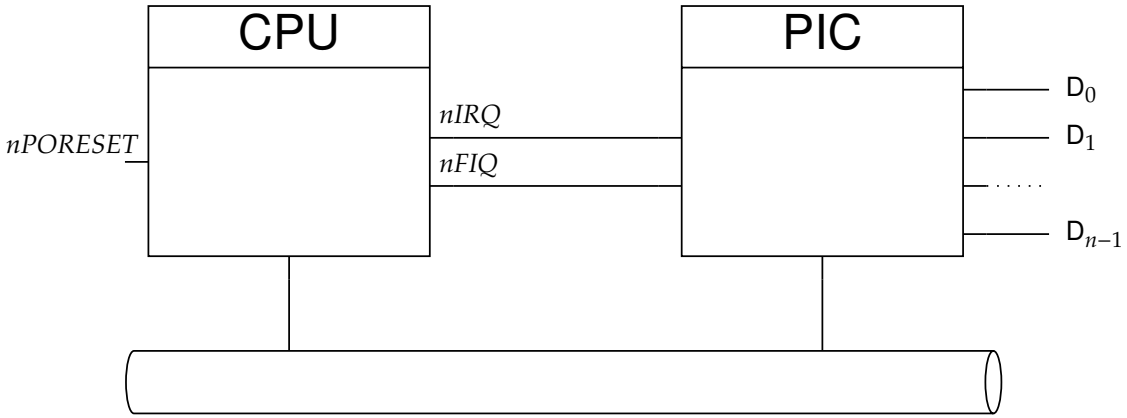


st.

- 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.

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 [12]).

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

ARMv7-A processor modes [10, 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

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	

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

ARMv7-A interrupt handling [10, Tables B1-3 + B1-4]

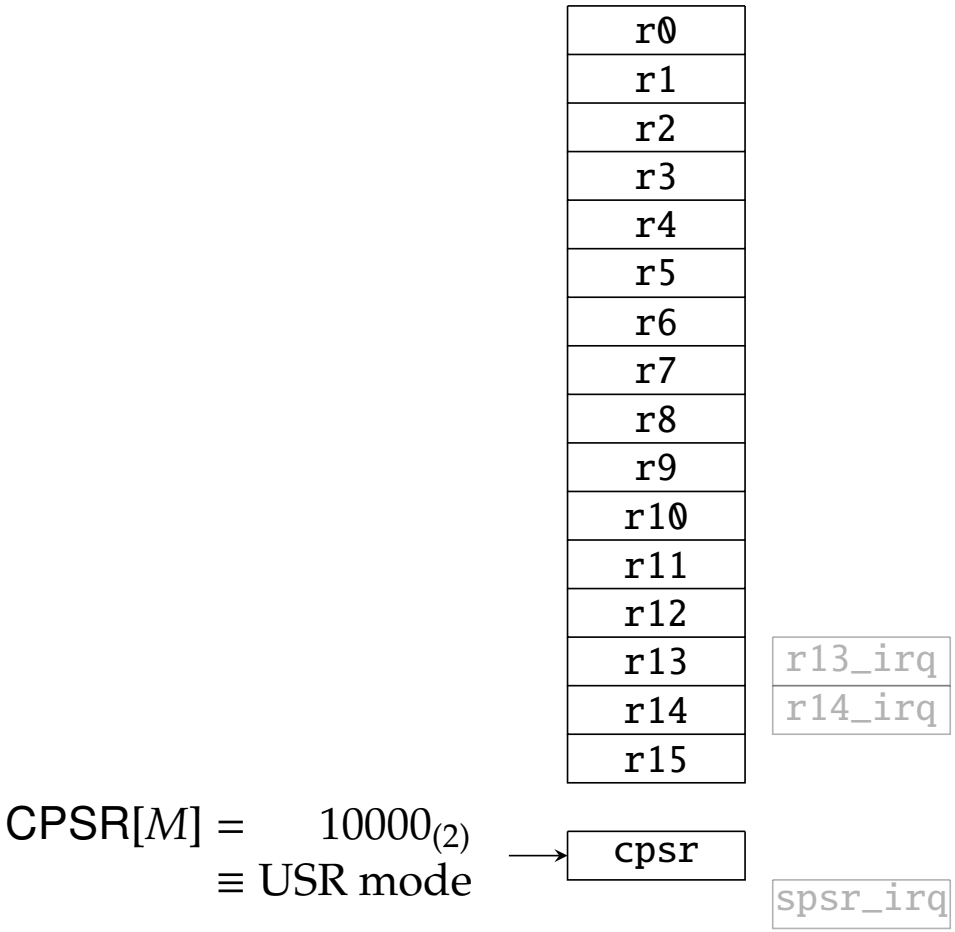
Type	Entry mode	Entry low address	Entry high address
Reset	SVC	$00000000_{(16)}$	$FFFF0000_{(16)}$
Undefined instruction	UND	$00000004_{(16)}$	$FFFF0004_{(16)}$
Software interrupt	SVC	$00000008_{(16)}$	$FFFF0008_{(16)}$
(Pre-)fetch abort	ABT	$0000000C_{(16)}$	$FFFF000C_{(16)}$
Data abort	ABT	$00000010_{(16)}$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

ARMv7-A interrupt handling [11, 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`

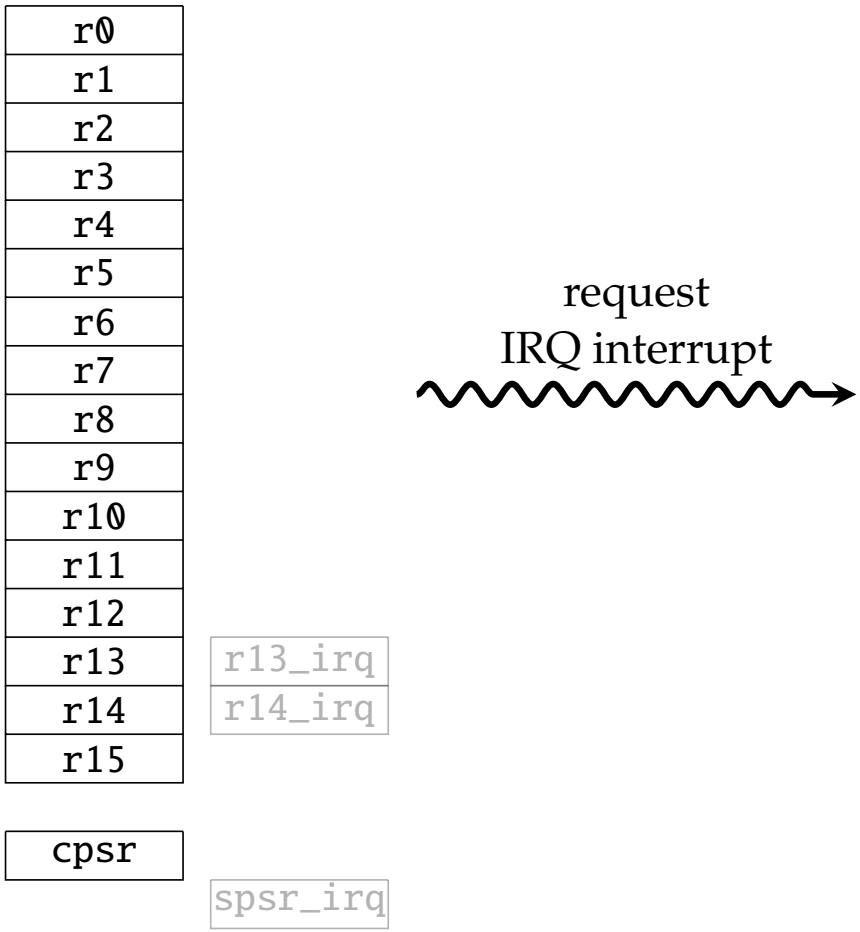
Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

- Example:



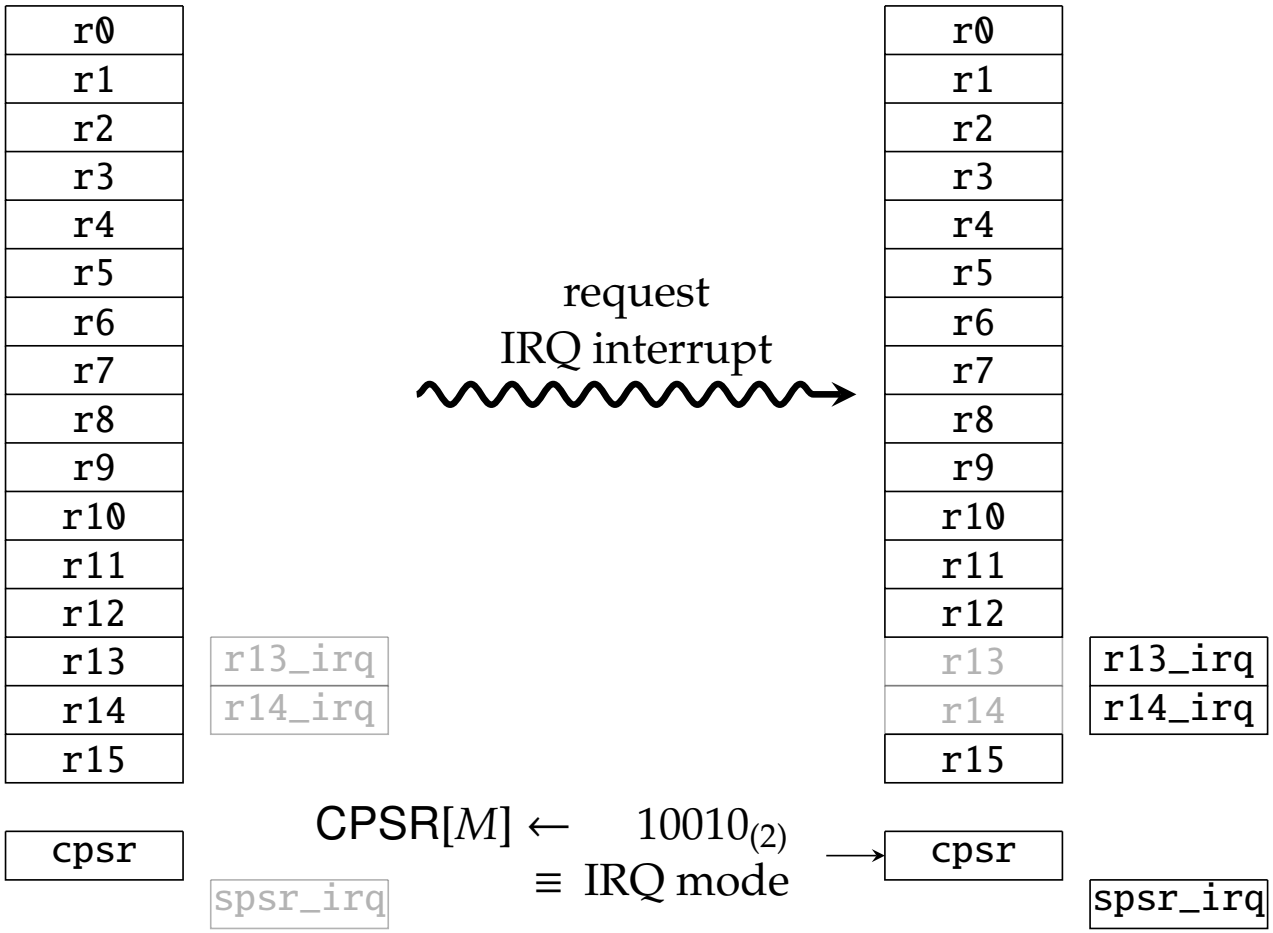
Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

- Example:



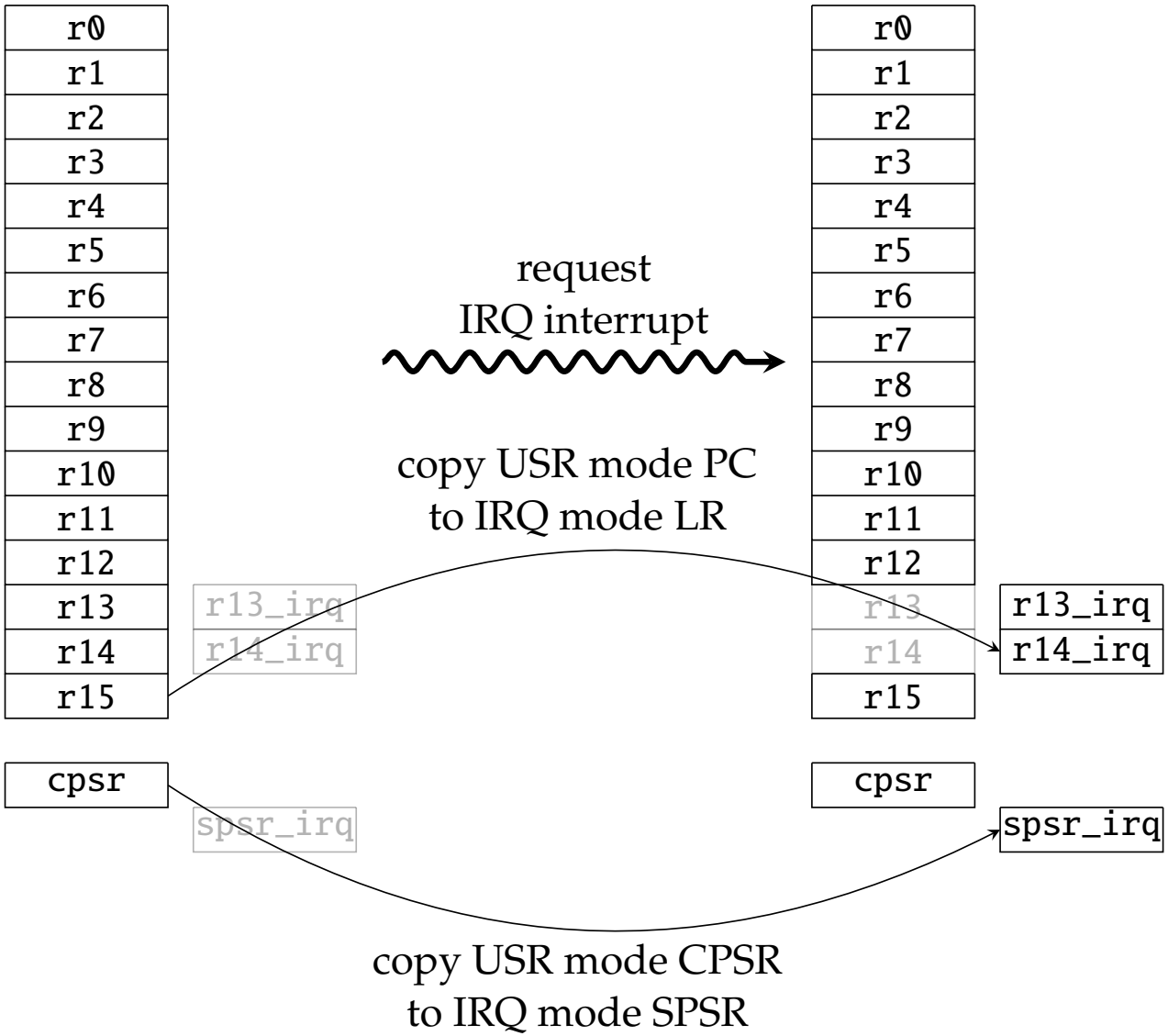
Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

- Example:



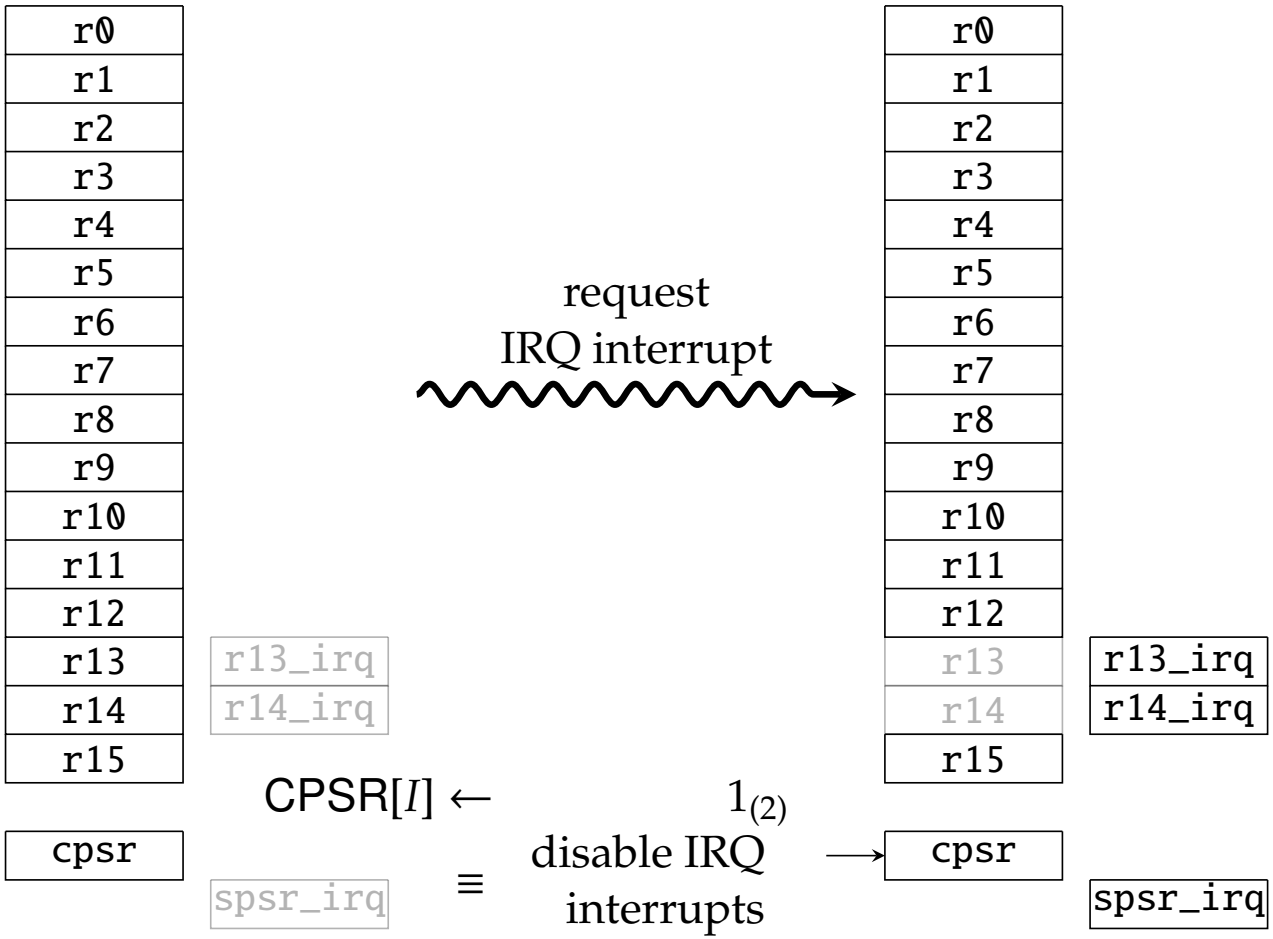
Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

- Example:



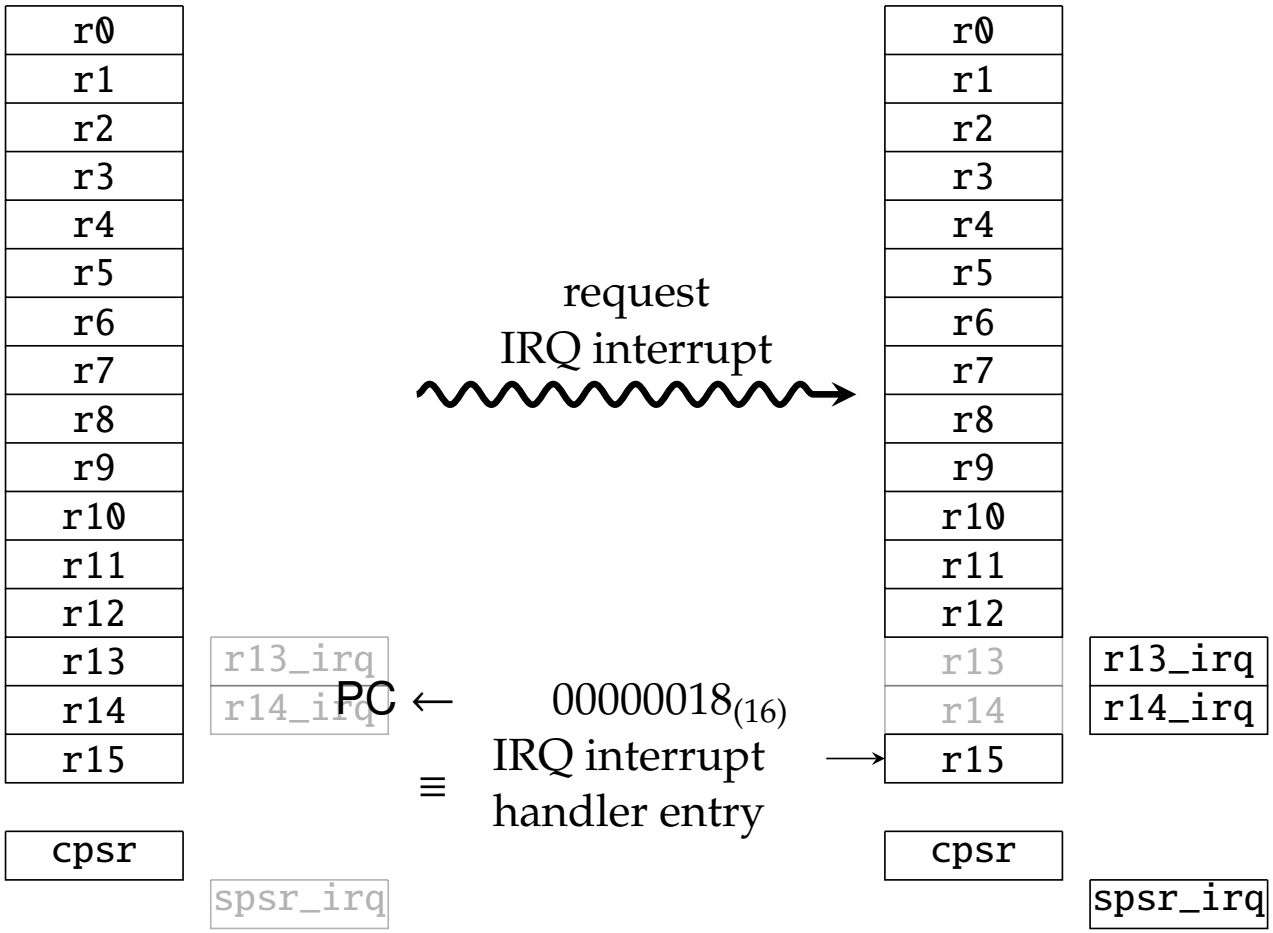
Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

- Example:



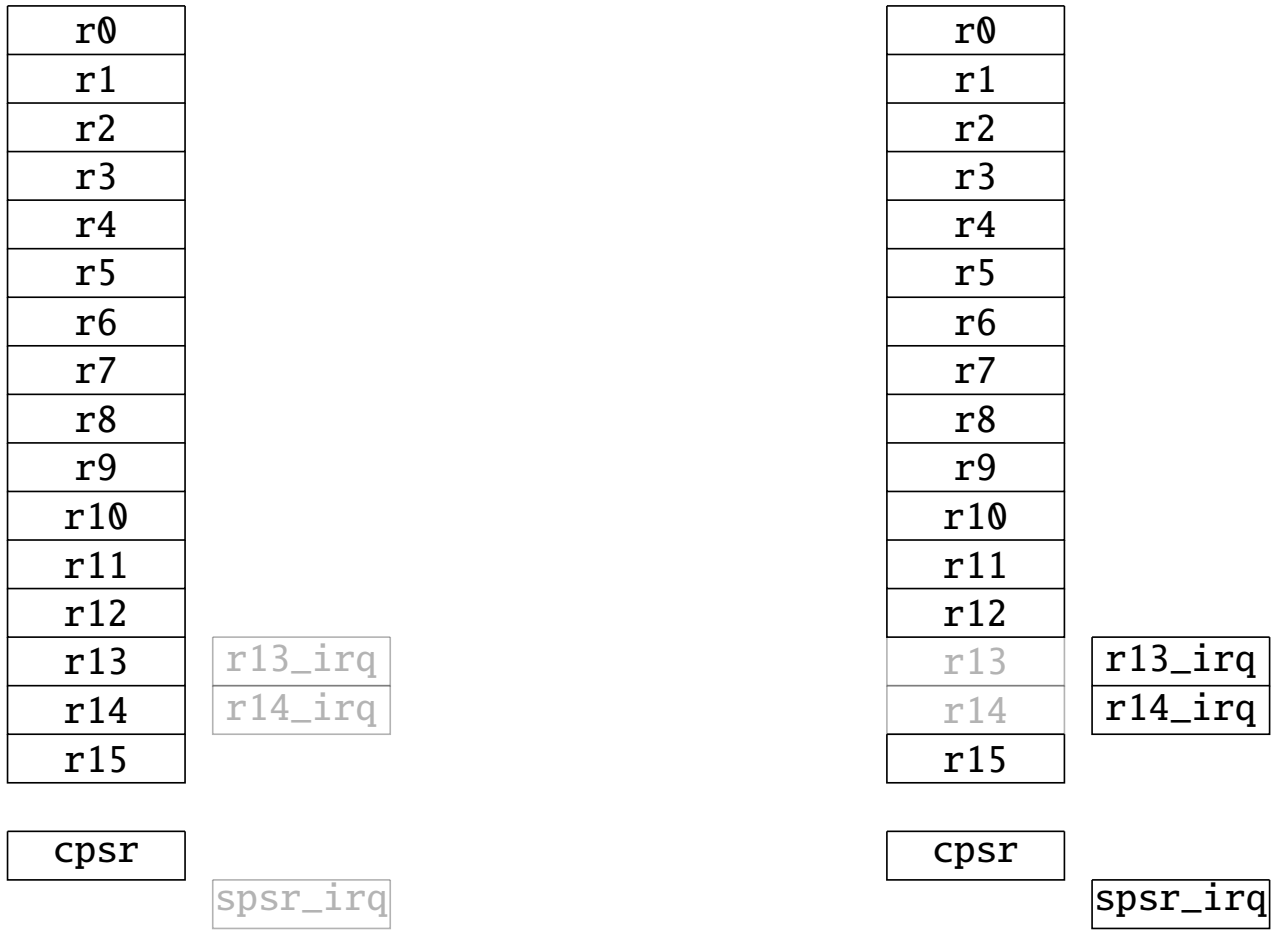
Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

- Example:



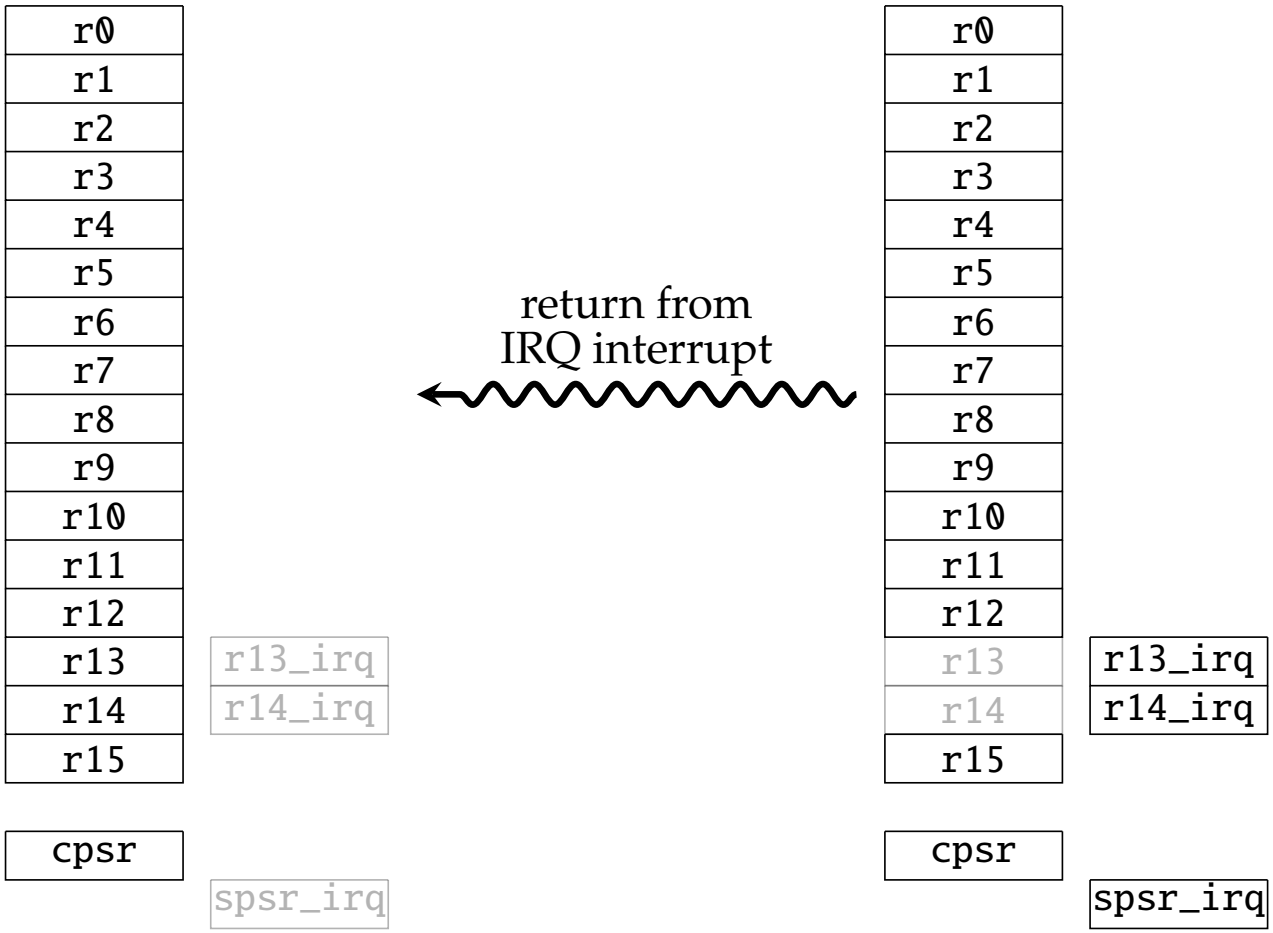
Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

- Example:



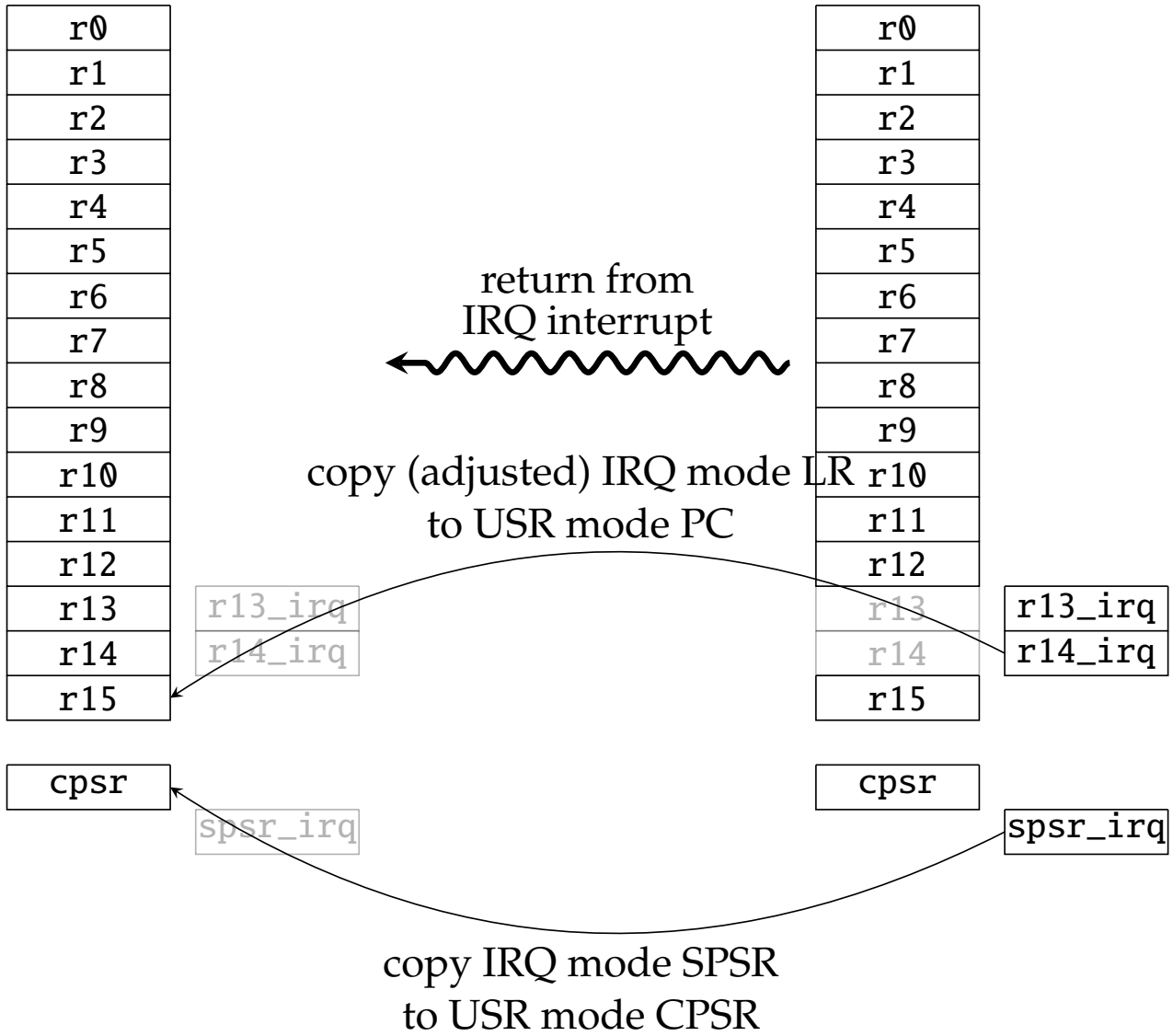
Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

- Example:



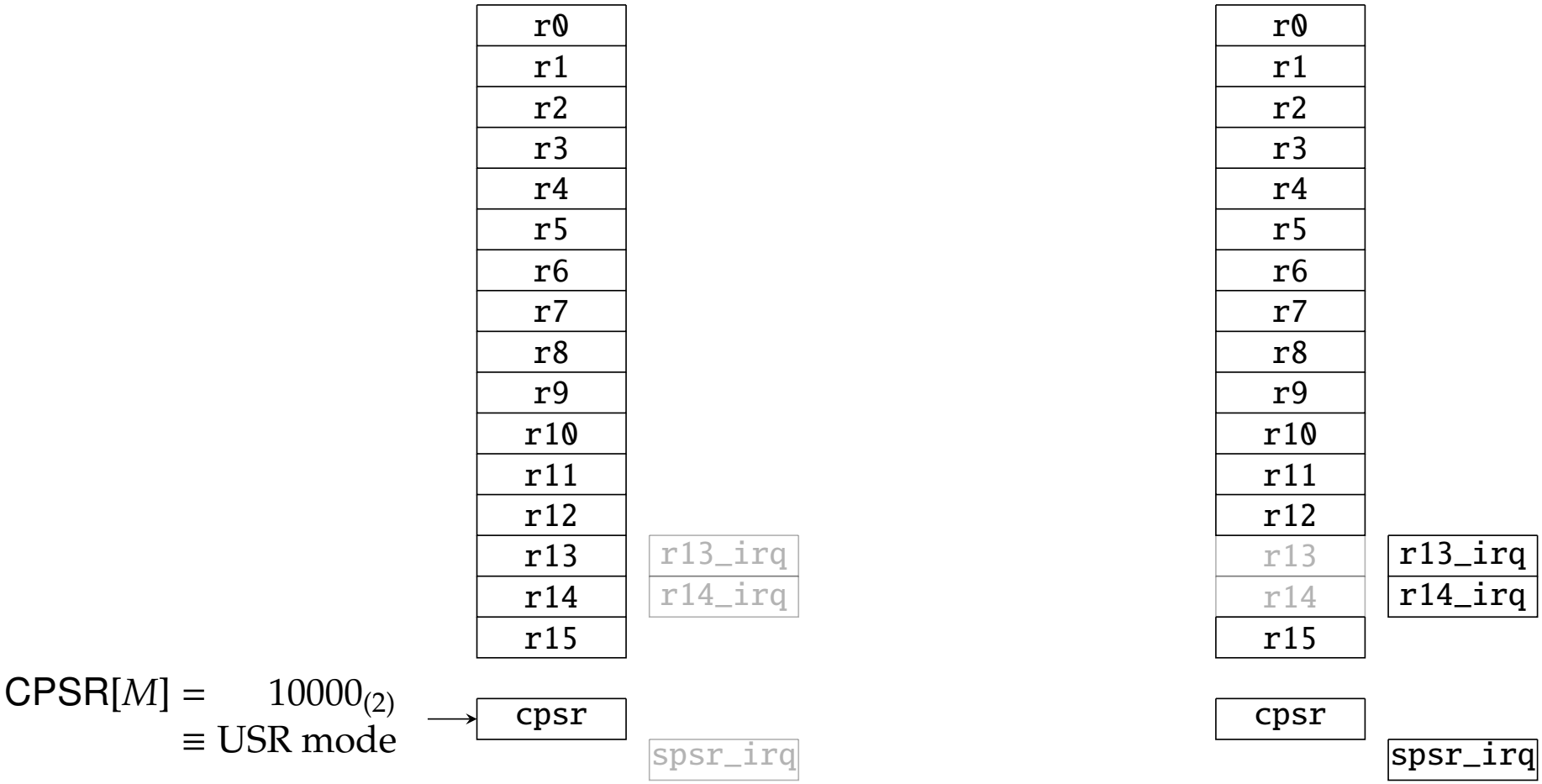
Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

- Example:



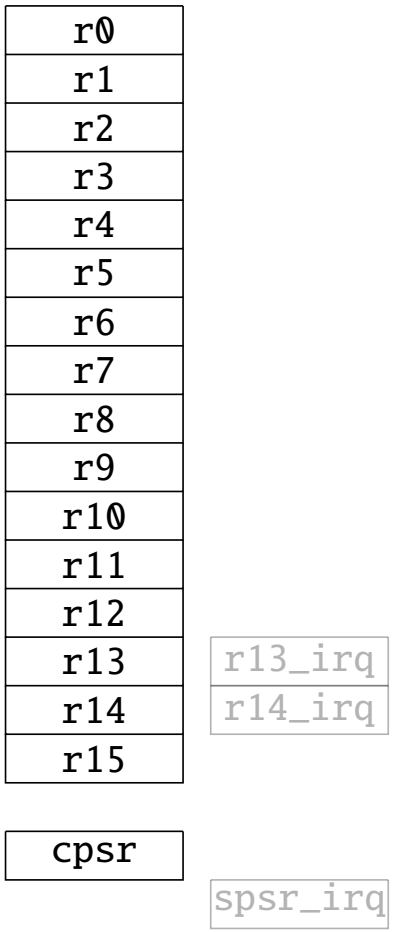
Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

- Example:



Implementation: Cortex-A8 (5) – putting it all together [10, Section B1.9.12]

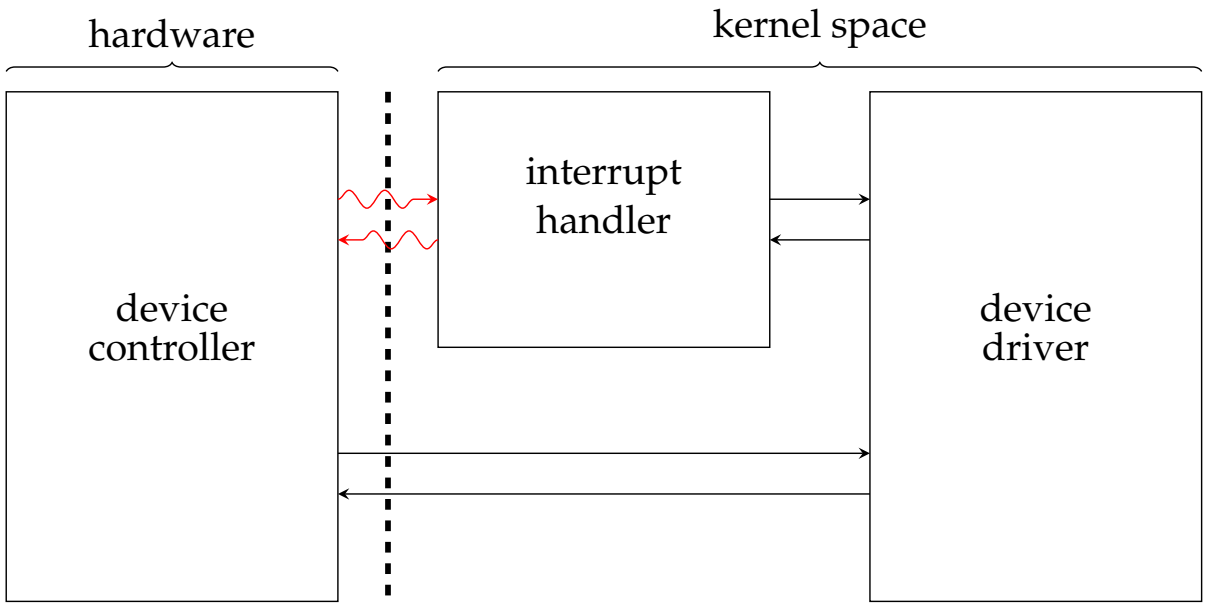
- Example:



Implementation: Cortex-A8 (6) – putting it all together [10, Section B1.9.12]

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

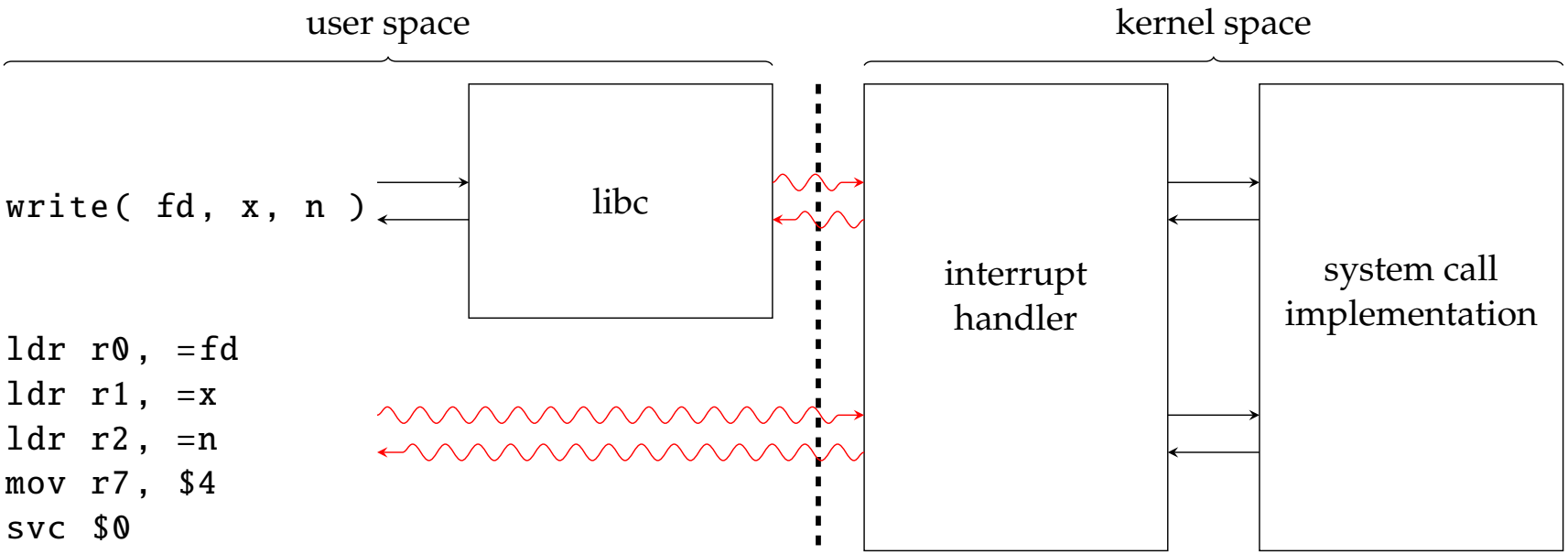
can get attention from (i.e., invoke functionality in) the interrupt-aware kernel, e.g.,



Implementation: Cortex-A8 (6) – putting it all together [10, 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.

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
- components attached to the bus are classified as
  - primary, or active, e.g., transmits requests, or
  - secondary, or passive, e.g., transmits responses.

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

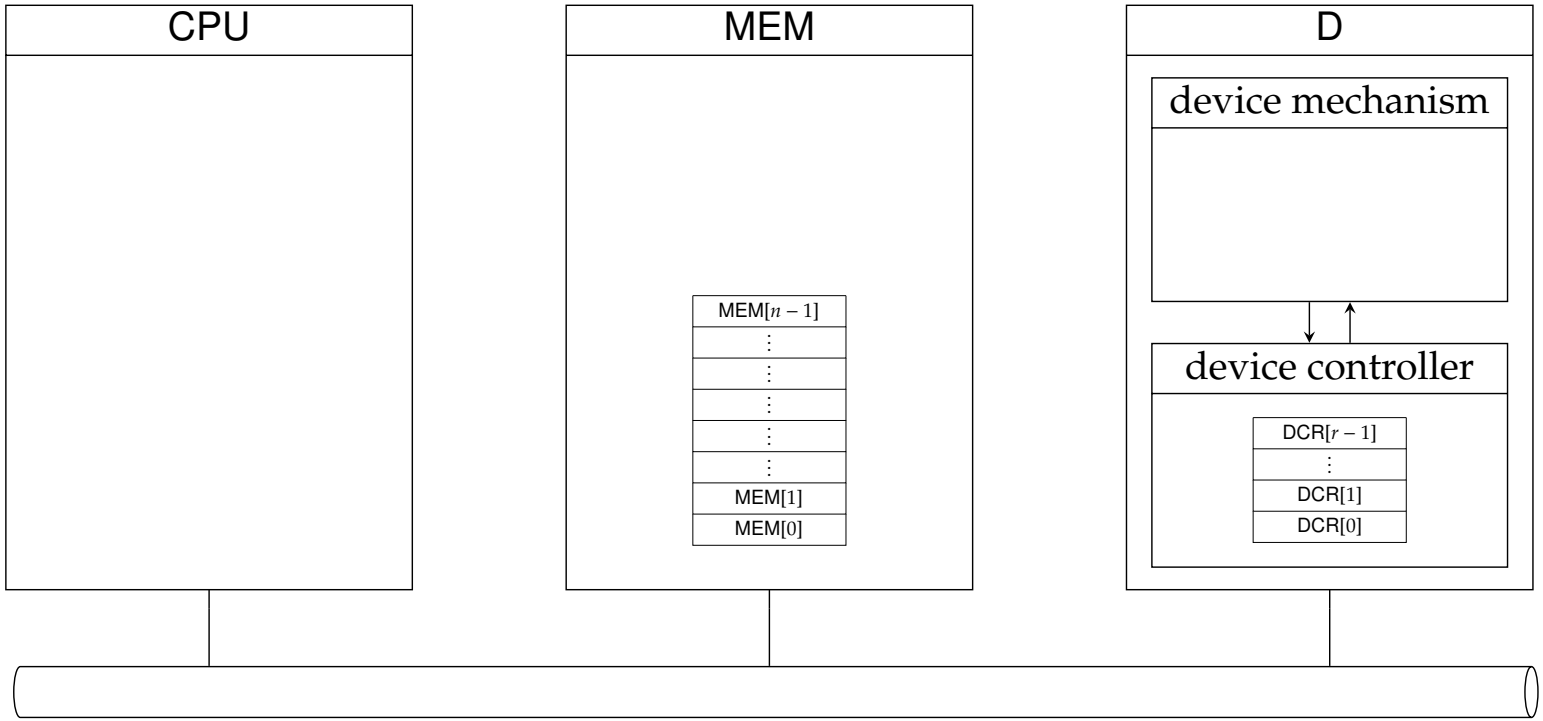
A given device is typically and imperfectly classified as either

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**.

Concept (3)

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



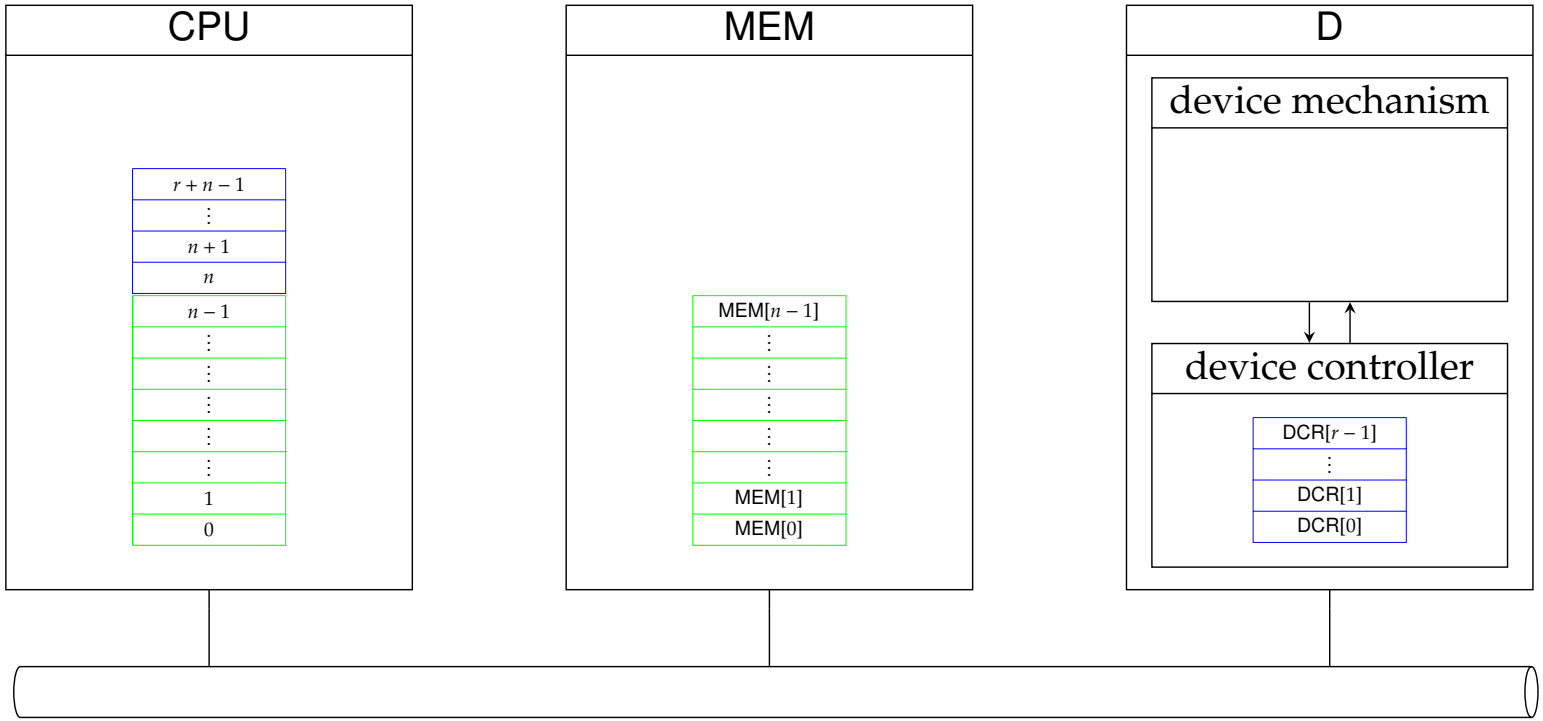
yielding a design space wrt. (at least)

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

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

Concept (3)

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



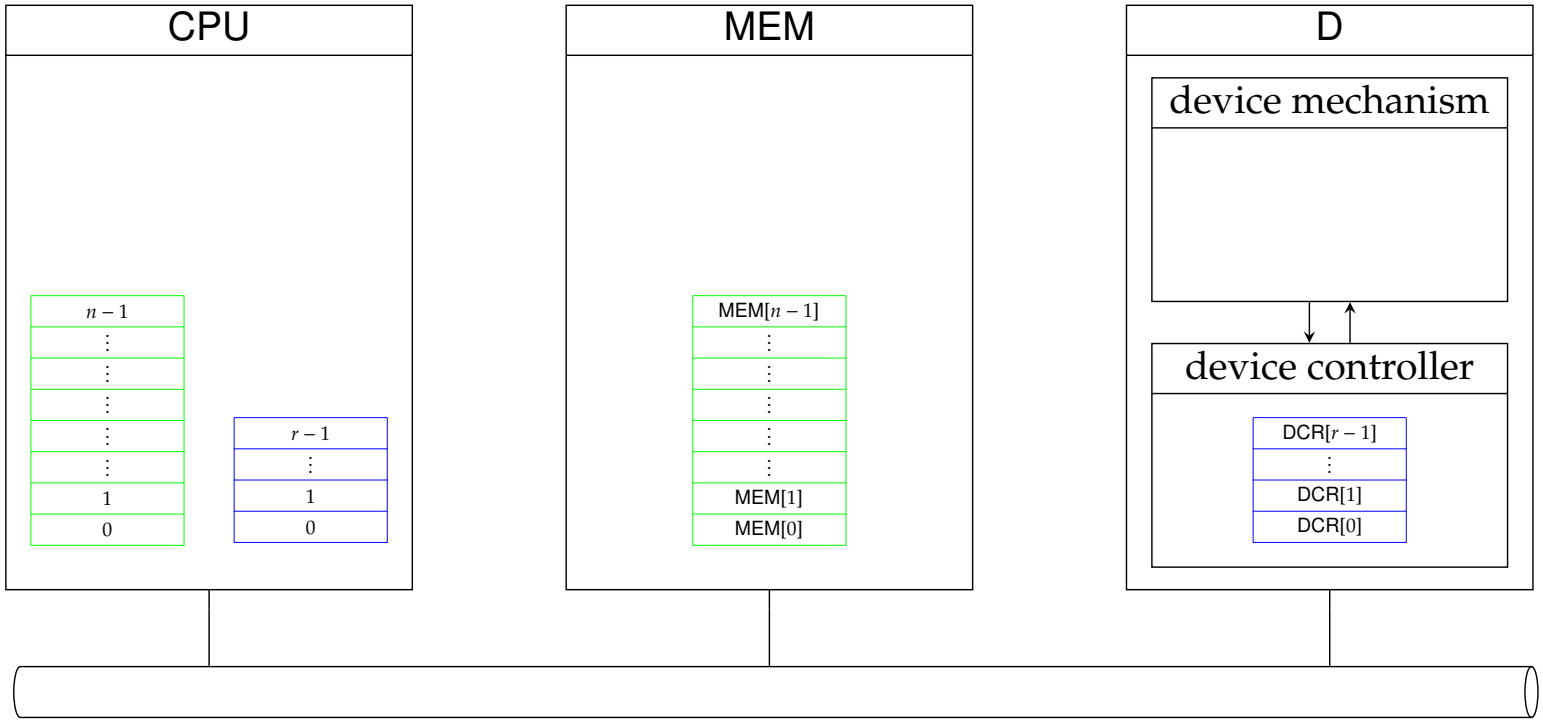
yielding a design space wrt. (at least)

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

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

Concept (3)

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



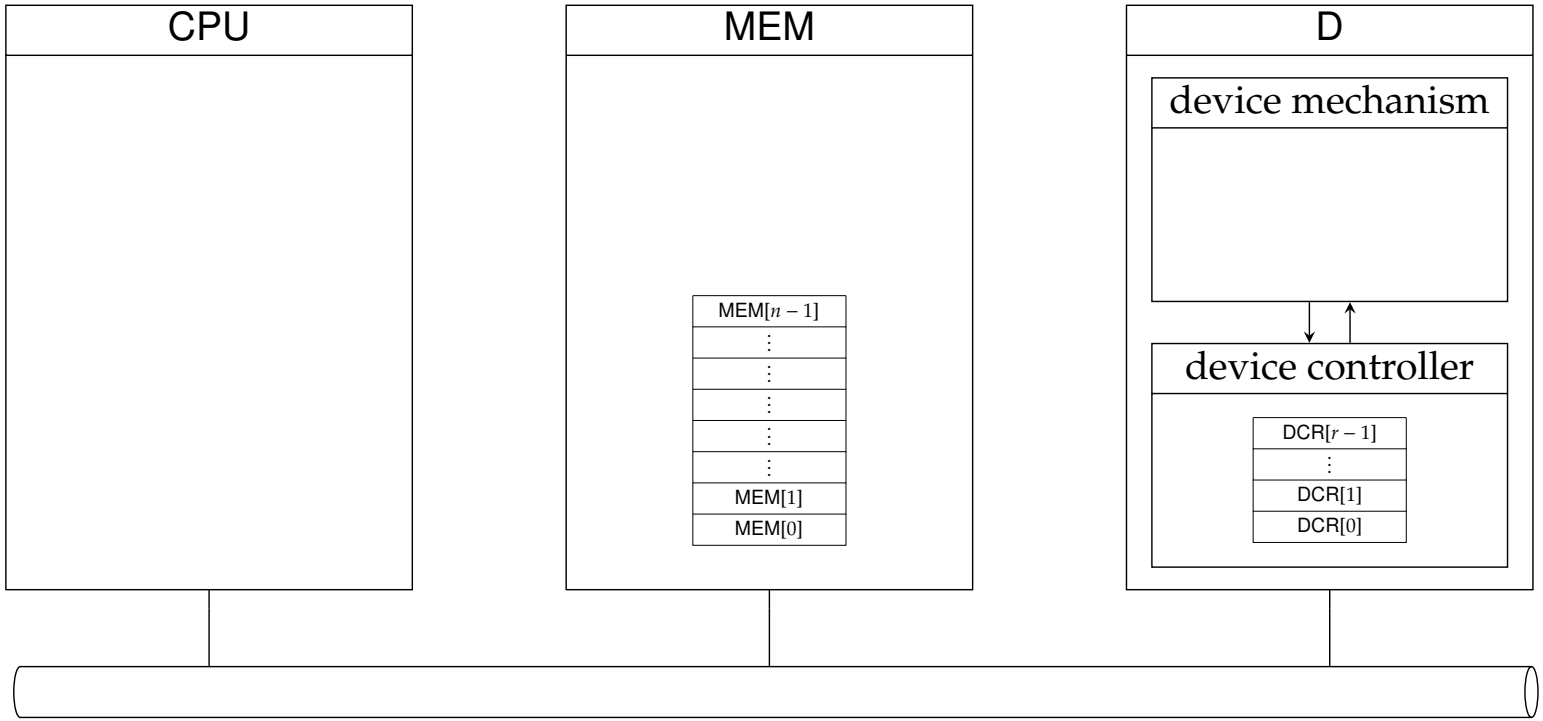
yielding a design space wrt. (at least)

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

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

Concept (3)

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



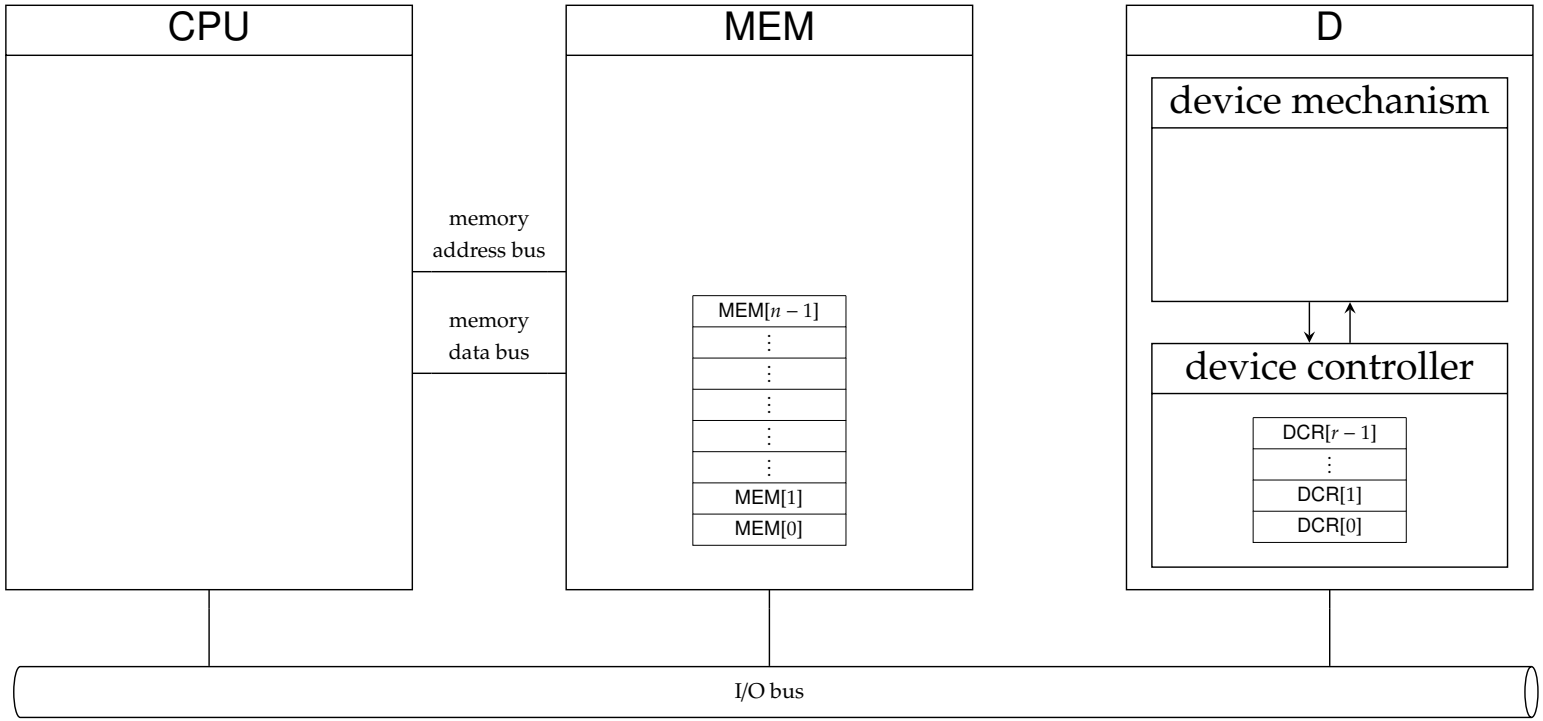
yielding a design space wrt. (at least)

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

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

Concept (3)

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



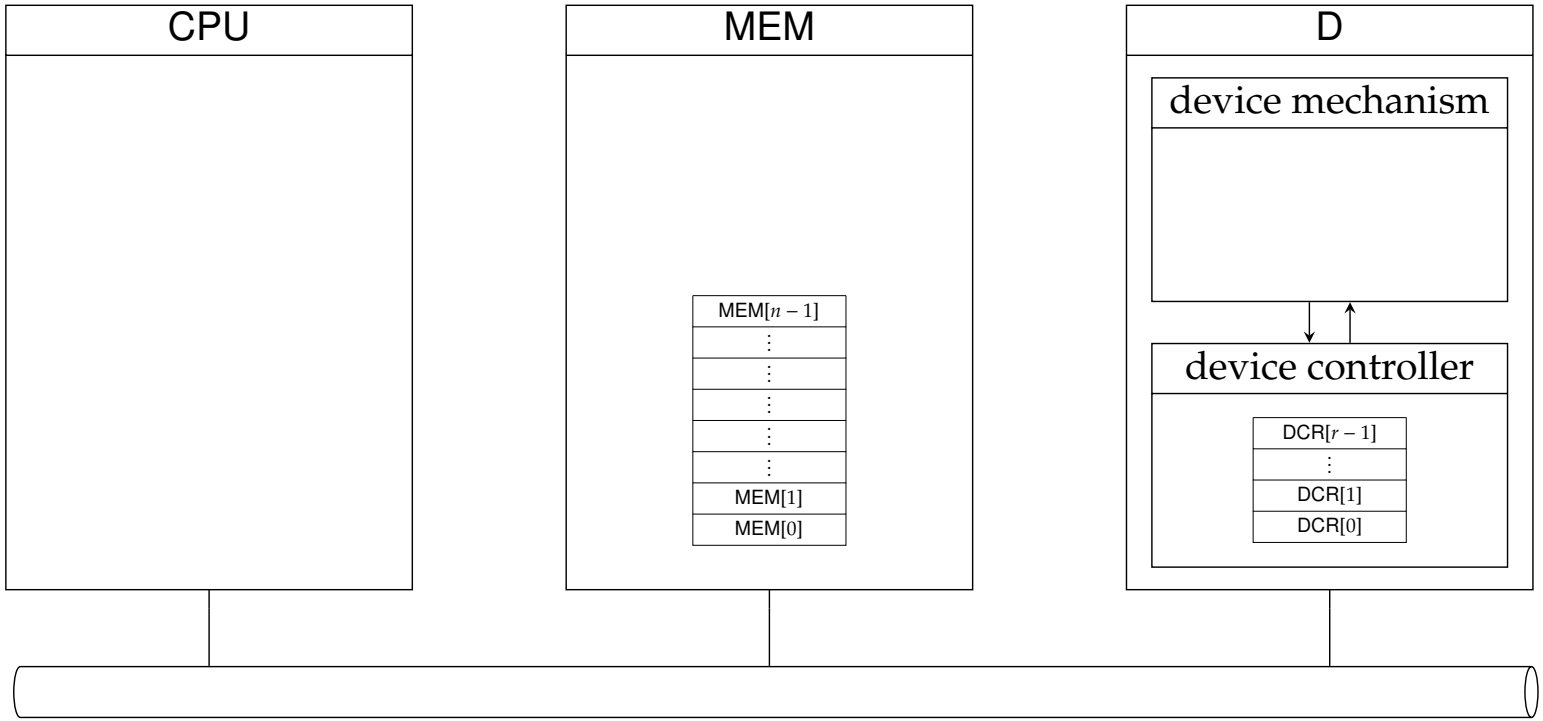
yielding a design space wrt. (at least)

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

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

Concept (3)

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



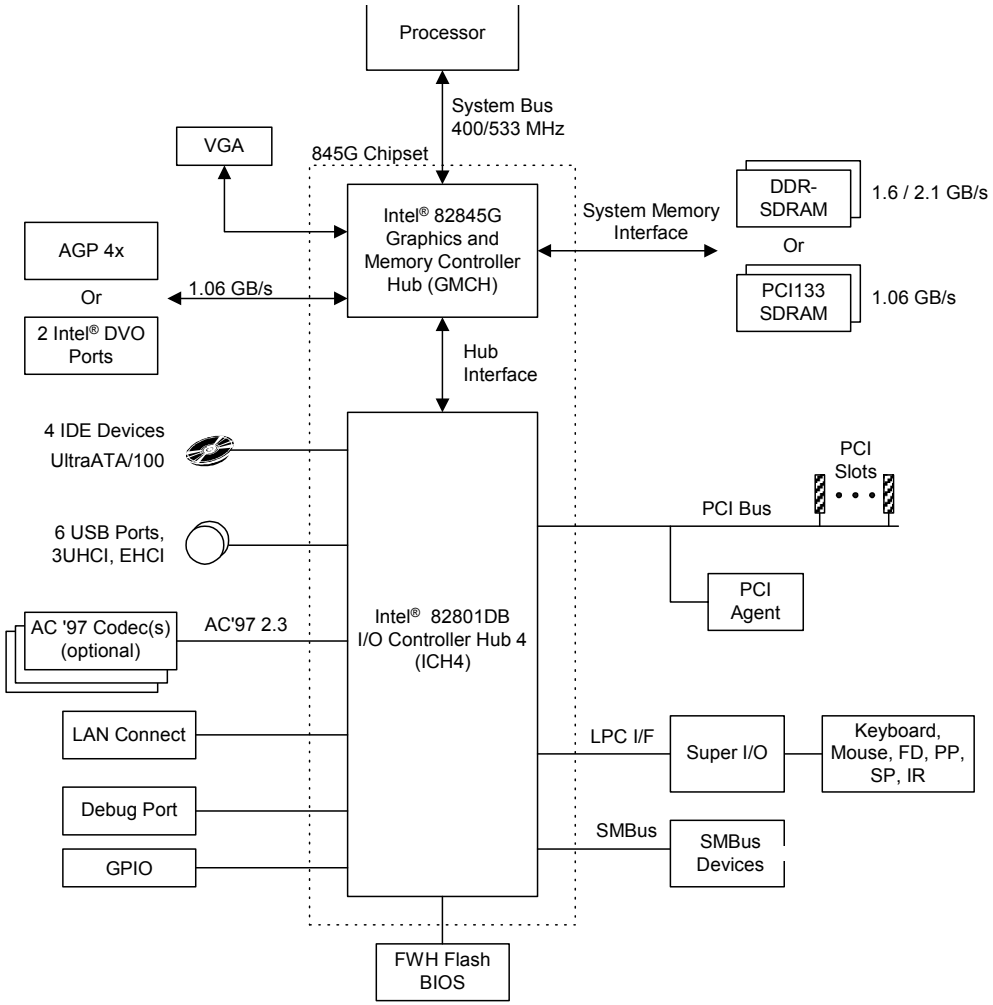
yielding a design space wrt. (at least)

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

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

An Aside: real bus (and chip set) architectures

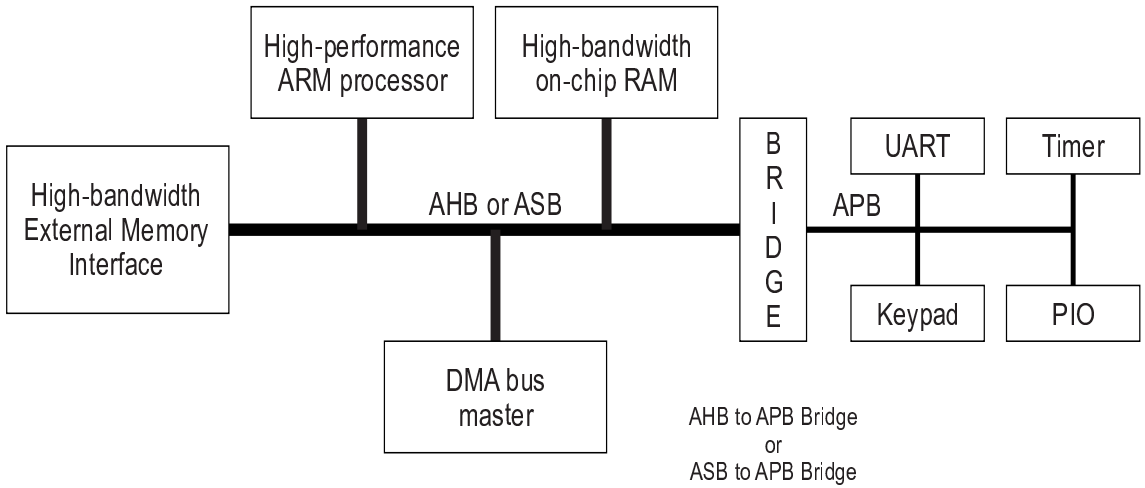
Example (Intel 845 "Brookdale", circa 2002)



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

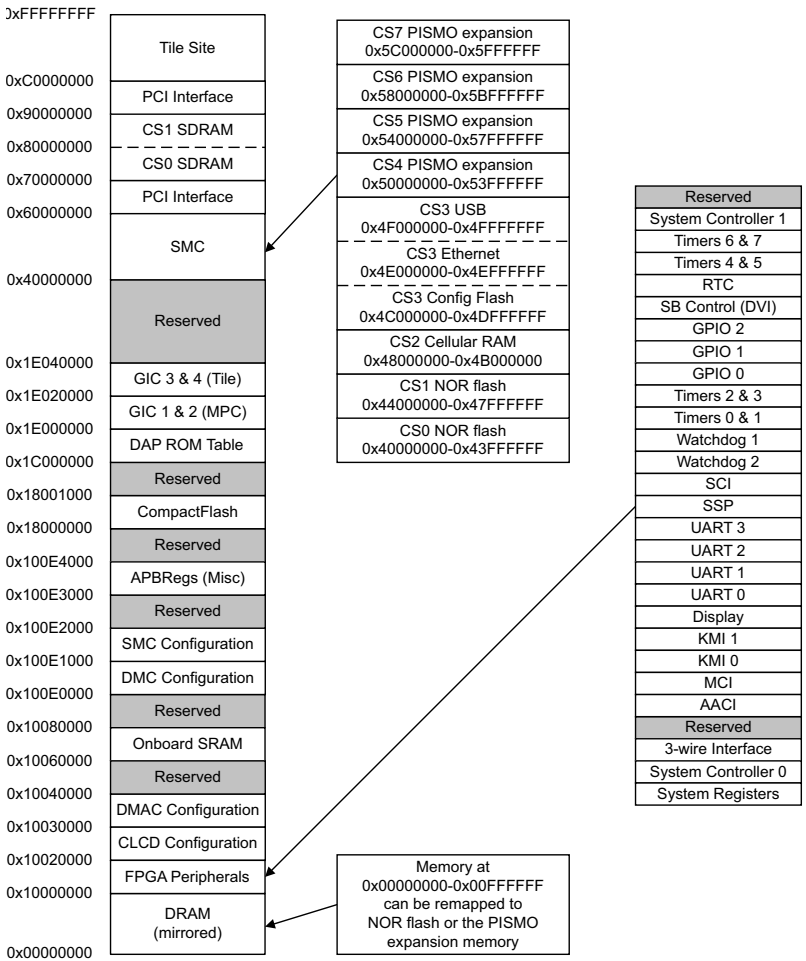
An Aside: real bus (and chip set) architectures

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



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

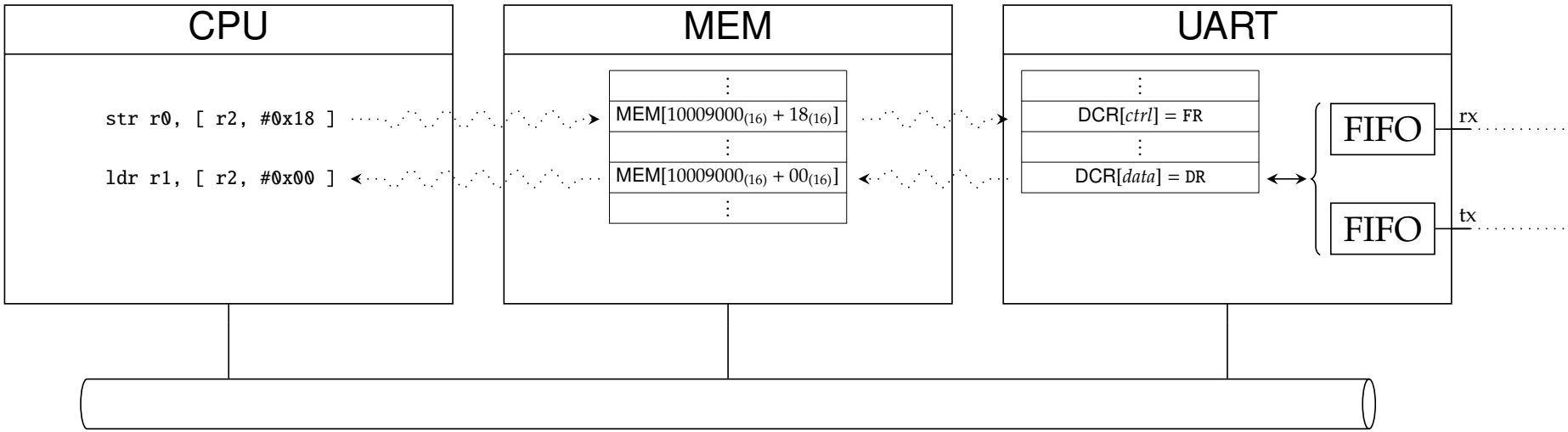
Example (Cortex-A8 Platform Baseboard memory map [13, Figure 4-1])



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

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

- 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

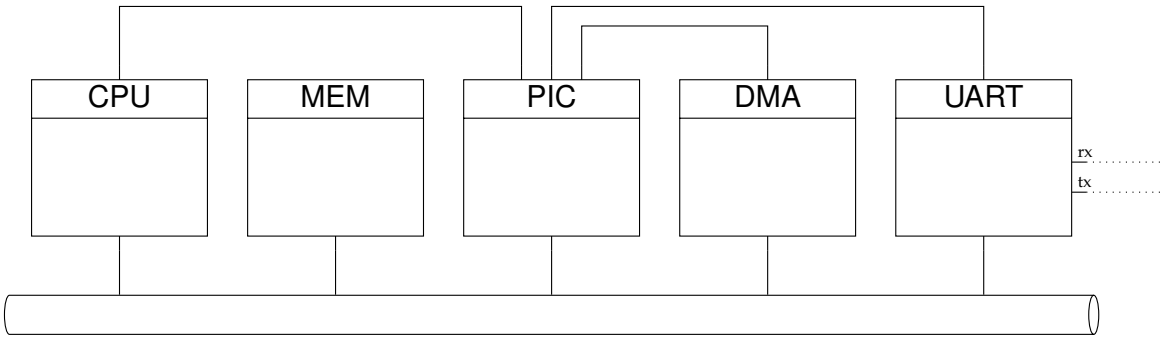
```
*( UART0 + 0x18 ) = x;
```

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).

Concept: I/O programming models (2)

- Example: consider



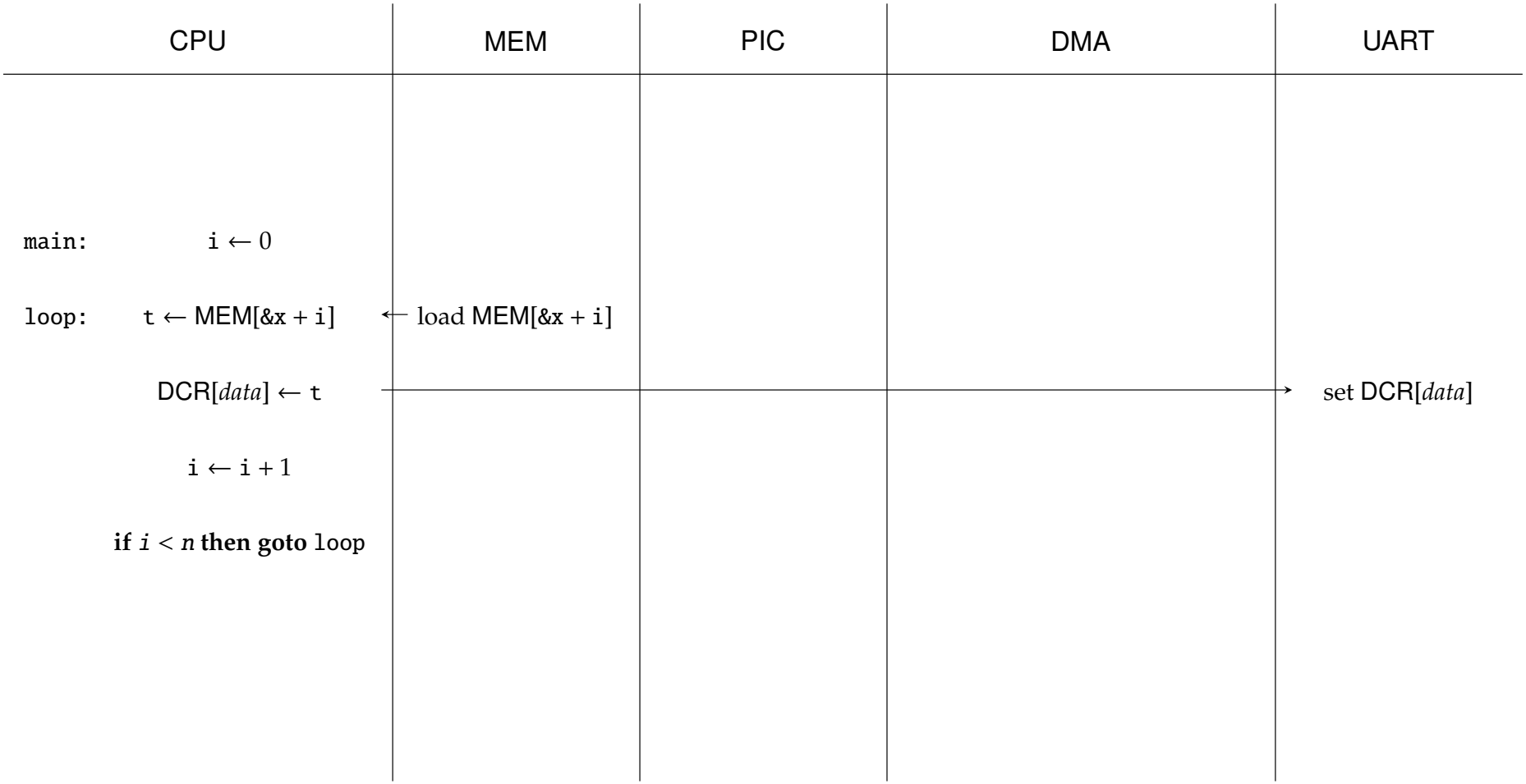
and a goal of transmitting some data in memory via the UART, i.e.,

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 }
```

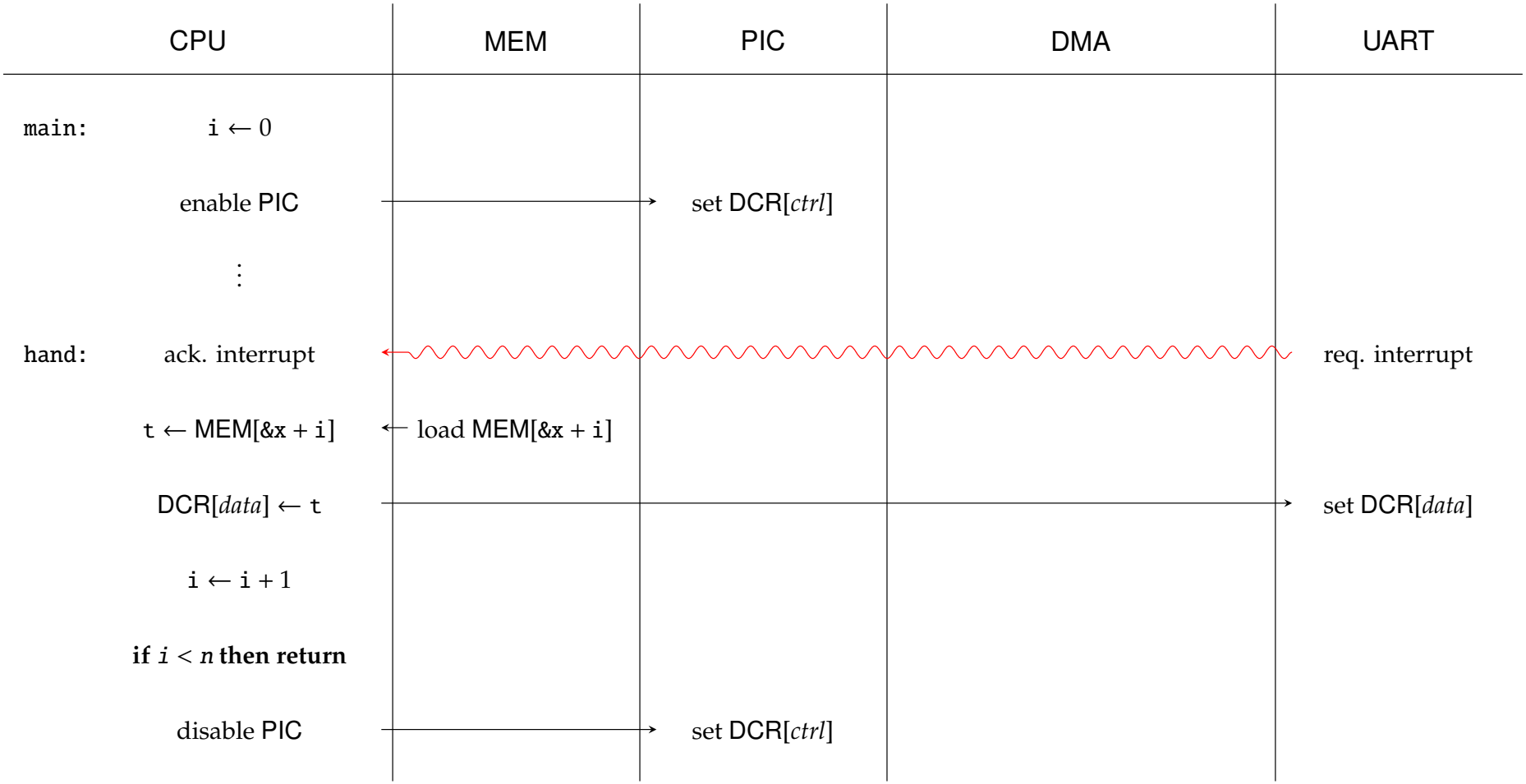
Concept: I/O programming models (3)

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



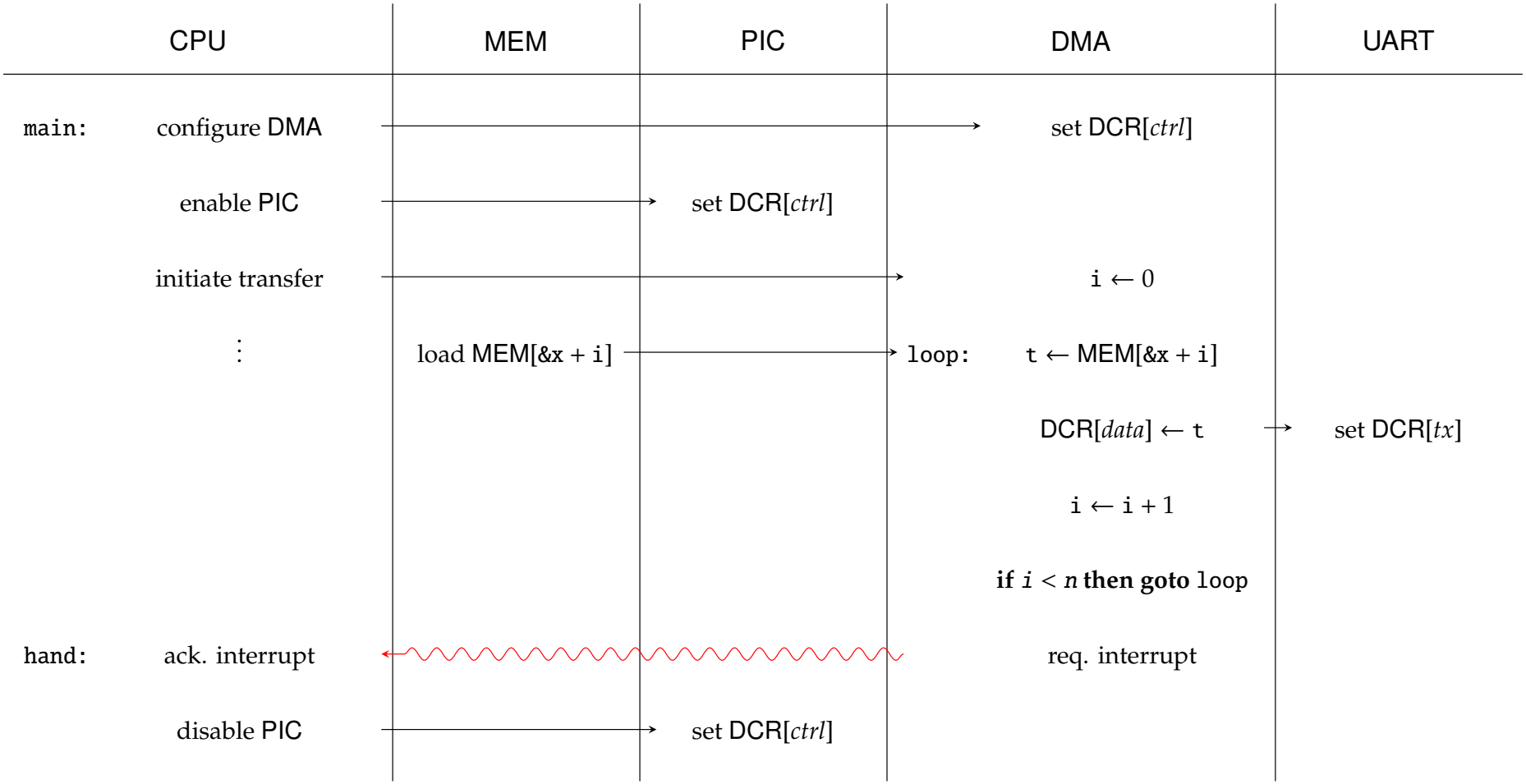
Concept: I/O programming models (3)

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



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.



Conclusions

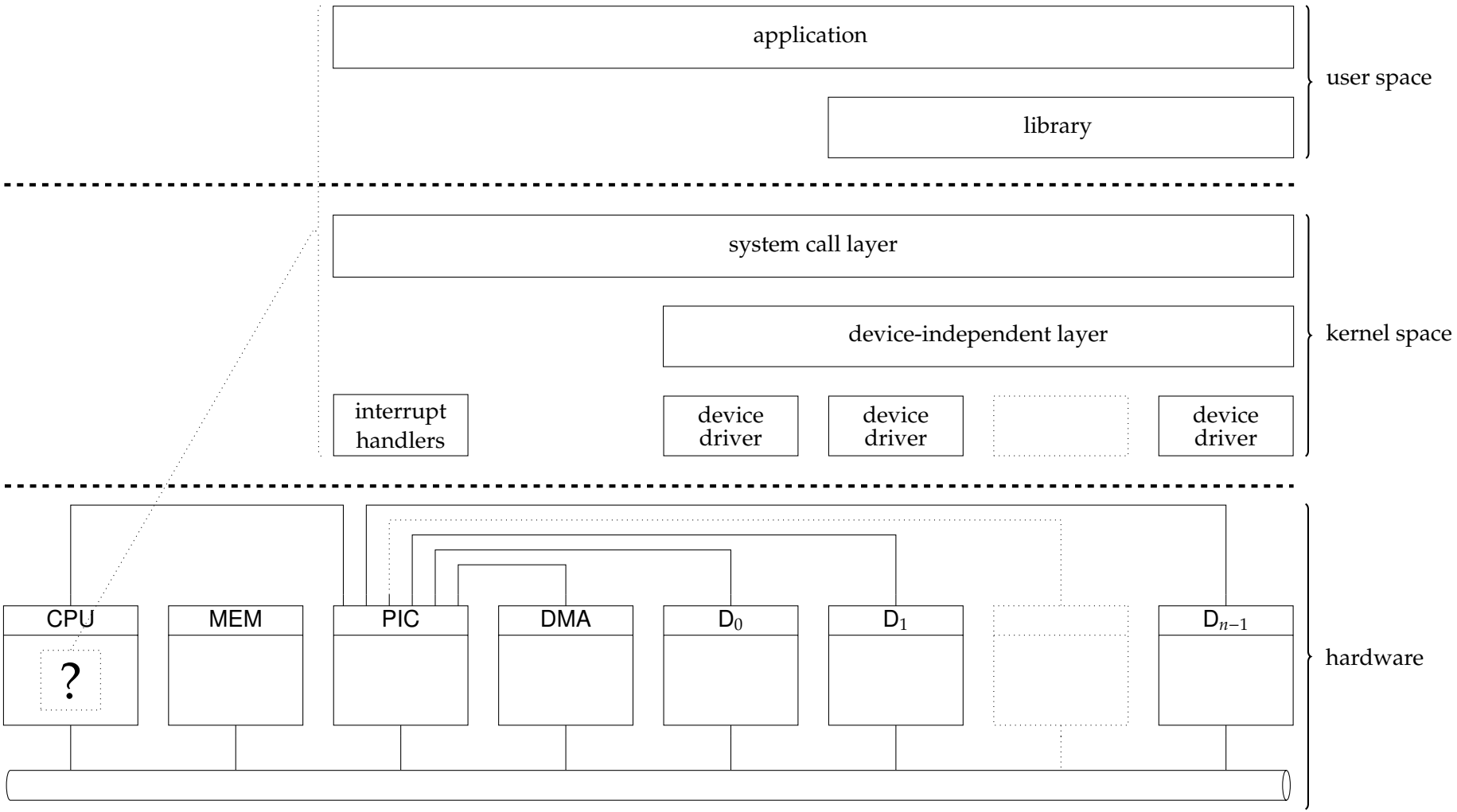
- Take away points:
  1. I/O is hard!
  2. The I/O sub-system must support
     - privilege management via invocation of **mode switches**,
     - the **system call interface**, allowing the kernel and user space software to interact, *and*
     - a suite of device-specific **device drivers**, allowing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

Conclusions



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. O'Reilly, 2005. URL: http://www.makelinux.net/ldd3/.
- J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 10: Interrupt handling". In: *Linux Device Drivers*. 3rd ed. O'Reilly, 2005. URL: http://www.makelinux.net/ldd3/.
- J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 15: Memory mapping and DMA". In: *Linux Device Drivers*. 3rd ed. O'Reilly, 2005. URL: http://www.makelinux.net/ldd3/.
- 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.

References

[1] *Wikipedia: Interrupt*. URL: http://en.wikipedia.org/wiki/Interrupt (see p. 44).

[2] J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 10: Interrupt handling". In: *Linux Device Drivers*. 3rd ed. O'Reilly, 2005. URL: http://www.makelinux.net/ldd3/ (see p. 44).

[3] J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 15: Memory mapping and DMA". In: *Linux Device Drivers*. 3rd ed. O'Reilly, 2005. URL: http://www.makelinux.net/ldd3/ (see p. 44).

[4] J. Corbet, G. Kroah-Hartman, and A. Rubini. "Chapter 9: Communicting with hardware". In: *Linux Device Drivers*. 3rd ed. O'Reilly, 2005. URL: http://www.makelinux.net/ldd3/ (see p. 44).

[5] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 13: I/O systems". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 44).

[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. 44).

[7] A.S. Tanenbaum and H. Bos. "Chapter 5: Input/output". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see p. 44).

[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. 6).

[9] W. Walker and H.G. Cragon. "Interrupt Processing in Concurrent Processors". In: *IEEE Computer* 28.6 (1995), pp. 36–46 (see pp. 2, 5, 6).

[10] *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. Tech. rep. DDI-0406C. ARM Ltd., 2014. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html (see pp. 9, 11–24).

[11] *Cortex-A8 Technical Reference Manual*. Tech. rep. DDI-0344K. ARM Ltd., 2010. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html (see p. 11).

[12] *PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual*. Tech. rep. DDI-0181E. ARM Ltd., 2004. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0181e/index.html (see pp. 7, 8).

[13] *RealView Platform Baseboard for Cortex-A8*. Tech. rep. HBI-0178. ARM Ltd., 2011. URL: http://infocenter.arm.com/help/topic/com.arm.doc.dui0417d/index.html (see p. 35).