



NTNU
Norwegian University of
Science and Technology

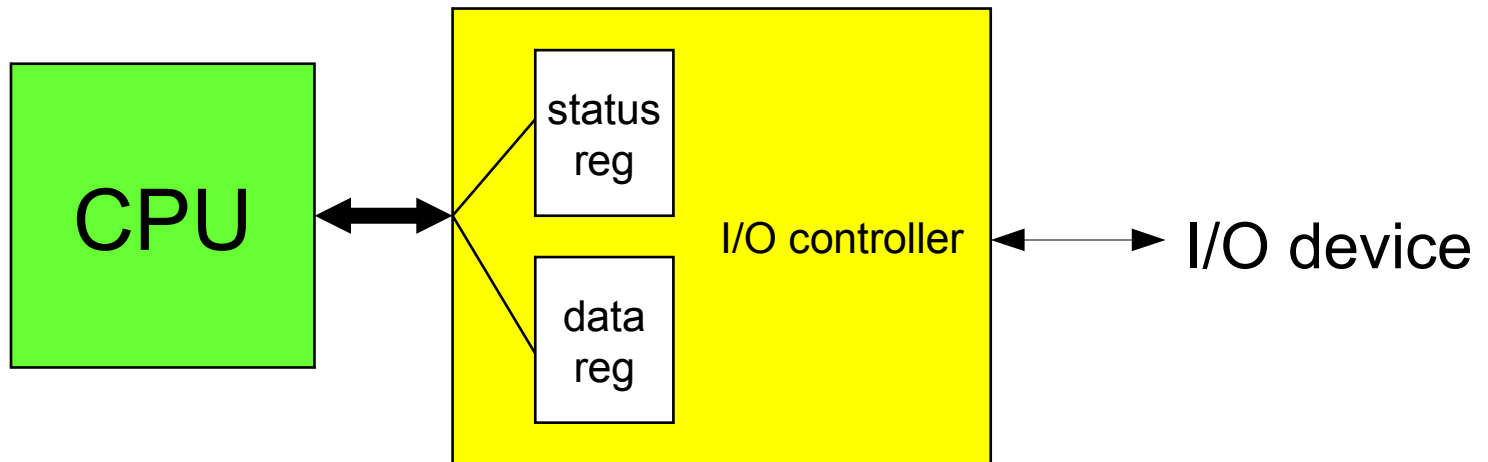
Lecture 4: I/O, exceptions, CPU performance

Asbjørn Djupdal
ARM Norway, IDI NTNU
2013

Lecture overview

- Accessing I/O devices
 - Memory mapped I/O
 - Polling
- Exceptions and processor modes
 - Interrupts
- Memory systems
 - Cache
 - Virtual memory
- CPU efficiency
 - Pipelines
 - Performance counters

I/O controllers and devices



Accessing I/O devices

- **Memory mapped**
 - General purpose load/store instructions
 - Most common I/O interface
- **Separate I/O address space**
 - Special purpose I/O instructions
 - Intel x86 provides in, out instructions
- A processor can have both I/O instructions and memory mapped I/O

ARM memory mapped I/O

```
DEVICE_BASE = 0x1000
```

```
REG1 = 0;
```

```
REG2 = 4;
```

```
LDR r1, #DEVICE_BASE
```

```
LDR r0, [r1, #REG1]           ; read from reg1
```

```
LDR r0, #8
```

```
STR r0, [r1, #REG2]          ; write to reg2
```

Memory mapped I/O, C example

```
#include <stdint.h>

#define DEVICE_BASE (void*)0x1000

#define REG1 0
#define REG2 4

uint8_t peek(volatile uint8_t *addr) {
    return *addr;
}

void poke(volatile uint8_t *addr, uint8_t val) {
    *addr = val;
}

void main() {
    uint8_t x = peek(DEVICE_BASE + REG1);
    poke(DEVICE_BASE + REG2, x);
}
```

- pointers
- Book example misses:
 - volatile
 - casting

Polling

```
#define OUT_CHAR (void*)0x1000
#define OUT_STATUS (void*)0x1004

#define READY 0

uint8_t peek(volatile uint8_t *addr) {
    return *addr;
}

void poke(volatile uint8_t *addr, uint8_t val) {
    *addr = val;
}

char *current = string;
while(*current != '\0') {
    while(peek(OUT_STATUS) != READY); // wait until ready
    poke(OUT_CHAR, *current);
    current++;
}
```

Polling

- Can be very wasteful
 - CPU spends lots of time waiting for device
 - Can not go to sleep while waiting, energy wasted
 - Hard to handle more than one I/O device (or other tasks) at the same time
 - But: Reacts fast
- Alternative method: [Interrupts](#)

Lecture overview

- Accessing I/O devices
 - Memory mapped I/O
 - Polling
- Exceptions and processor modes
 - Interrupts
- Memory systems
 - Cache
 - Virtual memory
- CPU efficiency
 - Pipelines
 - Performance counters

Exceptions

- An event that causes CPU to make a jump from its normal execution path
- Types:
 - Hardware exceptions (external)
 - Reset
 - Interrupts
 - Page faults
 - Software exceptions (internal) (traps)
 - Undefined instructions
 - Special instructions for creating exceptions
- Note: Terminology is not consistent

Exception behaviour

- Similar to subroutine call mechanism
- Exception forces CPU to:
 - save PC
 - update status register
 - jump to predetermined location (the exception handler)
- After the exception handler is done, the saved PC can be reloaded, and execution continues where it originally was

Exceptions

- Synchronous with instructions
- Pipelines and exceptions are tricky
 - Which instruction caused the exception?
 - How to restore state?
















Processor modes






- Useful to have different modes for different types of programs
 - Protective barriers between programs
- User mode
 - Restricted access
 - Can only see private memory, no access to I/O mem and privileged CPU registers.
- Supervisor mode:
 - Full access
 - Used when user mode programs need to do system calls

ARM processor modes

- Normal modes
 - User
 - All user code, not privileged
 - System
- Exception modes
 - Supervisor
 - System call
 - Abort
 - Page miss
 - Undefined
 - Undefined instruction
 - Interrupt
 - Normal interrupt
 - Fast interrupt
 - Interrupts for devices which needs immediate attention

ARM registers

Modes						
<div> <div>Privileged modes</div> <div>Exception modes</div> </div>						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	 R8_fiq
R9	R9	R9	R9	R9	R9	 R9_fiq
R10	R10	R10	R10	R10	R10	 R10_fiq
R11	R11	R11	R11	R11	R11	 R11_fiq
R12	R12	R12	R12	R12	R12	 R12_fiq
R13	R13	 R13_svc	 R13_abt	 R13_und	 R13_irq	 R13_fiq
R14	R14	 R14_svc	 R14_abt	 R14_und	 R14_irq	 R14_fiq
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		 SPSR_svc	 SPSR_abt	 SPSR_und	 SPSR_irq	 SPSR_fiq



indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

ARM exception handling

- Vector table at address 0

Vector_Table:

```
LDR pc, Reset_Addr
LDR pc, Undefined_Addr
LDR pc, SVC_Addr
LDR pc, Prefetch_Addr
LDR pc, Abort_Addr
NOP                               ;Reserved vector
LDR pc, IRQ_Addr
LDR pc, FIQ_Addr
```

```
Reset_Addr:      .word Reset_Handler
Undefined_Addr:  .word Undefined_Handler
SVC_Addr:        .word SVC_Handler
Prefetch_Addr:  .word Prefetch_Handler
Abort_Addr:      .word Abort_Handler
IRQ_Addr:        .word IRQ_Handler
FIQ_Addr:        .word FIQ_Handler
```

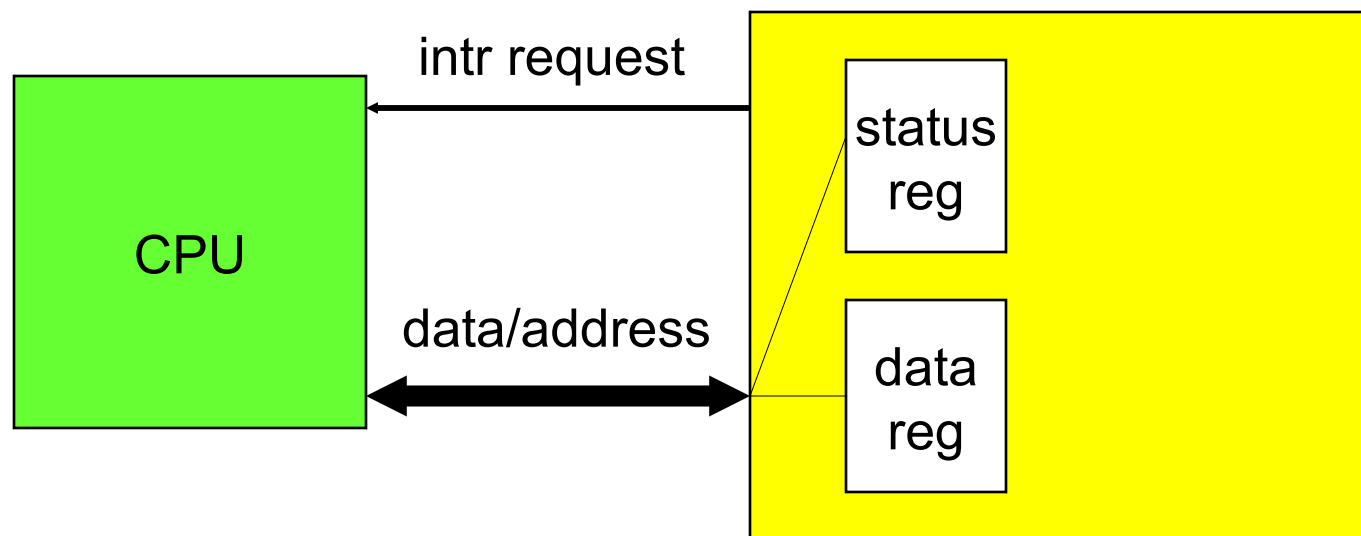

ARM exception handling

- When an exception occurs:
 - Switch register set
 - Save PC in R14
 - Save CPSR in SPSR
 - Jump to exception handler
- Return from exception handler
 - Move LR to PC and move SPSR to CPSR

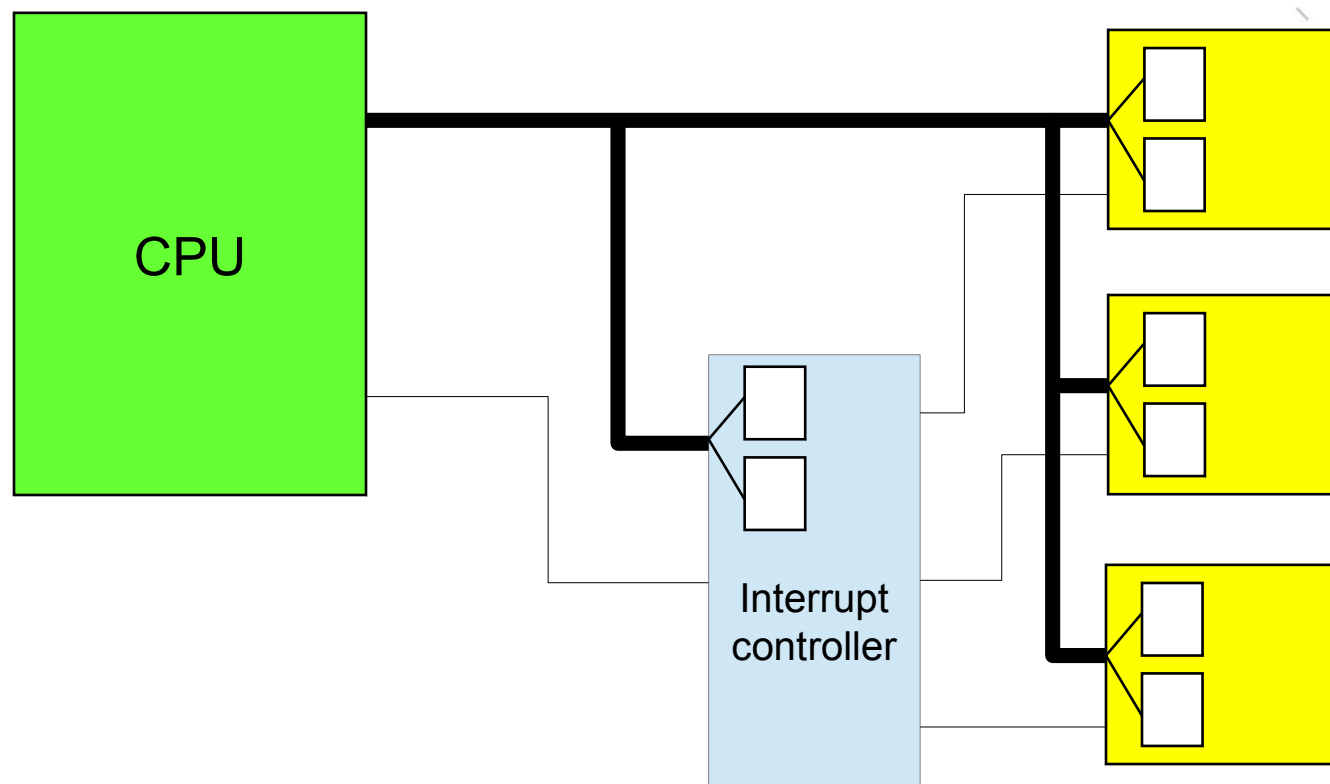
ARM system call from userspace

- User code only has access to local address space
 - Must enter privileged mode to access I/O
- SWI instruction
 - Exception handler is invoked in privileged mode
 - Performs syscall
 - returns to user space
- User code can never enter privileged mode, except through OS provided exception handler
 - OS has full control over all privileged access

Interrupt interface



Interrupt controllers



ARM interrupt handling

- Two different external interrupts
 - nIRQ input
 - nFIQ input
- nIRQ
 - Normal interrupt
 - Maskable
- nFIQ
 - Fast interrupt
 - Core can be configured with non-maskable FIQ

ARM Interrupt programming

- Must first setup interrupt vector at address 0
 - Assembly
- Write interrupt handler
 - Typically write entry and exit code in assembly, with a call to a C function in between
- The OS does all this for you.
 - Your job is to register your C function as an interrupt handler for your chosen system
 - Bare-metal programs must handle this themselves

Programming interrupt handlers

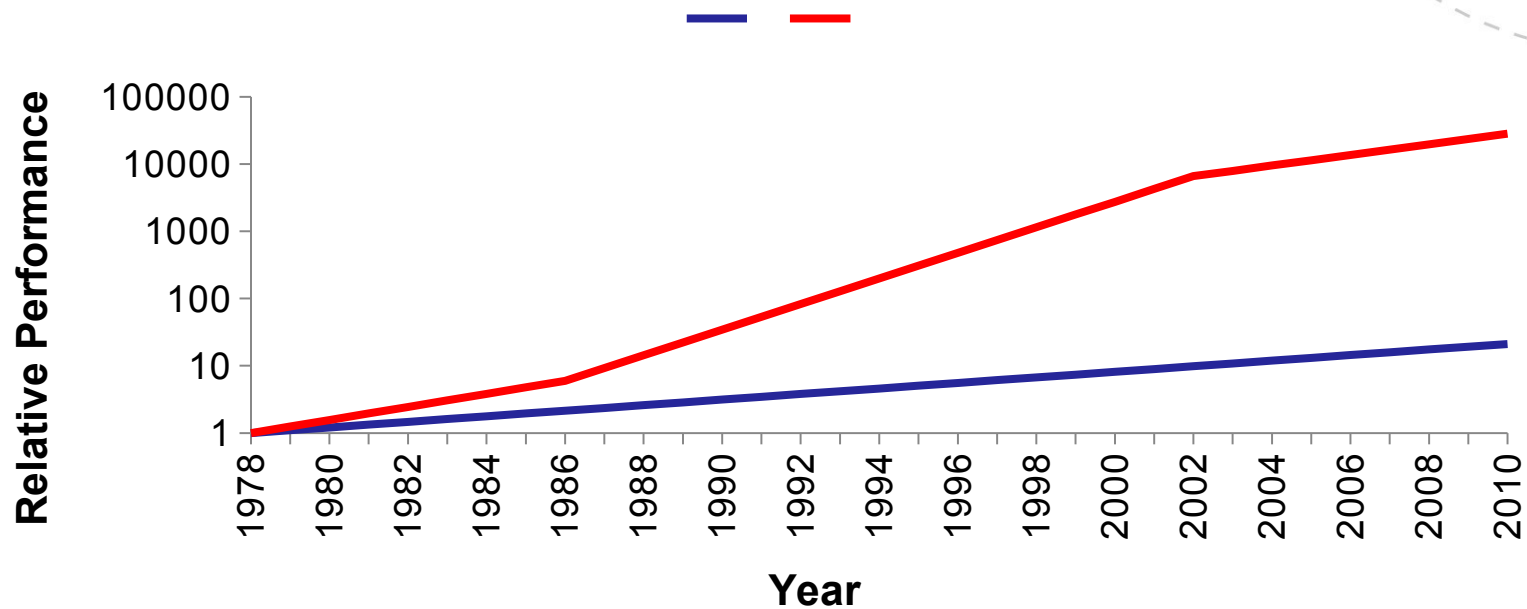
- Nested interrupts tricky
- Sharing data between interrupt handlers and other code can be tricky
 - Interrupt handlers can't do blocking calls (wait on semaphores), must return quickly
 - OS typically has mechanisms for dealing with this
 - Bare metal programs: Make your own
 - Protect code by temporarily disabling interrupts
 - Must have a clear idea of which operations are atomic
- Interrupts can be hard to debug
 - Can happen any time
 - Difficult bugs to reproduce

ARM interrupt C example

Lecture overview

- Accessing I/O devices
 - Memory mapped I/O
 - Polling
- Exceptions and processor modes
 - Interrupts
- Memory systems
 - Cache
 - Virtual memory
- CPU efficiency
 - Pipelines
 - Performance counters

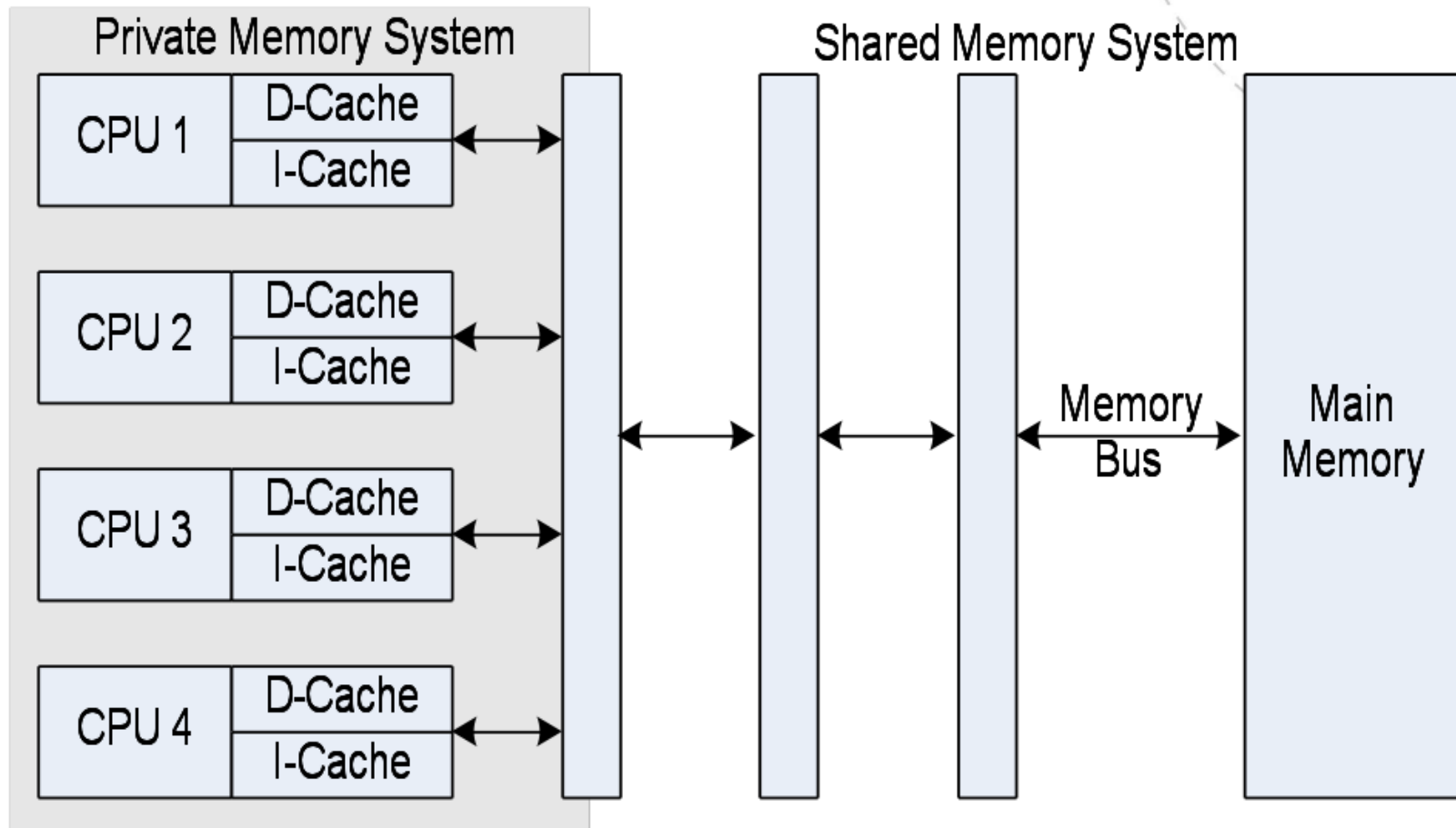
Why caching?



Observations

- Main memory is slow
- We can build faster memory if we make it smaller and closer to the CPU
- If most of our data fits in the smaller cache, we get faster execution
- If we automatically transfer data between main memory and cache, keeping the most used items in the cache, SW gets the illusion of both big and fast memory.

Multiple layers



Why caches work

- Principle of locality
 - Temporal locality
 - Spatial locality
- Motivation: Repeatedly processing data in an array
 - C arrays are in contiguous memory
 - Spatial locality: We access memory A, A+1, A+2,...
 - Temporal locality: We access the array often when we have started processing on it

Memory

09	
10	A
11	A+1
12	A+2
13	A+3
14	A+4
15	A+5
16	
17	x
18	y
19	

Cache operation

- May have caches for
 - Instructions
 - Data
 - Both
- Memory access time is no longer fixed
 - Fast if data is in cache
 - Slow if data must be fetched from main memory

Memory system performance

- h = cache hit rate
- t_{cache} = cache access time
- t_{main} = main memory access time
- Average memory access time (only one cache):

$$t_{\text{av}} = h * t_{\text{cache}} + (1-h) * t_{\text{main}}$$

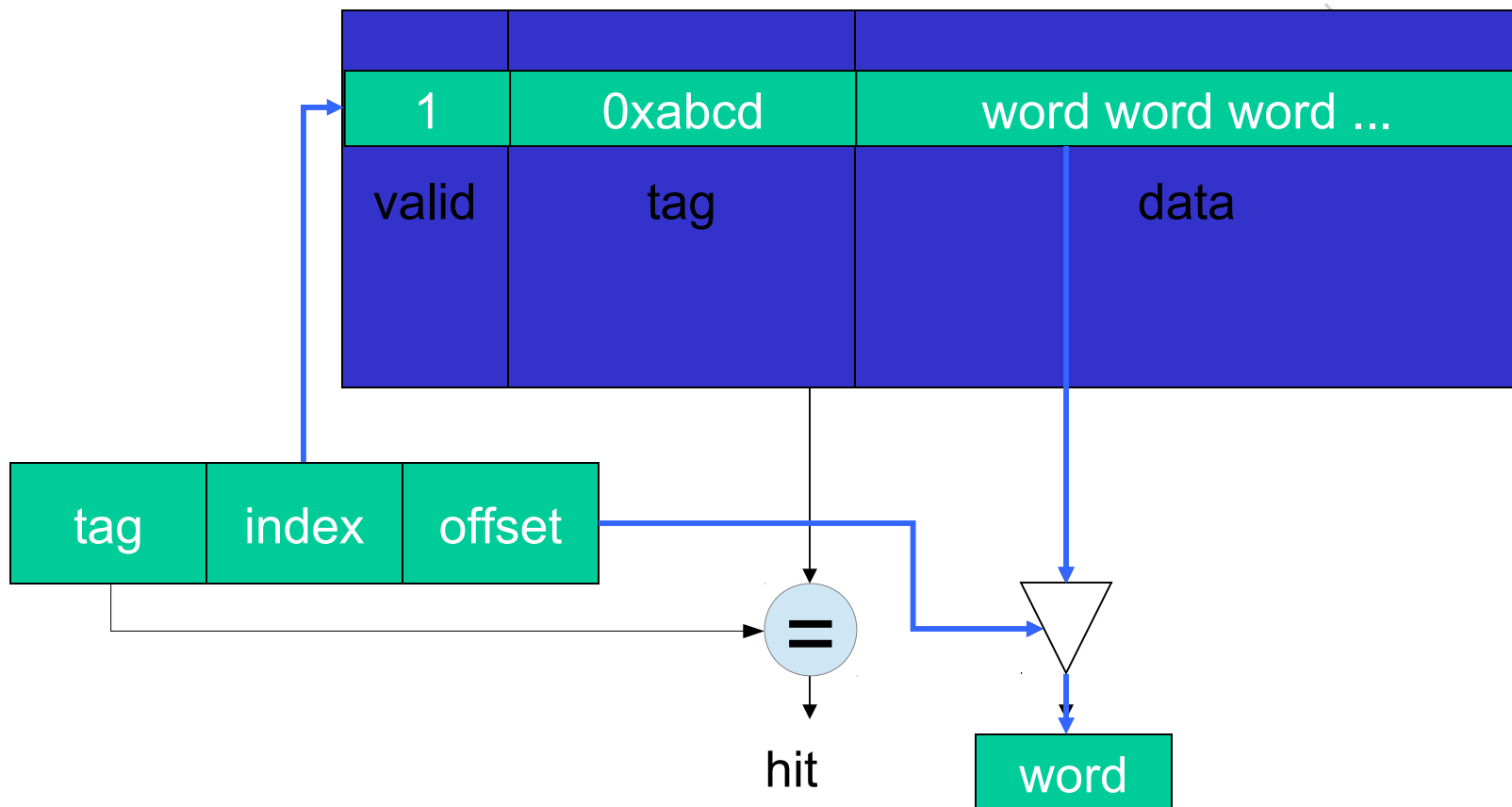
Terms

- **Cache hit**: Required location is in cache
- **Cache miss**: Required location is not in cache
- **Working set**: Set of locations used by program in a time interval
- **Cache block, cache line**: A data element stored in cache (often many words)

Cache organizations

- **Fully associative**: Any memory location can be stored anywhere in cache
- **Direct-mapped**: Each memory location maps onto exactly one cache entry
- **n-way set associative**: Each memory location can go into one of n sets

Direct mapped cache



n-way set associative

- Like n different direct mapped caches
- Any main memory location can be stored in any of the n sets
- Reduces conflicts misses
- Which cache entry to throw out to make room for new data? Replacement policies
 - Random
 - Least-recently used

Types of misses

- Compulsory (cold): Location has never been accessed
- Capacity: Working set is too large
 - These would not occur if cache was infinitely large
- Conflict: Multiple locations in working set map to same cache entry
 - Does not occur in fully associative cache

Write operations

- Write-through: Immediately copy to main memory
 - Simple
 - Keeps memory up to date
- Write-back: Write to main memory only when location is removed from cache
 - Reduces memory traffic
 - Memory not always up to date
 - Might require explicit cache flush for I/O buffers

Cache: Implications for SW

- Cache coherence
 - Several CPUs
- I/O memory
- Performance
 - Huge effect to make working set small enough for your cache

Virtual memory

- Provide CPU with virtual addresses, have a translation mechanism to map to physical addresses
- Why?
 - Each process can have its own virtual address space
 - Can provide larger virtual address space than physical memory
 - Swap to disk
 - Makes memory allocations more flexible
 - Programs requiring large contiguous memory blocks can get it even if physical memory is fragmented

Paging

- Divide memory into fixed size blocks, called pages
 - Each process has its own virtual address space
 - A virtual page address can be mapped to any physical page
- Memory Management Unit (MMU)
 - Translates from virtual to physical addresses
 - Transparent for SW (except for setup)

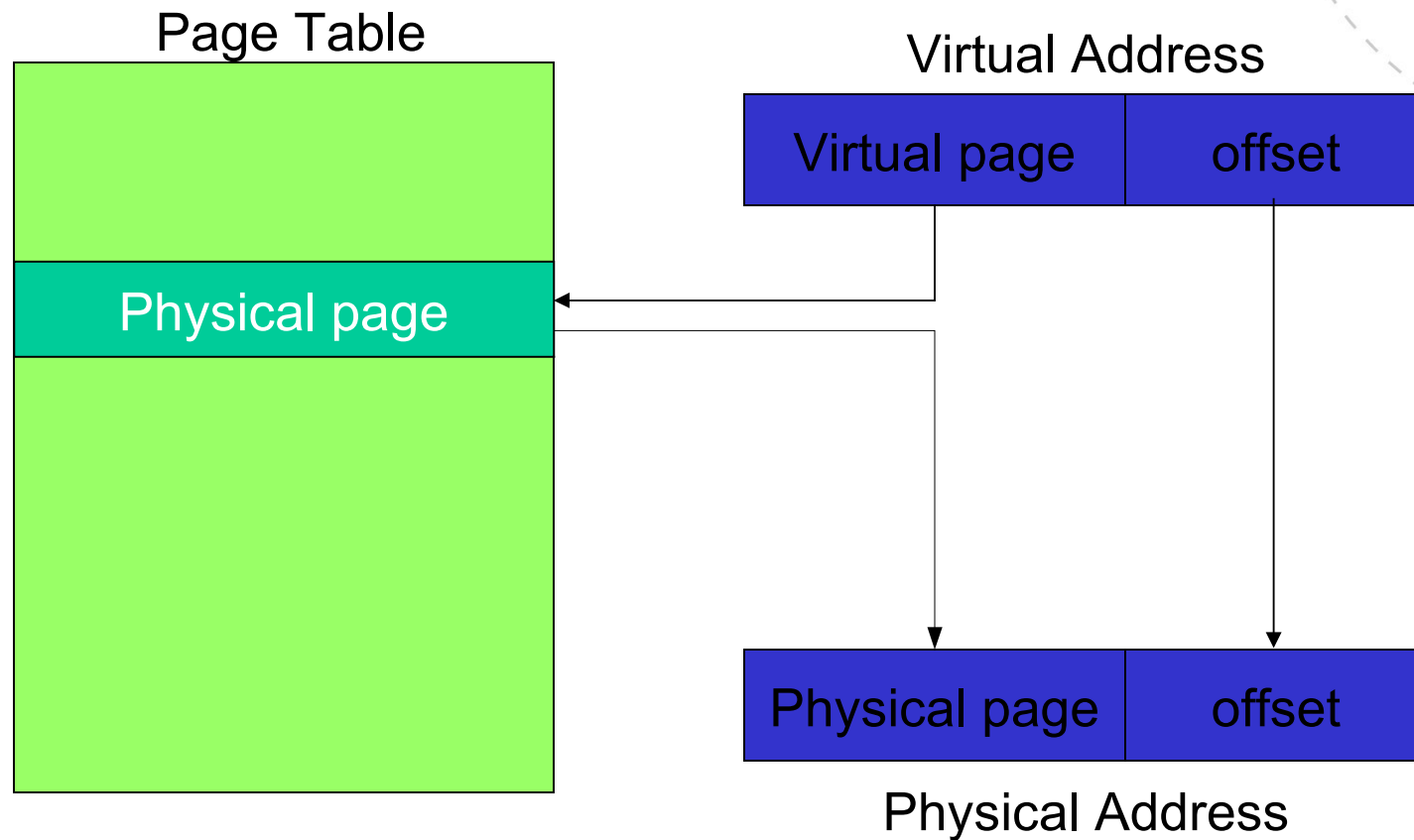
Memory

Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7
Page 8

Fragmentation

- Paging solves external fragmentation
- Internal fragmentation is still an issue

Page address translation



Translation lookaside buffer

- All memory addresses must be translated before every memory access
 - Too frequent to be possible in SW alone
 - Too frequent to go to page tables in RAM for every access
- Principle of locality is valid also here
 - Cache address translations: TLB
 - Associative cache

VM: Implications for SW

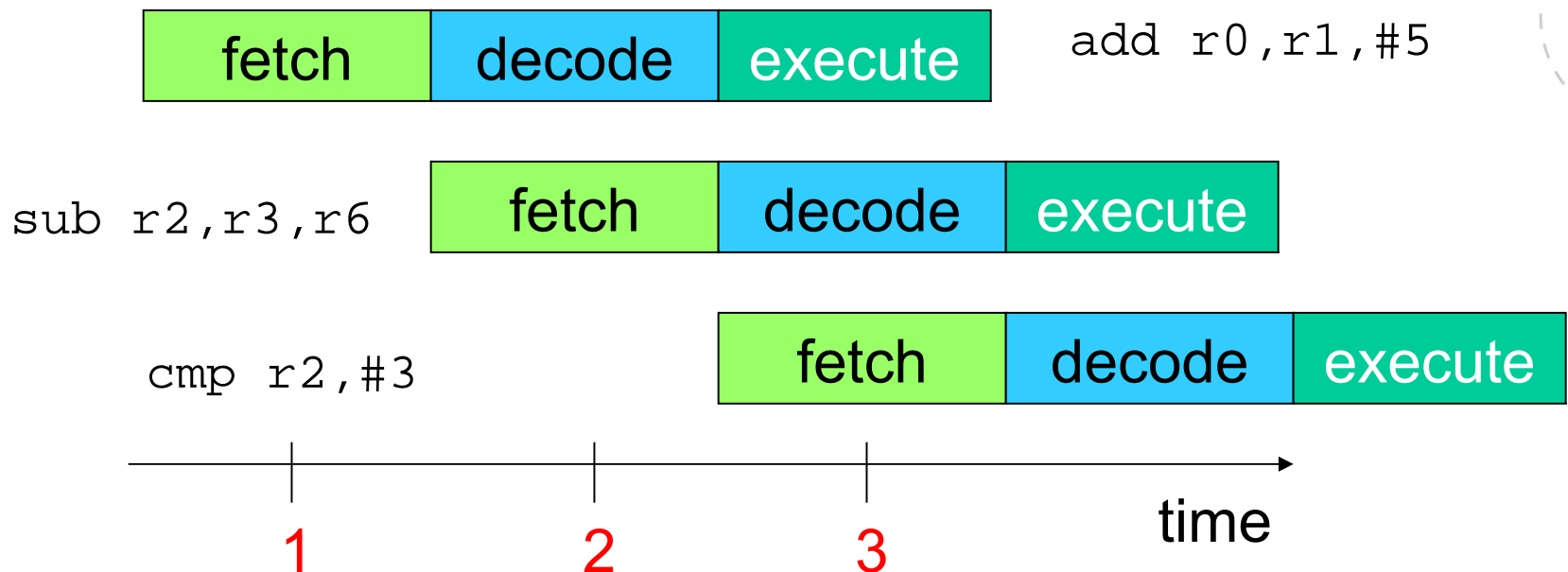
- For the application programmer:
 - Pretty much transparent
 - Performance
 - “Unlimited” memory
- For the driver programmer:
 - Accessing I/O memory means physical addresses can not be used
- For the OS programmer:
 - Page tables must be set up and maintained
 - MMU might be requirement for enabling caching
 - ARM: MMU holds information about which addresses can be cached

Lecture overview

- Programming I/O
 - Memory mapped I/O
 - Polling
- Exceptions and processor modes
 - Interrupts
- Memory systems
 - Cache
 - Virtual memory
- CPU efficiency
 - Pipelines
 - Performance counters

Pipelining

- Several instructions are executed at the same time, but at different phases of execution



Pipeline efficiency

- Performance measures
 - Latency: Time it takes an instruction to get through the pipeline
 - Throughput: Number of instructions executed per time period
- Pipelining increases throughput, but might increase latency
- Pipeline bubbles reduces throughput
 - Unused pipeline stage
 - Branches
 - Memory system delays
 - ...

Dependencies and hazards

- Dependencies

- Parallel instructions can be executed in parallel
- Instructions dependent on the result of other instructions are not parallel

```
ADD R1, R2, R3
```

```
SUB R4, R1, R5
```

- Property of instructions

- Hazards

- Situations where a dependency creates incorrect results if unhandled
- Property of the pipeline
- Not all dependencies give hazards

Dependency types

- Data dependencies
 - One instruction reads what a previous has written
- Name dependencies
 - Two instructions use the same register or memory location
 - But no flow of data between them
- Control dependencies
 - Instructions dependent on the result of a branch
- Remember: Independent of CPU architecture

Hazards

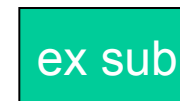
- Data hazards
 - Read After Write (RAW)
 - Write After Write (WAW)
 - Write After Read (WAR)
- Control hazards
 - Branches
- Structural hazards
 - A combination of instructions the pipeline can not handle

ARM data hazard

`ldmia r0,{r2,r3}`

`sub r2,r3,r6`

`cmp r2,#3`



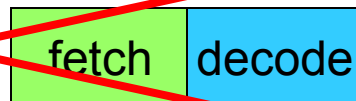
time

ARM control hazard

bne foo



sub r2,r3,r6



add r0,r1,r2



bne foo
sub r2, r3, r6

...

foo:
add r0, r1, r2

Avoiding control hazards

- A delayed branch executes n instructions after a branch, regardless if the branch was taken or not
 - Advantage
 - Requires no hazard detection and handling
 - Can be more efficient because no bubbles are inserted in pipeline
 - Disadvantage
 - Required that independent instructions are available, or else NOP must be inserted
 - Exposes pipeline design to the programmer (compiler), resulting in bugs or decreased performance
- Branch prediction unit
 - Guess if a branch is taken or not based on history
 - Penalty only if guess is wrong

Hazards: Implications for SW

- When handled by a dedicated HW hazard unit:
 - Performance
- When not handled by HW:
 - SW must take care of all hazards
 - Reorder instructions or insert NOP instructions
- ARM hazard handling:
 - All in HW

Performance counters

- Modern CPUs have internal counters for various performance related events
 - Cycles
 - Instructions
 - Cache hits
 - Cache misses
 - ...
- PAPI: Performance API, system independent API for accessing performance counters. Requires OS and HW support
- Linux: `perf_events`

Performance counter example

Next lecture

- Next week: Work on exercise 1
 - No lecture
- Next lecture: Thursday week 8
 - C programming
 - Bus based systems