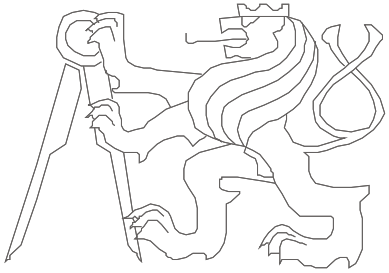
Computer Architectures

https://cw.fel.cvut.cz/wiki/courses/b35apo/start

External Events Processing and Protection

Pavel Píša, Petr Štěpán, Richard Šusta,
Michal Štepanovský, Miroslav Šnorek



Czech Technical University in Prague, Faculty of Electrical Engineering

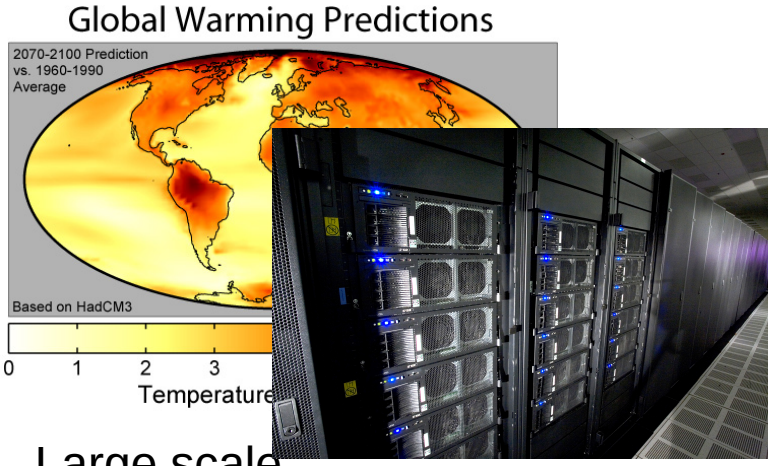
The Computer Basic Building Blocks (repeating)

- Central Processing Unit (CPU)
- Memory – for data and code ordered into hierarchy
  - registers (fast CPU local memory)
  - cache (L1, L2, etc)
  - main memory (RAM – DDR),
  - external memory (disk)
- Interconnection – buses, networking
  - ISA, PCI, PCIexpress

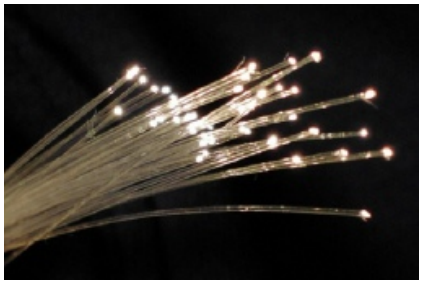
What Is Purpose to Have These Building Blocks

Entertainment, games, video

Enterprise applications, accountancy, bank systems, inventory, online shops



Large scale mathematical and modeling computation (global climatic forecast and analysis, nuclear fusion, etc.)

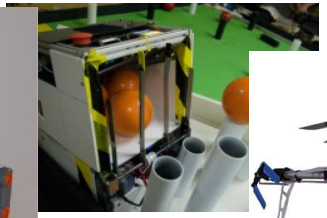


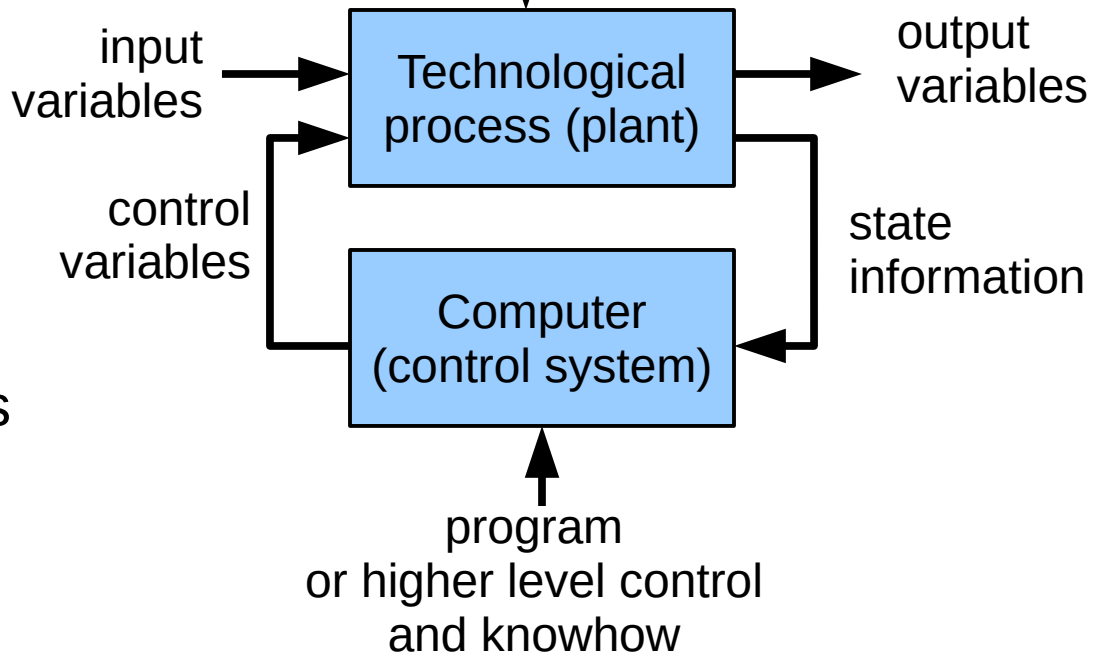
Communications, as a main target (phone, mobile) or as a way to achieve data exchange for other tasks and applications

And many others areas of use ...

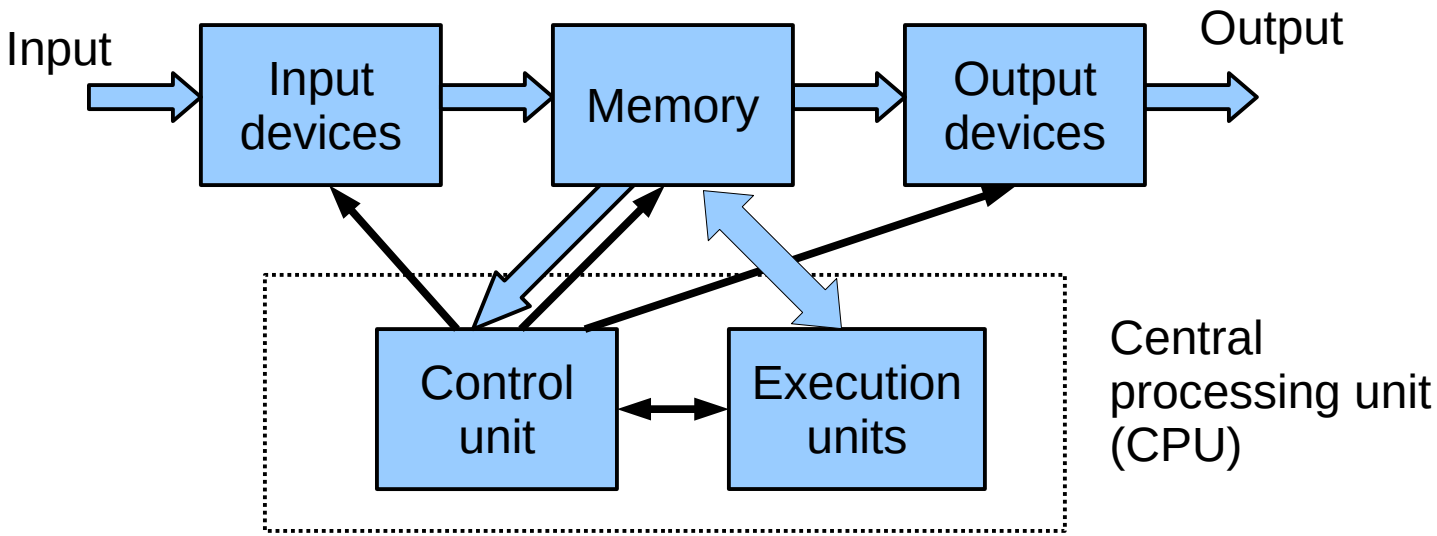
Computer as Controller in Field Applications

1. complex process (fast computation.)
2. cheap serially produced units
3. very flexible (programmable)
4. hierarchic control available
5. precise evaluation (display)
6. complex algorithms (only memory and time constraints)





Data Flow in the Computer System



Different demands properties of data processing

- Batch processing (a task controls data access as it is processing these data)
- Interactive (events driven – by user or when external requests or event arrives)
- Real-time control – computation results delivered late are of no or inferior value

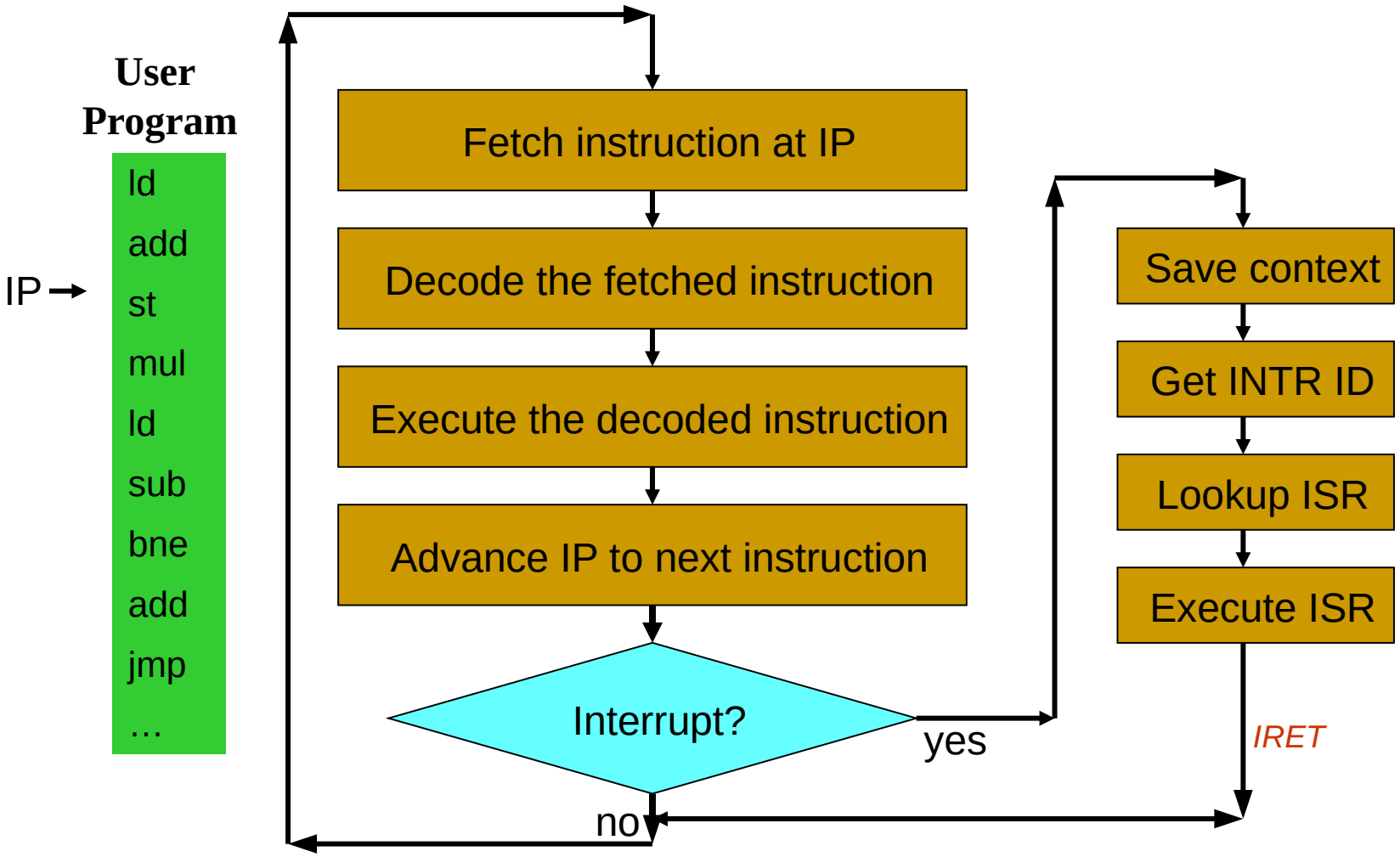
Input-output (I/O) Subsystem (repeated)

- Input only peripherals
  - Common ones: keyboard, mouse, video camera
  - Logic inputs, physical quantities – usually converted to analog electrical signal and then by A/D converter to numerical value accessible on input port and other sensors
- Output only peripherals
  - Video output (2D, 3D + acceleration), audio output
  - Outputs with physical effect, 3D printer (rapid prototyping), technological process control (D/A converters, PWM) and many other kinds of actuators
- Bidirectional
  - Hard disk, communication interfaces
  - Most of above listed “unidirectional” peripherals requires read and write access for their setup, monitoring and parameters control

Methods to Transfer Data between Peripherals and CPU

- Programmed input/output (PIO) with polling
  - CPU loops in cycle and waits for status information signaling available input data or space in output buffer
- Interrupt driven programmed input/output (PIO)
  - Program/operating system configures peripheral but does not wait for data. Data arrival is signaled by interrupt (asynchronous event/exception). The data are read in interrupt service routine.
  - Output is initiated by CPU write of data to a register if space is available. Ready for next data it signaled by interrupt.
- Direct memory access – DMA
  - CPU setups source and destination, transfer is realized by specialized unit.
- Intelligent peripherals/controllers, bus master DMA

Interrupt/Exception as Part of CPU Cycle



Exceptions and Interrupts

- Exceptions – anomalous or exceptional situations (blocking further regular execution) requiring special processing
  - Main recognized sources of exceptions
    - Undefined instruction (RISC V - unknown opcode)
    - Arithmetic exception (divide by zero, overflow - not on RISC V)
    - System call (syscall instruction)
  - Data unavailable or write fault
    - Bad address or page marked as invalid
    - Bus error detected (parity, ECC, acknowledge limit exceed)
- Asynchronous/external exceptions (interrupts)
  - Maskable, can be disabled in state/control world of CPU, possibly based on source priority (peripherals, timers, counters)
  - Non-maskable – HW faults, supervision circuits (Watch Dog)

Steps of Exception or Interrupt Processing

- Exception is accepted/processed usually unconditionally, external interrupt only if not masked or if non-maskable
- CPU state vector is saved including PC (on system stack or to the special registers)
- Program Counter is preset to the starting address of handler according to exception type or even interrupt source number
- Servicing routine starting at that address is executed
- It stores state of other registers on stack, communicates with peripheral, loads missing page, informs about nonrecoverable task fault or whole system, etc.
- If recoverable – restores registers values to state before entry
- Routine is finalized by special exception return instruction which switches CPU into previous state and allows continuation of interrupted code

Exceptions Sources on RISC-V

- Exceptions caused by hardware malfunctioning:
  - **Machine Check**: Processor detects internal inconsistency;
  - **Bus Error**: on a load or store instruction, or instruction fetch;
- Exceptions caused by some external causes (to the processor):
  - **Reset**: A signal asserted on the appropriate pin;
  - **NMI**: A rising edge of NMI signal asserted on an appropriate pin;
  - **Hardware Interrupts**: Six hardware interrupt requests can be made via asserting respective signal.

    Hardware interrupts can be masked by setting appropriate bits in Status register;

Exceptions Sources on RISC-V - continued

- Exceptions that occur as result of instruction problems:
  - **Address Error**: a reference to a nonexistent memory segment, or a reference to Kernel address space from User Mode;
  - **Reserved Instruction**: A undefined opcode field (or privileged instruction in User mode) is executed;

- Exceptions caused by executions of special instructions:
  - **Syscall**: A Syscall instruction executed;
  - **Break**: A Break instruction executed;

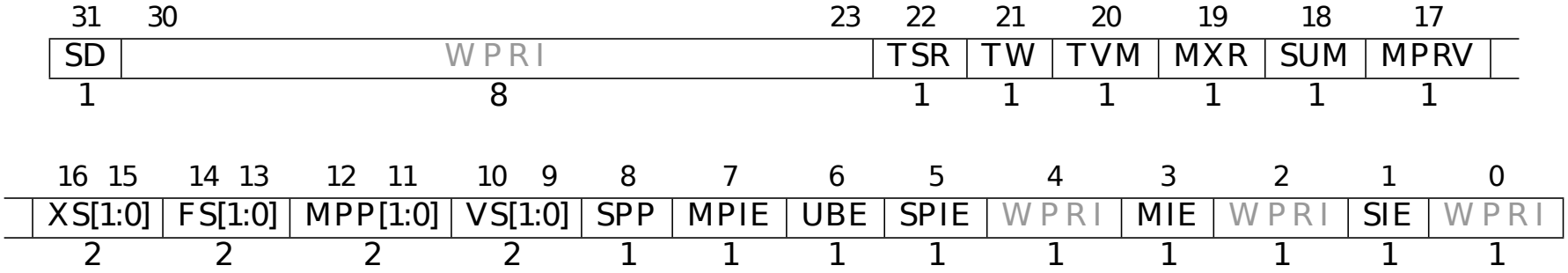
RISC-V – Exceptions Status and Control Registers

Register name	Register number	Usage
mstatus	0x300	Machine status register.
misa	0x301	ISA and extensions
mie	0x304	Machine interrupt-enable register.
mtvec	0x305	Machine trap-handler base address.
mscratch	0x340	Scratch register for machine trap handlers.
mepc	0x341	Machine exception program counter.
mcause	0x342	Machine trap cause.
mtval	0x343	Machine bad address or instruction.
mip	0x344	Machine interrupt pending.
mtinst	0x34A	Machine trap instruction (transformed).

The RISC-V Instruction Set Manual – Volume II: Privileged Architecture
https://riscv.org/technical/specifications/

RISC-V – Machine Status Register (RV32)

Machine Status Register (mstatus)



Field	Bit(s)	Usage
SIE	1	Supervisor global interrupt enable
MIE	3	Machine global interrupt enable (for handler atomicity)
SPIE	5	SIE before trapping to system mode
MPIE	7	MIE before trapping to machine mode
SPP	8	Priority mode before trapping to system mode
VS	10:9	Inform if floating point save is needed
MPP	12:11	Priority before trapping into machine mode

RISC-V – Machine Cause Register (RV32)

Machine Cause Register (mcause)

The register informs handler what caused the trap into machine mode.
If the MSB bit (RV32 bit 31, RV64 bit 63) is set then the source is asynchronous exception/peripheral/external interrupts. The exception code corresponds to source and corresponding position of enable and source pending bit in the **mie** and **mip** registers. **mepc** points to the interrupted (the first unprocessed) instruction. Simple return by **mret** instruction is possible to continue in the background program execution.

When MSB is clear, synchronous exception source caused the trap. **mepc** points to the causing instruction. When blocking cause (i.e. page fault) is resolved instruction can be restarted by simple **mret**. If the reason is syscall (**ecall** instruction) or invalid instruction which is emulated by handler, then **mepc** has to be advanced after instruction before **mret**.

RISC-V – Exception Sources Encoding

IRQ bit	Number	Cause of exception
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint exception
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store/AMO page fault
0	24 – 31	*Designated for custom use*

RISC-V – Exception Sources Encoding for Interrupts

IRQ bit	Number	Cause of exception
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
1	≥16	*Designated for platform use*

- **mie** register enables individual interrupt sources, bit # matches source
- **mip** register informs about actually pending sources
- **mstatus.MIE** global enable (1) / disable (0)
- There is another set of the registers for supervisor level of the control
- **sstatus**, **sie**, **sip, sscratch**, **scause** etc. Their structure is the same but they are accessible from the supervisor (system) mode and machine level control is not allowed

RISC-V – Exception/Interrupt Processing

CPU accepts interrupt request, exception or (mach/sys/user) **ecall** opcode

```
mepc <= pc and switches to machine privilege mode
mstatus.MPP and mstatus.MPV set to preceding privilege mode
mcause <= exception code, mcause[XLEN-1] <= 1 if interrupt
mstatus.MPIE <= mstatus.MIE, mstatus.MIE <= 0
PC <= mtvec (for vectored mode and interrupts BASE+4×cause)
```

Interrupt service routine/exception handler startup is responsible for

- identification of request cause **csrr rd, mcause**
- CPU state can be controlled by CSR instructions
- **csrrw *rd*, *csr*, *rs1***, **csrrwi *rd*, *csr*, *uimm5***
- **csrr(s/c)(i) *rd*, *csr*, *rs1*** or ***uimm5***)
- **csrr *rd*, *csr*** pseudo for **csrrs *rd*, *csr*, x0**
- **csrw *csr*, *rs*** pseudo for **csrrw x0, *csr*, *rs1***

The **mret** instruction finalizes exception handling and enables exceptions and interrupts for machine mode

```
priviledge mode from mstatus.MPP and mstatus.MPV
mstatus.MPV <= 0, mstatus.MPP <= 0
mstatus.MIE <= mstatus.MPIE
pc <= mepc and continue execution in mode restored from mstatus.MPP
```

Precise Exception Processing

- If interrupt/exception is successfully handled (i.e. missing page has been swapped in, etc.) and execution continues at instruction before which interrupt has been accepted, then interrupted code flow is not altered and cannot detect interruption (except for delay/timing and cases when state modification is intended/caused by system call)
- Remark: Precise exception handling is most complicated by delayed writes (and superscalar CPU instruction reordering) which leads to synchronous exceptions detected even many instruction later than causing instruction finishes execution phase. Concept of state rewind or “transactions” confirmation is required for memory paging in such systems.
- Commit stage is the last stage in the pipeline in which the exception can arise, the instruction will not generate after it

Evaluation of the Exception Sources on RISC-V

- Software cause evaluation (polled exception handling)
  - All exceptions/interrupts start same routine at same address – i.e. for RISC-V **pc** is set to **mtvec** value. For supervisor mode **stvec**.
  - Routine reads source from status register (RISC-V: **mcause** register)
- Vectored exception handling
  - CPU support hardware identifies cause/source/interrupt number
  - Array of ISR start addresses is prepared on fixed or preset (VBR – vector base register) address in main memory
  - CPU computes index into table based on source number
  - CPU loads word from given address to PC
- Non-vectored exception handling with more routines/initial addresses assigned to exception classes and IRQ priorities
- Additional combinations when more addresses are used for some division into classes or some helper HW provides decoding speedup. RISC-V common exception handler address, but optional IRQ mode with exception start address offsets to **mtvec**.

Exception Processing Example – Setup

```
_start:
   addi  a0, zero, 0x101
   la    t0, skip
   csrrw zero, mepc, t0
   mret  // test exception ret

   addi  a0, zero, 0x105
   addi  a0, zero, 0x106
skip:
   addi  a0, zero, 0x107
   csrrs t0, mepc, zero

   ebreak

   la    t0, handle_exception
   csrrw zero, mtvec, t0

   la    t0, task_control_block
   csrrw zero, mscratch, t0

   csrrsi zero,mstatus,8 //MIE=1
```

```
   addi t0, zero, 16 // UART RX
   addi t1, zero, 1
   sll  t1, t1, t0  // bit mask
   csrrs zero, mie, t1

   li   a0, SERIAL_PORT_BASE
   li   t0, SERP_RX_ST_REG_IE_m
   sw   t0, SERP_RX_ST_REG_o(a0)

   // Background task
   addi t0, zero, 0x0001
   li   a0, SPILED_REG_BASE
Loop:
   csrrs t1, mepc, zero // check
   sw   t1,SPILED_REG_LED_LINE(a0)
   srl  t2, t0, 31
   sll  t0, t0, 1
   or   t0, t0, t2
   lw   t2, ..._KNOBS_8BIT(a0)
   sw   t2, ..._LED_RGB1(a0)
   xori t2, t2, -1
   sw   t2, ..._LED_RGB2(a0)
   beq  zero, zero, loop
```

Exception Processing Example – Interrupt Routine

```
handle_exception:
    csrrw   tp, mscratch, tp        // store previous and take system tp
    sw      sp, TCB_SP(tp)          // store stack pointer
    sw      ra, TCB_RA(tp)          // store return address
    sw      t0, TCB_T0(tp)          // store rest of clobberable regs
    sw      a0, TCB_A0(tp)
    ...
    csrr    t0, mcause              // is it Rx interrupt?
    blt     t0, zero, handle_irq    // branch to interrupts processing
    ...
    // handle synchronous exception

ret_from_exception:
    lw      sp, TCB_SP(tp)          // restore stack pointer
    lw      ra, TCB_RA(tp)          // restore return address
    lw      t0, TCB_T0(tp)          // restore rest of clobberable regs
    lw      a0, TCB_A0(tp)
    ...
    csrrw   tp, mscratch, tp        // Swap back TCB to mscratch
    mret                            // Return from exception pc <= mepc
```

Exception Processing Example – Interrupt Routine cont.

```
handle_irq:                             // t0 mcause
    slli    t0, t0, 2                   // shift out sign, left sources * 4
    /* the t0 would be used to point into irq handlers table */
    /* check only for UART RX interupt for simplicity 8 */
    addi    a0, zero, 16 * 4            // UART RX is the first platform irq
    beq     t0, a0, handle_uart_rx_irq // it is UART RX
    /* mask out unknown sources */
    srli    t0, t0, 2                   // make t0 back simple source index
    addi    a0, zero, 1
    sll     a0, a0, t0                  // generate bit mask for source
    csrrc   zero, mie, a0               // mie = mie & ~a0
    j       ret_from_exception

handle_uart_rx_irq:
    li      a0, SERIAL_PORT_BASE   // Setup base of UART
    lw      t0, SERP_RX_DATA_REG_o(a0)// Read received character
    sw      t0, SERP_TX_DATA_REG_o(a0)// echo it back to terminal
    j       ret_from_exception
```

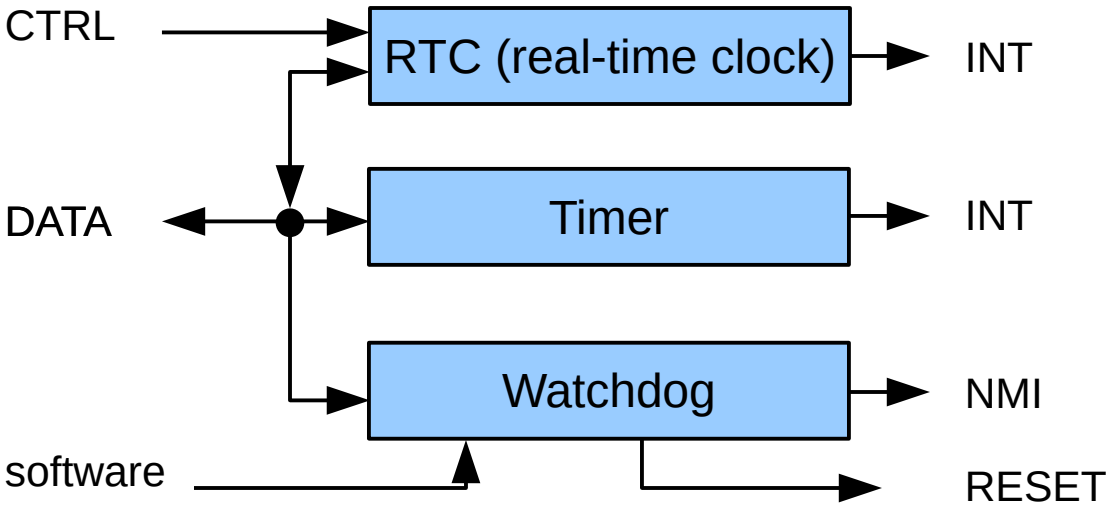
Complete example at
https://gitlab.fel.cvut.cz/b35apo/stud-support/-/blob/master/seminaries/qtrvsim/uart-echo-irq/uart-echo-irq.S
QtRVSim simulator in RISC-V version cannot run this example yet, QtMips MIPS version supports IRQs.

Asynchronous and Synchronous Exceptions/Interrupts

- External interrupts/exceptions are generally asynchronous – i.e. they are not tied to some instruction
  - RESET- CPU state initialization and (re)start form initial address
  - NMI - non-maskable interrupt (temperature/bus/EEC fault)
  - INT - maskable/regular interrupts (peripherals etc.)
- Synchronous exceptions (and or interrupts) are result of exact instruction execution
  - Arithmetic overflow, division by zero etc.
  - TRAP - debugger breakpoint, exception after each executed instruction for single-stepping, etc.
  - Modification of interrupted code flow state (registers, flags, etc.) is expected for some of these causes (unknown instruction emulation, system calls, jump according to program provided exception tables, etc.)

Real-time Clocks and Supervisor (Watchdog) Circuits

- real-time clocks
  - provide real/wall clock time (local/UTC)
- timer
  - periodic or one shot timer interrupt (timer INT), time finctions
- supervisor/watchdog circuits
  - protects system against SW and HW faults and power supply lost/faults (watchdog, power fail)



Programmed Input/Output (PIO) With Polling

```
DoSomethingWithData:
    Wait4Device:
        in( dx, al );
        test( 1, al );
        jnz Wait4Device;
     << Do something with the Data>>
     jmp DoSomethingWithData;
```

Example: Randall Hyde (randyhyde_at_earthlink.net) e-mail 14 Jun 2004

- The most inferior solution, CPU waits in a loop for data ready (busy wait)
- Even if is not possible to use CPU at that time do do some other valuable work (more about time sharing, multi processing, threading, user and scheduling later), the looping results in energy/power waste

Interrupt Driven Programmed Input/Output (PIO) on x86

```
InterruptServiceRoutine:
      << Get data and move to a shared memory location >>
      mov( 1, DataAvailable );
      iret();

MainThreadLoop:
    << Tell I/O device we want data >>
    Wait4Data:
         OptionalHALT or OtherDataProcessing;
         test( 1, DataAvailable );
         jnz Wait4Data;
     <<Do Something With Data >>
     jmp MainThreadLoop;
```

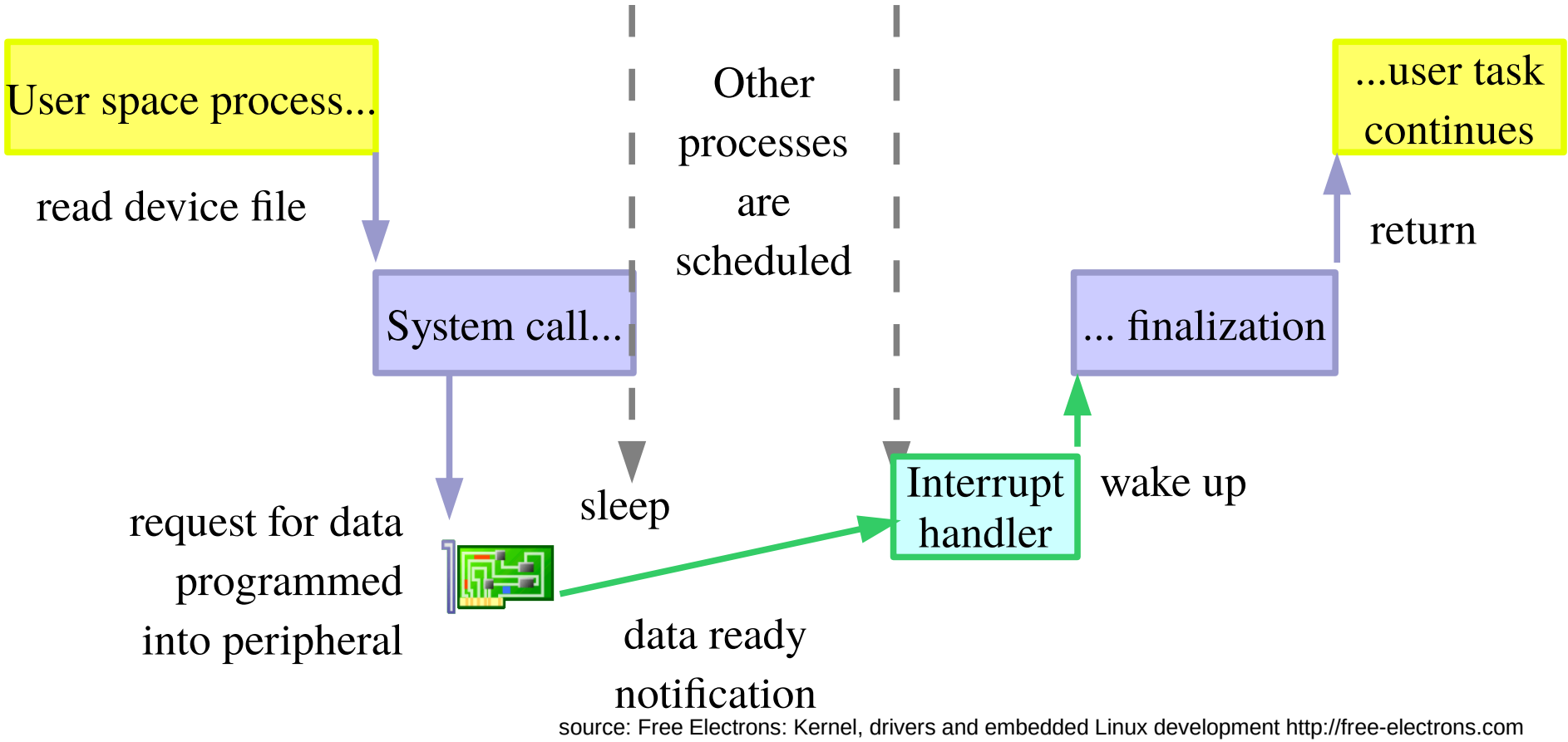
- Peripheral takes care for data availability signaling to CPU – the interrupt signal is activates and interrupt/exception is serviced
- The overall situation is not better for above shown example, but if task scheduling is added then actual/waiting task can be suspended and some other ready/released task can proceed and use CPU until data arrival. Then suspended task is activated again at end of interrupt processing

Linux Kernel: Event Waiting with Context Switch – Schedule

```
static DECLARE_WAIT_QUEUE_HEAD(foo_wq);
volatile int event_pending;

irqreturn_t foo_irq_fnc(int intno, void *dev_id)
{
  <<read device status, store what can be lost and stop/mask IRQ>>
  event_pending = <<indicate even arrival>>;
  wake_up_interruptible(&foo_wq);
  return IRQ_HANDLED;
}

static ssize_t foo_read(struct file *fp, char __user *buf,
                        size_t len, loff_t *off)
{
  wait_event_interruptible_timeout(foo_wq, event_pending != 0);
  << check error state etc. signal_pending(current) >>
  << process event_pending and event_pending = 0 >>
  err = copy_to_user(buf, internal_buffer, len);
  return len;
}
```

Interrupt – Operating Systems Level I/O Processing

When peripheral transfers data, task is suspended/waiting (and other work could be done by CPU). Data arrival results in IRQ processing, CPU finalizes transfer and original task continues



source: Free Electrons: Kernel, drivers and embedded Linux development http://free-electrons.com

RTEMS: Wait for Event with use of Scheduler

```
rtems_isr mmcsd_irq_handler(rtems_irq_hdl_param data)
{
  MMCSD_Dev *device=(MMCSD_Dev *)data;
  rtems_event_send(device->waiter_task_id, MMCSD_WAIT_EVENT);
}

static int mmcsd_read(MMCSD_Dev *device, rtems_blkdev_request *req)
{
  rtems_status_code status;
  rtems_event_set   events;
  rtems_interval    ticks;
  rtems_id          self_tid;

  rtems_task_ident(RTEMS_SELF, 0, &self_tid);
  device->waiter_task_id = self_tid;
  status=rtems_event_receive(MMCSD_WAIT_EVENT | MMCSD_EVENT_ERROR,
                             RTEMS_EVENT_ANY|RTEMS_WAIT, ticks, &events);
  << process event fill sg = req->bufs - List of scatter/gather buffers >>
  req->req_done(req->done_arg, RTEMS_SUCCESSFUL, 0);
  return 0;
}
```

- The example is simplified. Temporary task (TID) registration in the driver state structure is not used. The device is serviced by **worker thread** which is created during driver/its instance initialization.

RTEMS: Semaphore Used for Interrupt Event Notification

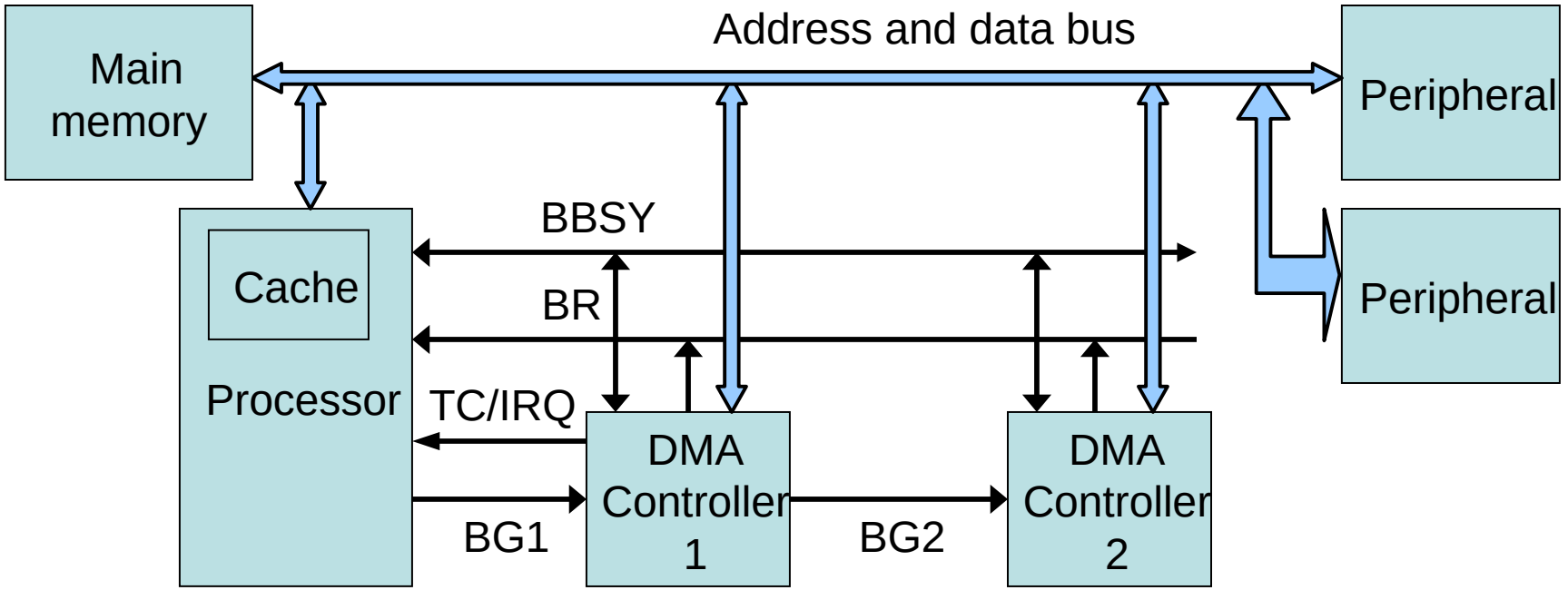
```
static rtems_id my_semaphore;

rtems_isr my_irq_handler(rtems_irq_hdl_param valu)
{
        if (<<check if really from device>>) {
                rtems_semaphore_release(my_semaphore);
        }
}

wait for event
    rtems_semaphore_obtain(semaphore, RTEMS_WAIT, RTEMS_NO_TIMEOUT);

initialize semaphore in the driver init
    rtems_semaphore_create(rtems_build_name('s','e','m','a'),
                  0/*initial value*/, RTEMS_FIFO, 5/*priority*/,
                  &my_semaphore/*location to store new sem ID*/);
```

- Similar semaphore based solution can be used for VxWorks or Linuxu kernel. These APIs are internal kernel mechanisms, POSIX/ANSI standards does not specify mechanisms for interrupts management and servicing.

Windows: Interrupt and Deferred Procedure Call

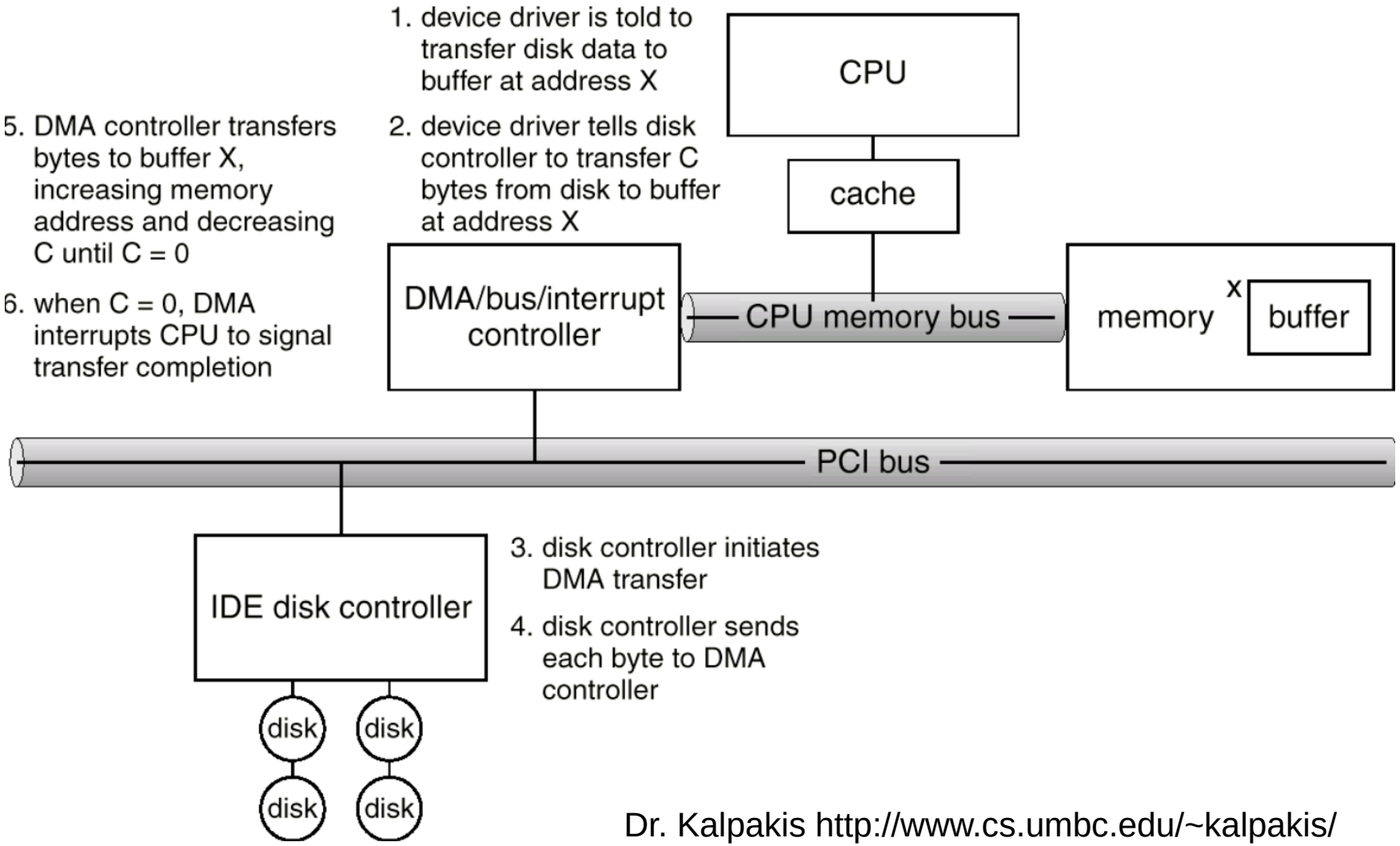
```
VOID NTAPI ulan_bottom_dpc(IN PKDPC Dpc,IN PVOID contex,
                      IN PVOID arg1,IN PVOID arg2);

KSERVICE_ROUTINE InterruptService;
BOOLEAN uld_irq_handler( _In_  struct _KINTERRUPT *Interrupt,
  _In_  PVOID ServiceContext)
{
    …
    KeInsertQueueDpc(&(udrv)->bottom_dpc,NULL,NULL);
    return TRUE;
}

status =
IoConnectInterrupt(&udrv->InterruptObject,
    uld_irq_handler,            // ServiceRoutine
    udrv,                       // ServiceContext
    NULL,                       // SpinLock
    udrv->irq,                  // Vector
    udrv->Irql,                 // Irql
    udrv->Irql,                 // SynchronizeIrql
    udrv->InterruptMode,        // InterruptMode
    TRUE /*FALSE for ISA? */,   // ShareVector
    udrv->InterruptAffinity,    // ProcessorEnableMask
    FALSE);                     // FloatingSave
```

Direct Memory Access - DMA



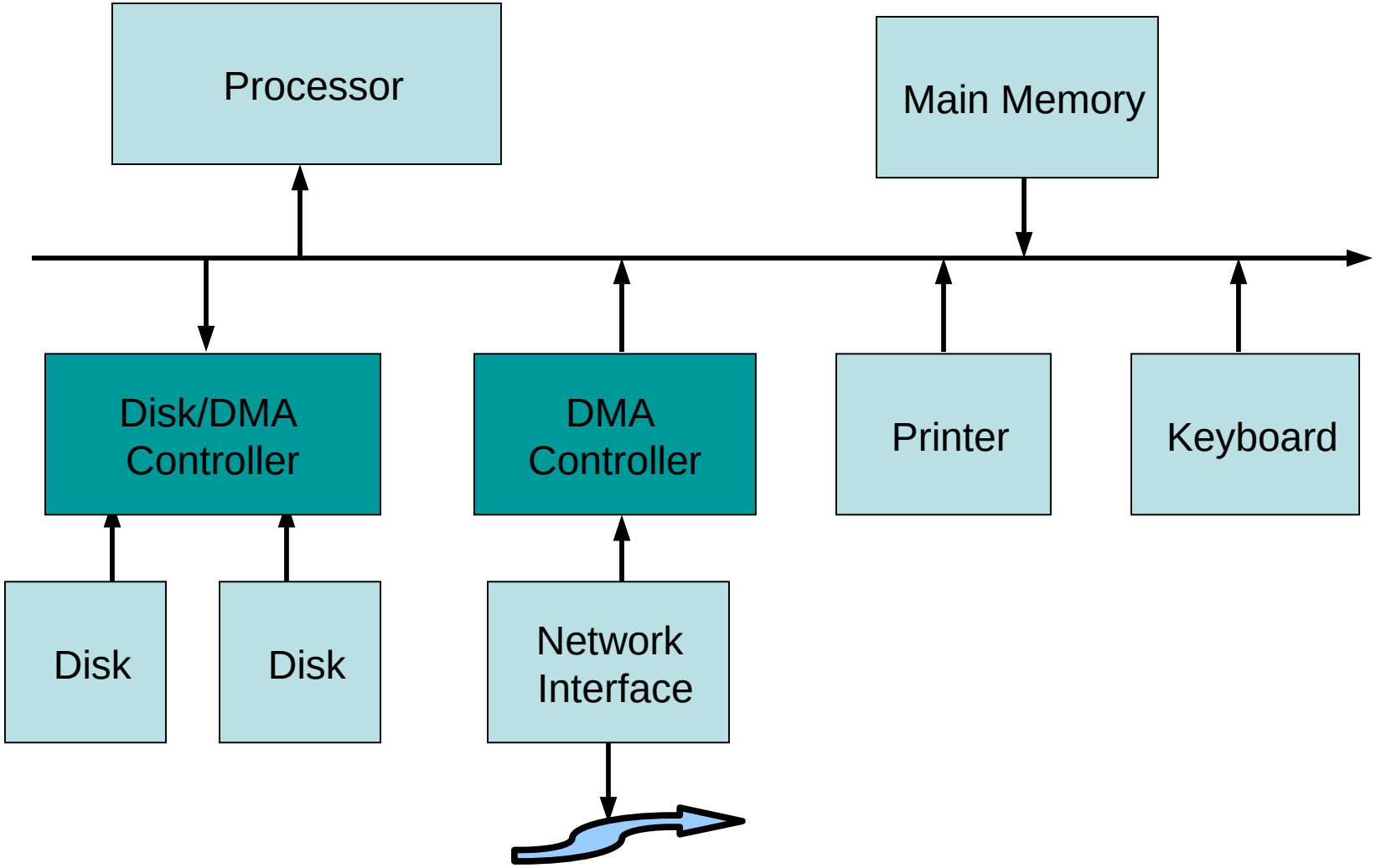
- Computer system is equipped by unit(s) specialized for data transfers
- Large size data transfers do not trash/displace data at CPU caches
- Program/OS initializes peripheral and setups parameters for transfer
- Then DMA unit source, destination, request line are programmed, DMA unit signals end of the transfer by interrupt

Example of DMA Transfer from Hard-disk

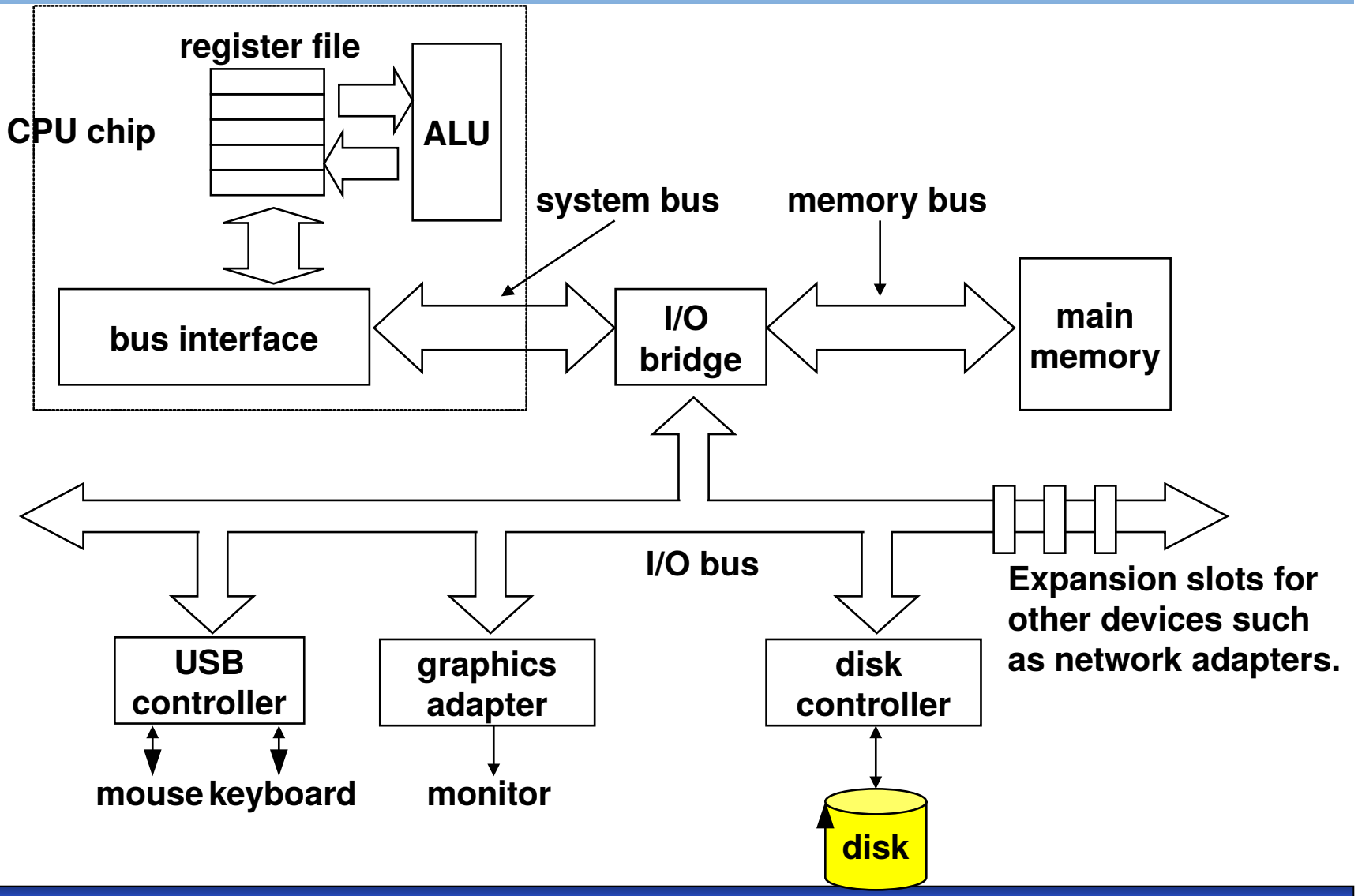


Dr. Kalpakis http://www.cs.umbc.edu/~kalpakis/

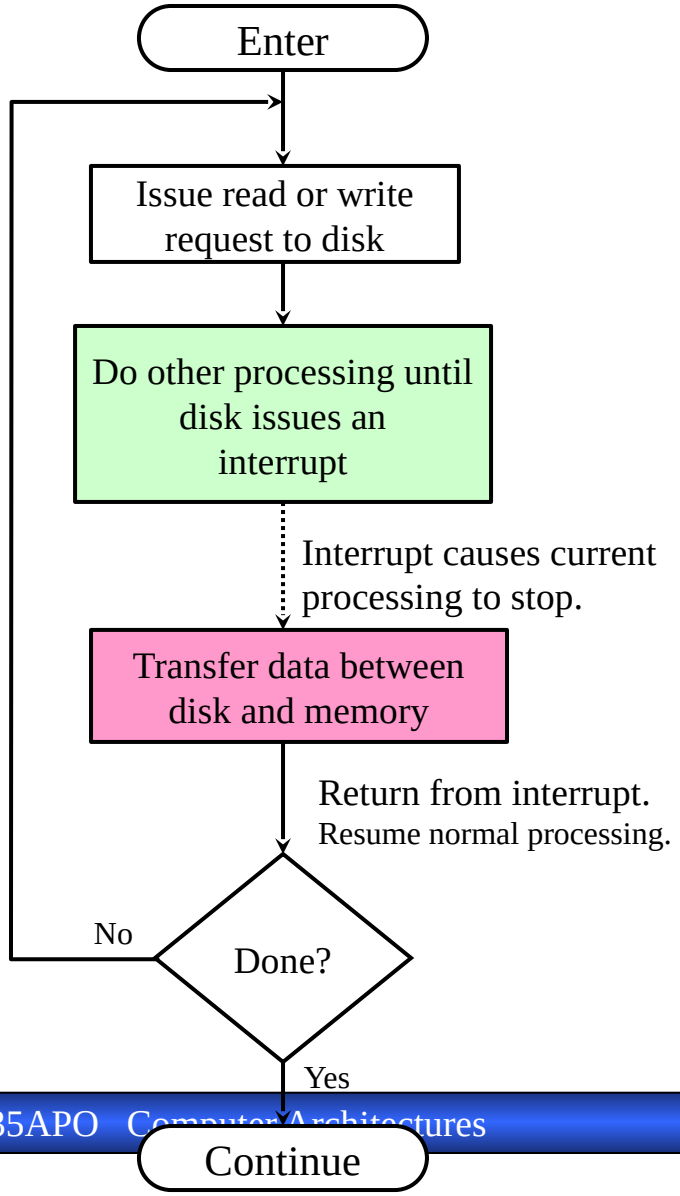
Decentralized Controllers/DMA – Integration into Peripherals



A Typical Hardware System



Slow Interrupt Driven Disk Transfer

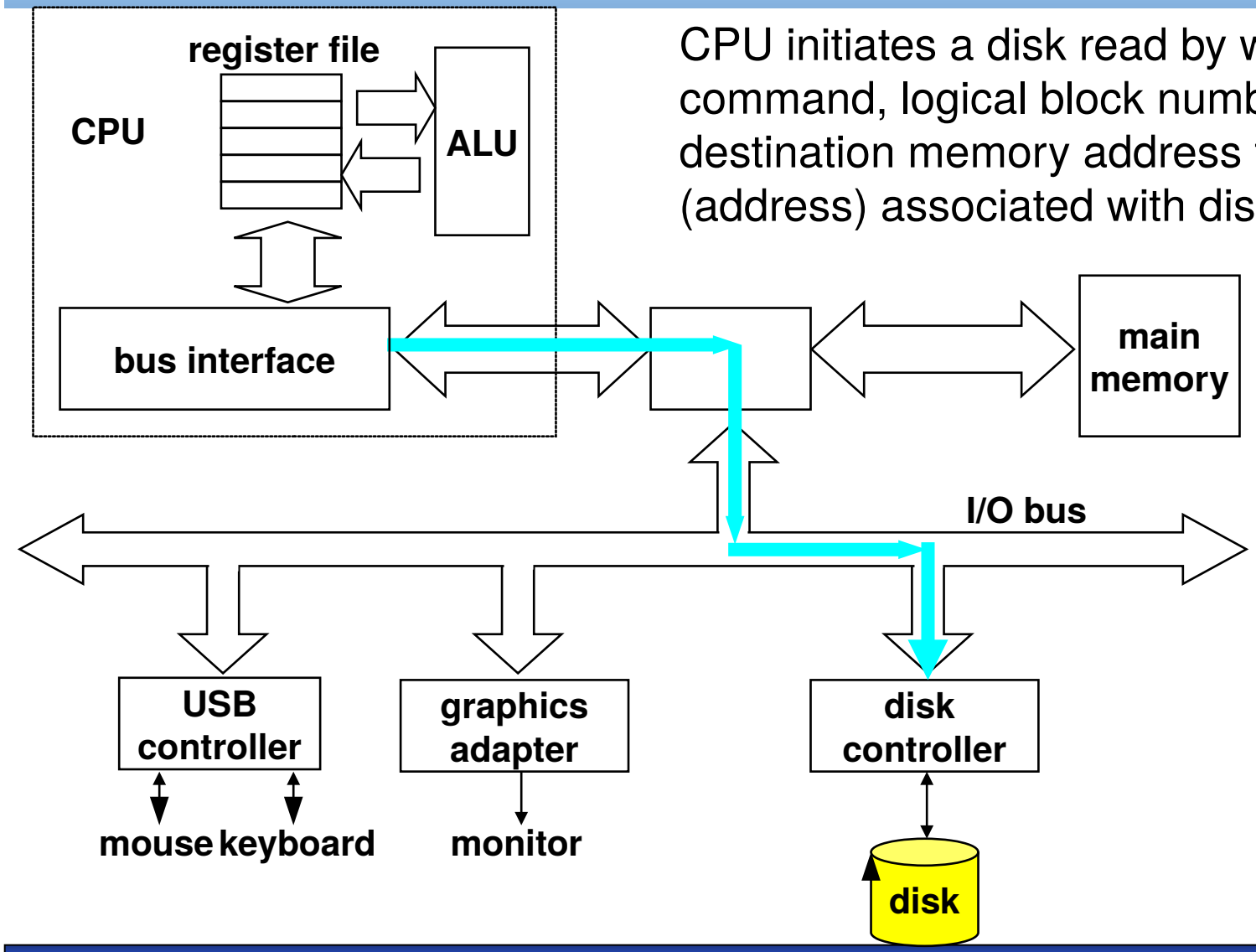


1. CPU issues read command
2. I/O module gets data from peripheral while CPU performs other work
3. I/O module interrupts CPU
4. CPU requests data
5. I/O module transfers data

Bus Master DMA and IO (Co)Processors

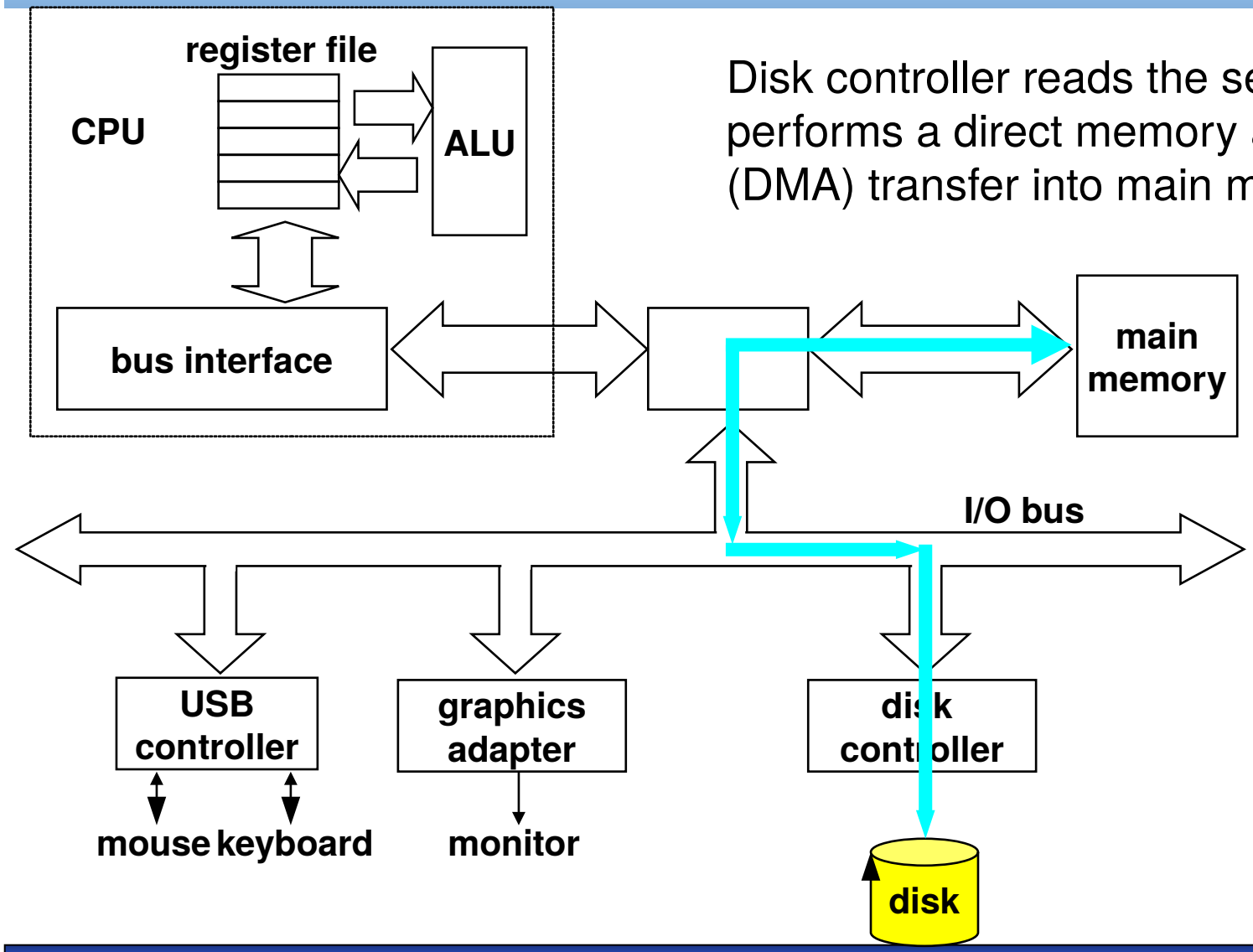
- Intelligent peripherals
- Peripheral is equipped by own controller (CPU)
  - Finite state machine
  - Input/output processor (IOP) etc.
- Transfer processing sequence
  - Superordinate CPU/system stores sequence of the data and control blocks into main memory
  - Configures or programs controller integrated into peripheral and that controls data transfers from/to main memory
  - After all transfers are finished (sometimes after the whole first packet received) signals CPU that state by interrupt
- CPU/operating system processes interrupt and reschedules to task waiting for data

DMA Reading a Disk Sector: Step 1



CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.

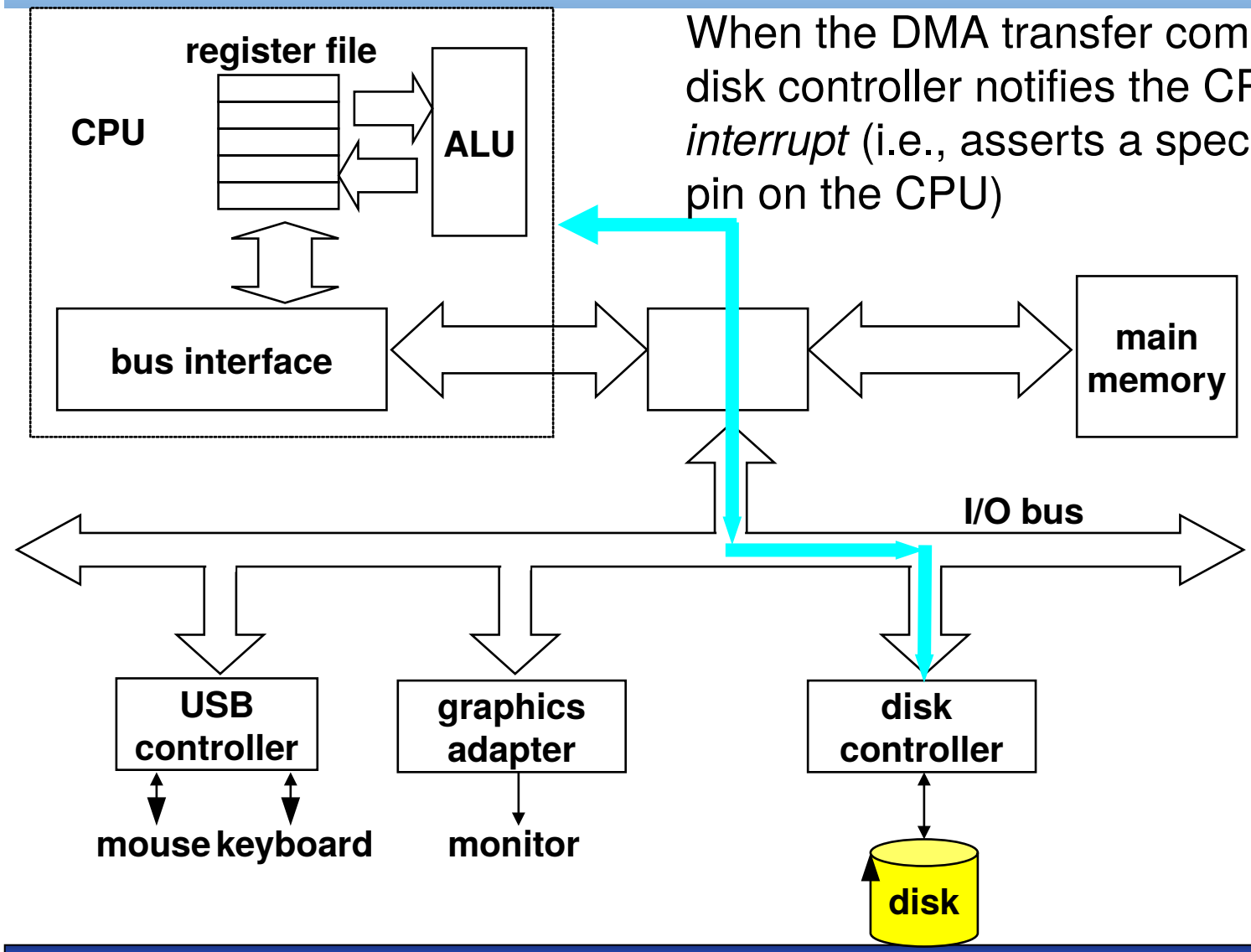
DMA Reading a Disk Sector: Step 2



Disk controller reads the sector and performs a direct memory access (DMA) transfer into main memory.

Reading a Disk Sector: Step 3

When the DMA transfer completes, the disk controller notifies the CPU with an *interrupt* (i.e., asserts a special "interrupt" pin on the CPU)



Where the problems lie? DMA and I/O pitfalls

Memory Mapped Peripherals and Data Consistency/Coherence

- Input/output operations and CPU
  - The caching has to be disabled for address ranges where input and or output ports/registers/memory is mapped
  - Pipelined instruction processing alone does not cause problems (except for read after write)
  - Data forwarding, subsequent access (load/store) bypassing and out of order instructions processing collides with I/O code
  - Special synchronization instructions or HW support on CPU level is then necessary to stall instruction execution till (all) previous transfers finis
    - MIPS IV - **sync** (l*x* a sx is finished before subsequent l*x*)
    - RISC-V – **fence** instruction group, strongly ordered I/O regions
    - PowerPC
      - **eieio** (Enforce In-Order Execution of I/O) Instruction
      - **sync** not only for I/O access but even for I memory reads
  - The similar has to be done on compiler level to suppress unintended optimizations (volatile, ...)

Paul E. McKenney: Memory Ordering in Modern Microprocessors

Wikipedia: http://en.wikipedia.org/wiki/Memory_ordering

Atomic Operations, Compilers and STL

- C++ std::atomic_int, std::atomic_intptr_t, …

```
typedef enum memory_order
{
    memory_order_relaxed, memory_order_consume,
    memory_order_acquire, memory_order_release,
    memory_order_acq_rel, memory_order_seq_cst
} memory_order;
```

- C1x

Good source of information for C/C++ language and standard libraries is https://en.cppreference.com
For std::atomic
https://en.cppreference.com/w/cpp/header/atomic

C++11 Memory Model and GCC implementation

C++11 memory models

- __ATOMIC_RELAXED – No barriers or synchronization.
- __ATOMIC_CONSUME – Data dependency only for both barrier and synchronization with another thread.
- __ATOMIC_ACQUIRE – Barrier to hoisting of code and synchronizes with release (or stronger) semantic stores from another thread.
- __ATOMIC_RELEASE – Barrier to sinking of code and synchronizes with acquire (or stronger) semantic loads from another thread.
- __ATOMIC_ACQ_REL – Full barrier in both directions and synchronizes with acquire loads and release stores in another thread.
- __ATOMIC_SEQ_CST – Full barrier in both directions and synchronizes with acquire loads and release stores in all threads.

Atomic Operations Defined by C++11 Standard

- type __atomic_load_n (type *ptr, int memmodel)
  RELAXED, SEQ_CST, ACQUIRE and CONSUME
- void __atomic_load (type *ptr, type *ret, int memmodel)
- __atomic_store_n (type *ptr, type val, int memmodel)
  RELAXED, SEQ_CST, RELEASE
- void __atomic_store (type *ptr, type *val, int memmodel)
- __atomic_exchange_n (type *ptr, type val, int memmodel)
  RELAXED, SEQ_CST, ACQUIRE, RELEASE and ACQ_REL
- void __atomic_exchange (type *ptr, type *val, type *ret, int memmodel)

C++11 Compare and Swap (CAS)

- bool __atomic_compare_exchange_n (type *ptr, type *expected, type desired, bool weak, int success_memmodel, int failure_memmodel)
- bool __atomic_compare_exchange (type *ptr, type *expected, type *desired, bool weak, int success_memmodel, int failure_memmodel)

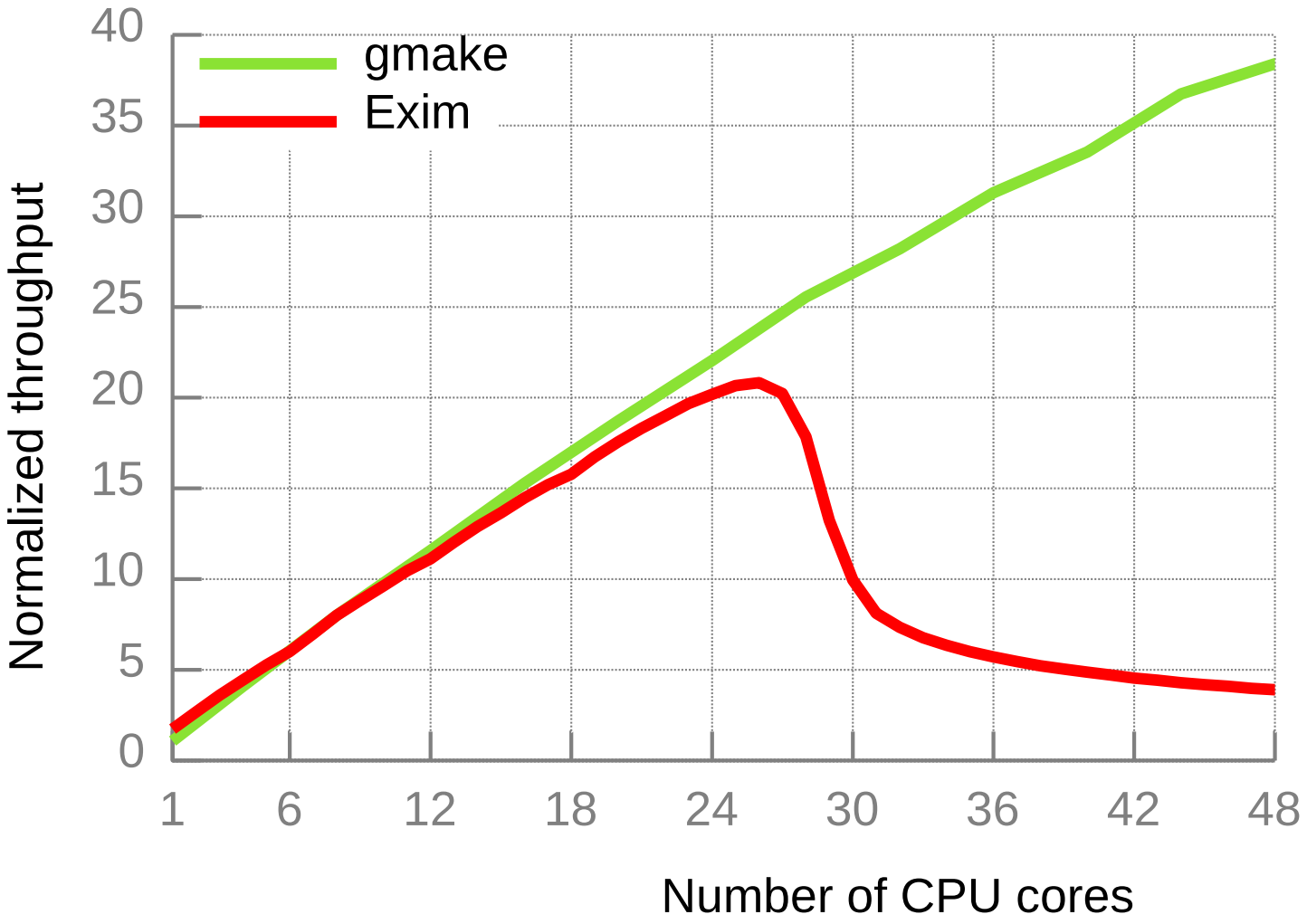
```
int compare_and_swap(int* ptr, int oldval, int newval)
{
  ATOMIC();
  int old_reg_val = *ptr;
  if (old_reg_val == oldval)
     *ptr = newval;
  END_ATOMIC();
  return old_reg_val;
}
```

C++11 Arithmetic and Logic Operations

- type __atomic_add_fetch (type *ptr, type val, int memmodel)

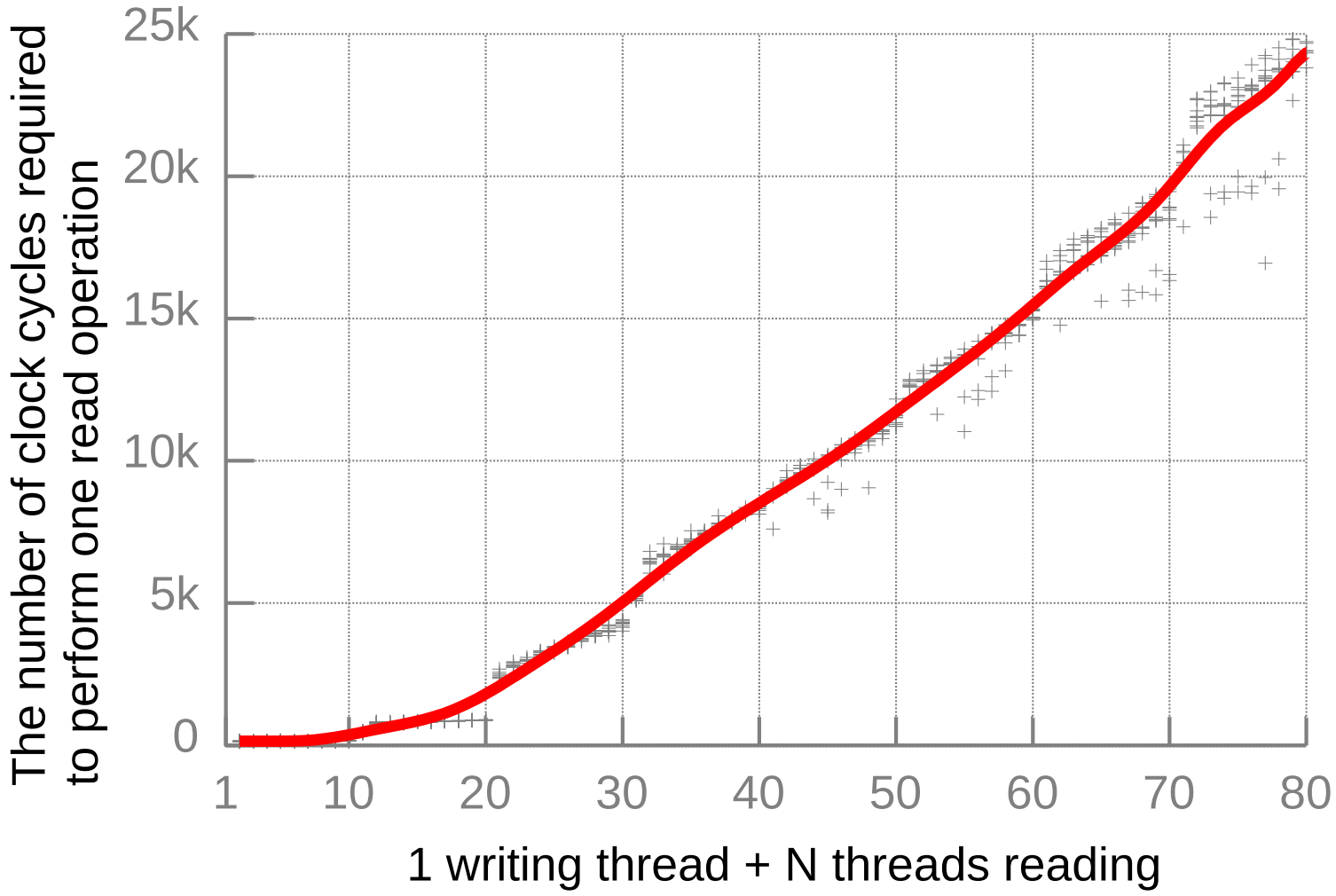
  add, sub, and, xor, or, nand
- type __atomic_fetch_add (type *ptr, type val, int memmodel)
- bool __atomic_test_and_set (void *ptr, int memmodel)
- void __atomic_clear (bool *ptr, int memmodel)
- void __atomic_thread_fence (int memmodel)
- void __atomic_signal_fence (int memmodel)
- bool __atomic_always_lock_free (size_t size, void *ptr)
- bool __atomic_is_lock_free (size_t size, void *ptr)

Scalability Bottleneck in Memory Access from Multiple Cores

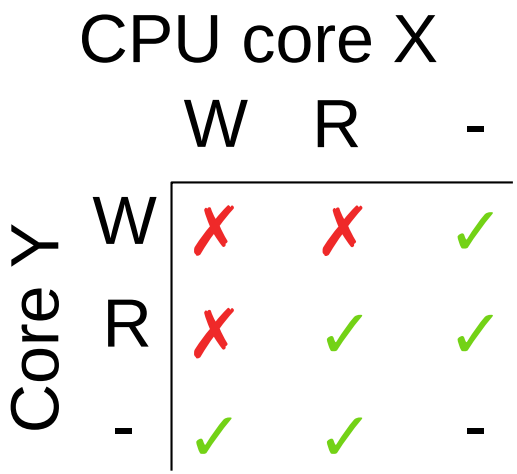


Example of single shared written cache line ruining application throughput

Price of Collisions in Single Row of the Memory Cache



Which Algorithms and Approaches are Scalable?



Source
The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors by Austin T. Clements

Program Constructions That Are Scalable for Multiple Threads

- Scalability: use scalable data structures
  - Linear arrays and arrays radix
  - Hash tables
  - **Do not** use binary / balanced trees for shared data
- Delaying action / cleaning - defer work, reference tracking, read copy update RCU postponed release / cancellations
- Prevent pessimistic operations by optimist check
  - Only when the check of the object determines that change is required proceed with actions required for change (locking etc.) of an entry or file file, etc.
- At the level of work with the operating system use only such operation that is necessary
- Use access (F_OK) to check existence of a file instead of checking the return code of the open or read operations

DMA and Data Consistency

- DMA transfers originate/target main memory bypassing cache
- CPU writes has to be finished before (writeback!)
- Data from peripheral stored to memory cannot be used unitila (partial) cache invalidation or previous flush is issued
- CPU/memory management unit needs to control cacheability of given pages/cache rows
  - PowerPC
    - **dcbf** (Data Cache Block Flush), **clcs** (Cache Line Compute Size), **clf** (Cache Line Flush), **cli** (Cache Line Invalidate), **dcbi** (Data Cache Block Invalidate), **dcbst** (Data Cache Block Store), **dcbt** (Data Cache Block Touch), **dcbtst** (Data Cache Block Touch for Store), **dcbz**/**dclz** (Data Cache Block Set to Zero), **dclst** (Data Cache Line Store), **icbi** (Instruction Cache Block Invalidate), **sync** (Synchronize)/**dcs** (Data Cache Synchronize)
  - MIPS – specialized instruction named **cache**
  - RISC-V
    - **Fence**, more variants, memory usually coherent between cores and DMA