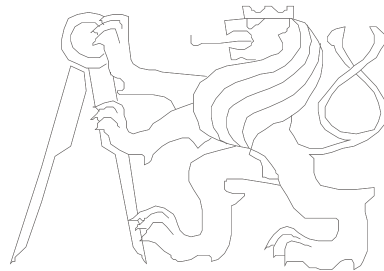
Computer Architectures

External events processing and protection

Pavel Píša, Michal Štepanovský, Miroslav Šnorek



Czech Technical University in Prague, Faculty of Electrical Engineering

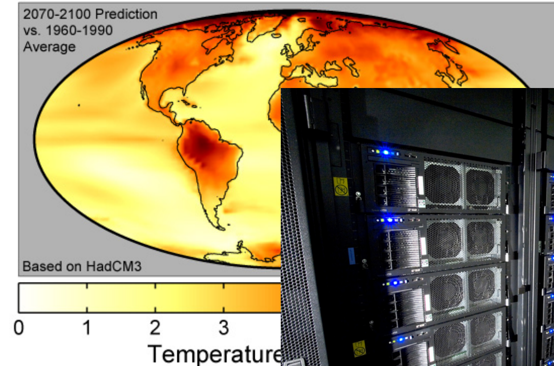
Basic building blocks (repeating)

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

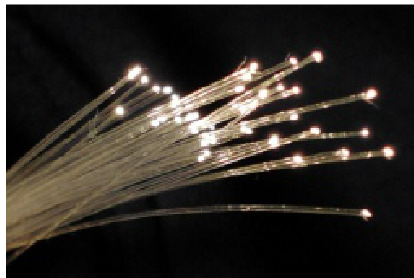
What is purpose to have these building blocks

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.)

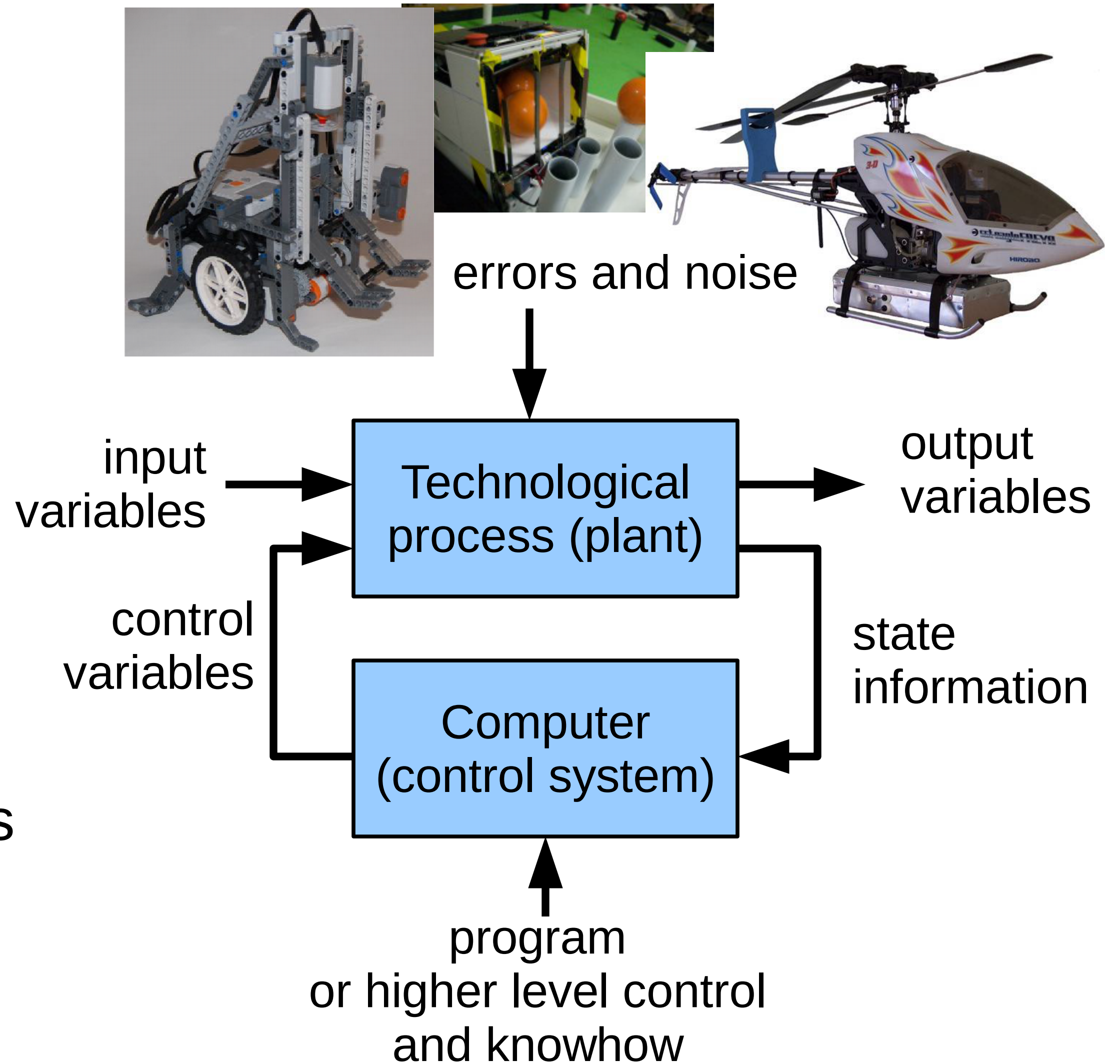


Communications, as a main target (phone, mobile) or as a way to achieve data exchange for other tasks and applications

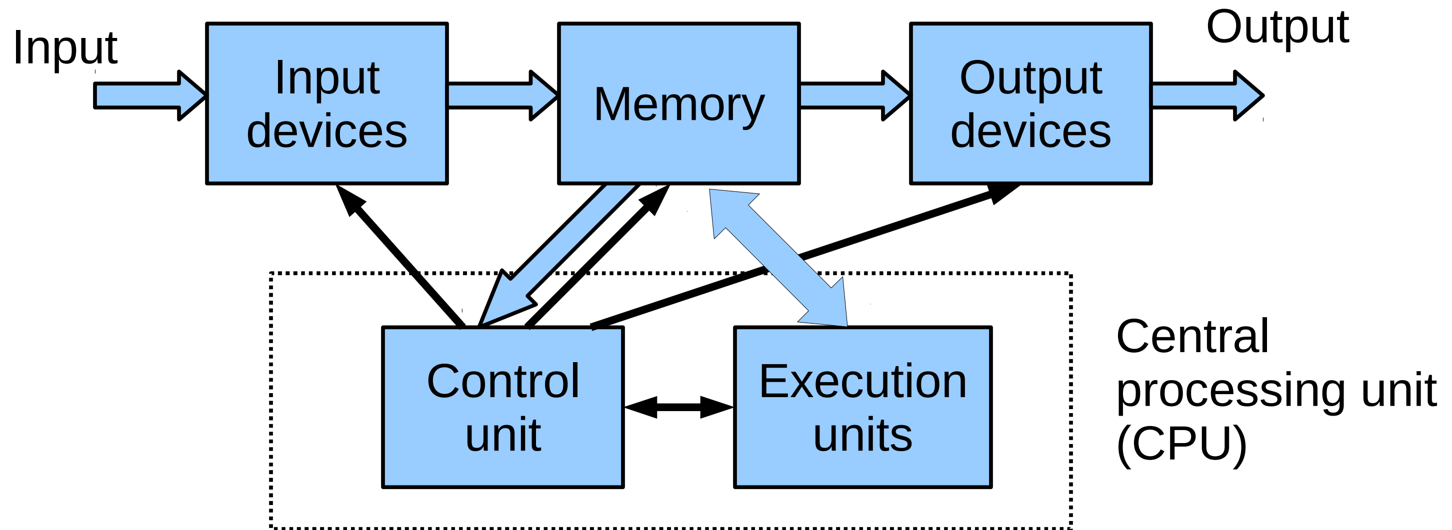
And many others areas of use ...

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 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

- 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 of transferring data between peripheral 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

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)

```
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;
}
```

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

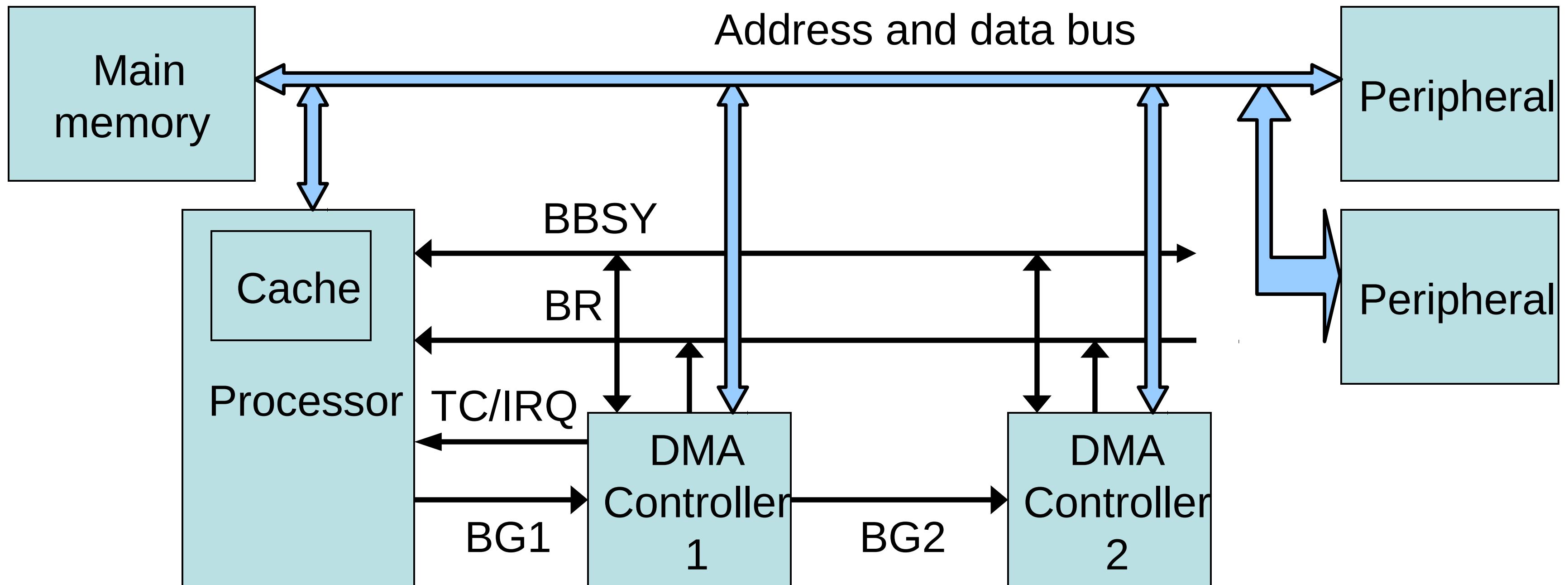
```
static rtems_id my_semaphore;

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 do not specify mechanisms for interrupts management and servicing.

Windows: Interrupt and deffered procedure call

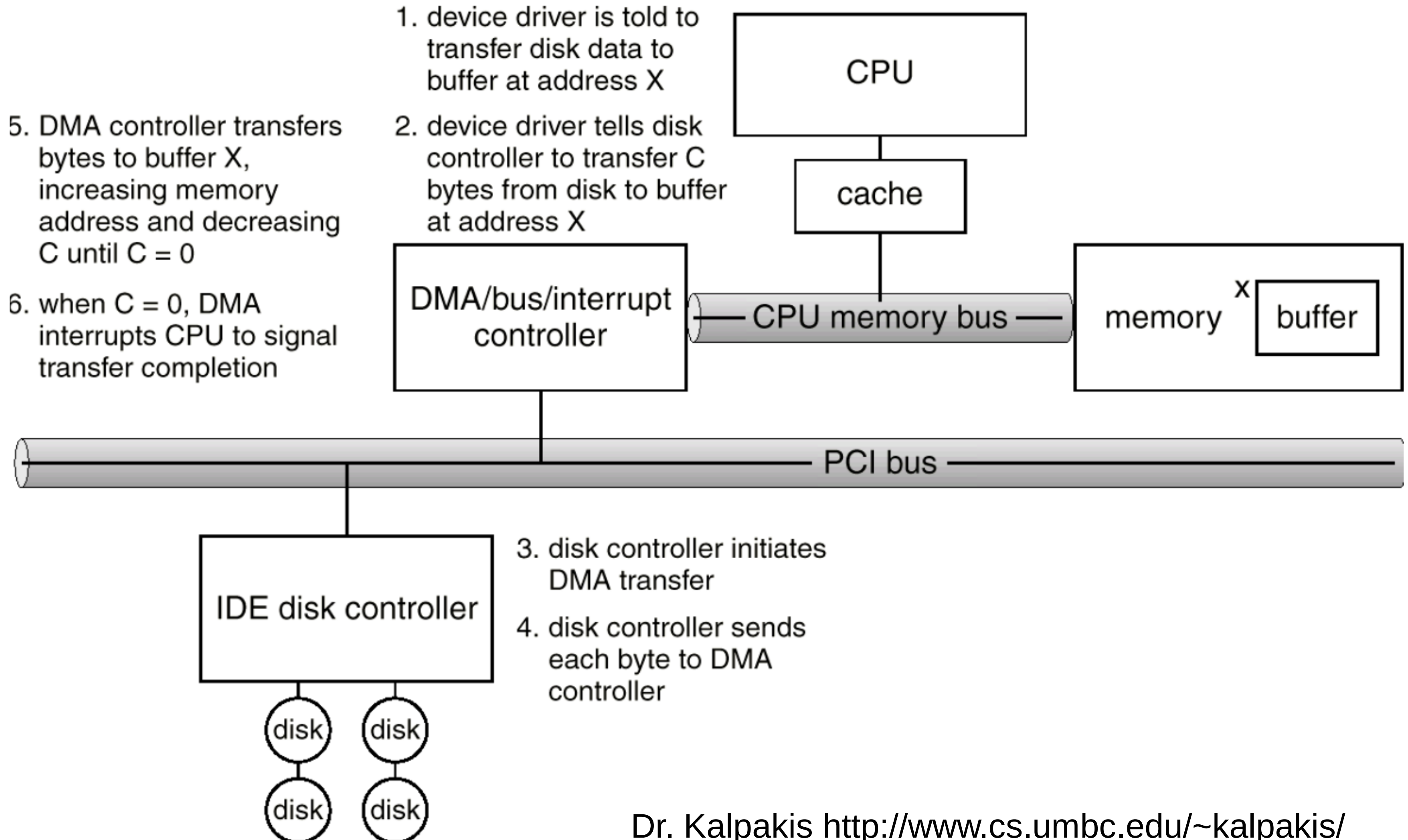
```
VOID NTAPI ulan_bottom_dpc(IN PKDPC Dpc,IN PVOID contex,
                     IN PVOID arg1,IN PVOID arg2);

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



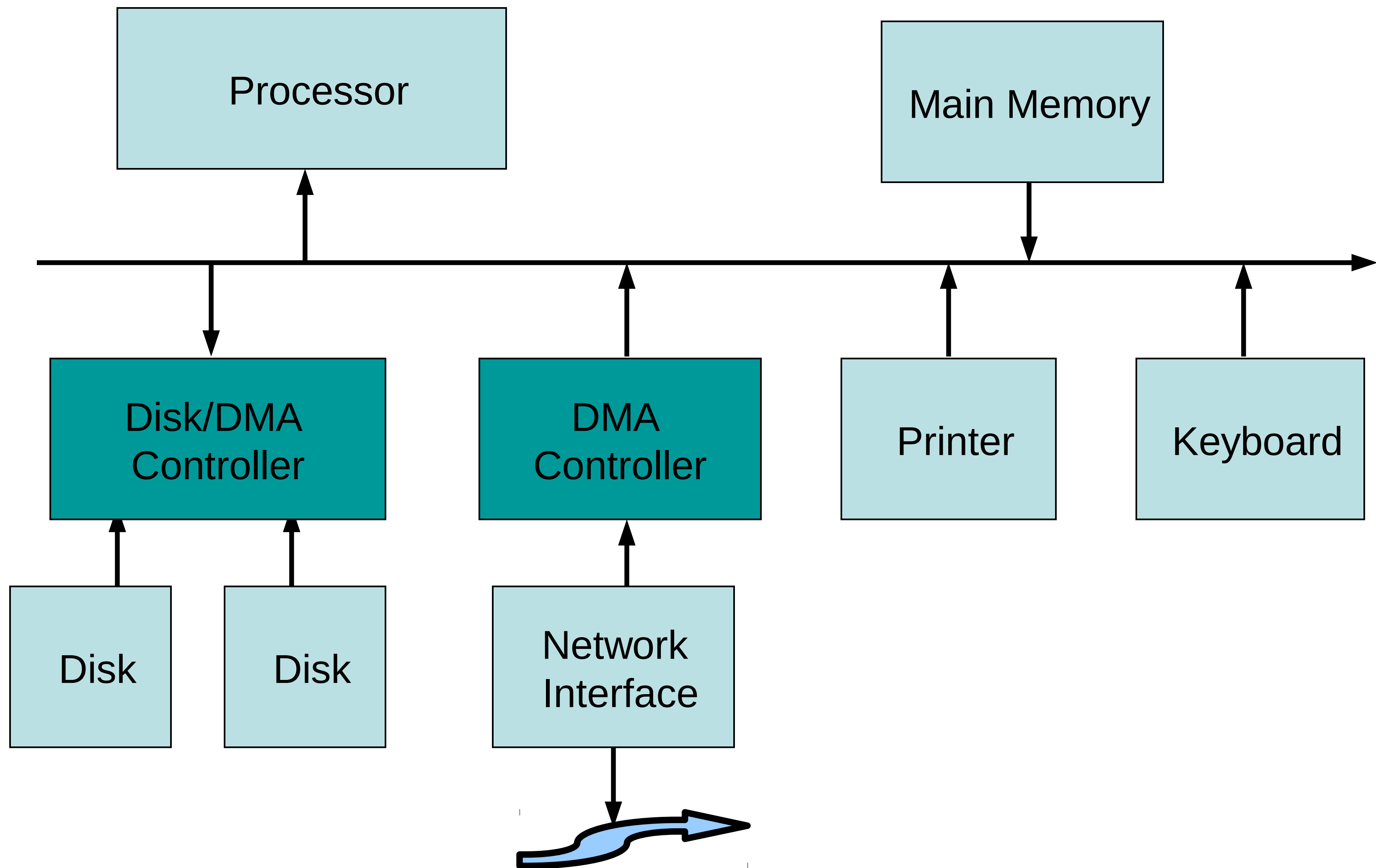
- Computer system is equipped by unit(s) specialized for data transfers
- Large size data transfers do not trash/displace data at CPU caches
- Program/OS initializes peripheral and setups parameters for transfer
- Then DMA unit source, destination, request line are programmed, DMA unit signals end of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

Example of DMA transfer for hard-disk



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

Decentralized controllers/DMA – integration into peripherals



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

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*)
    - 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

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

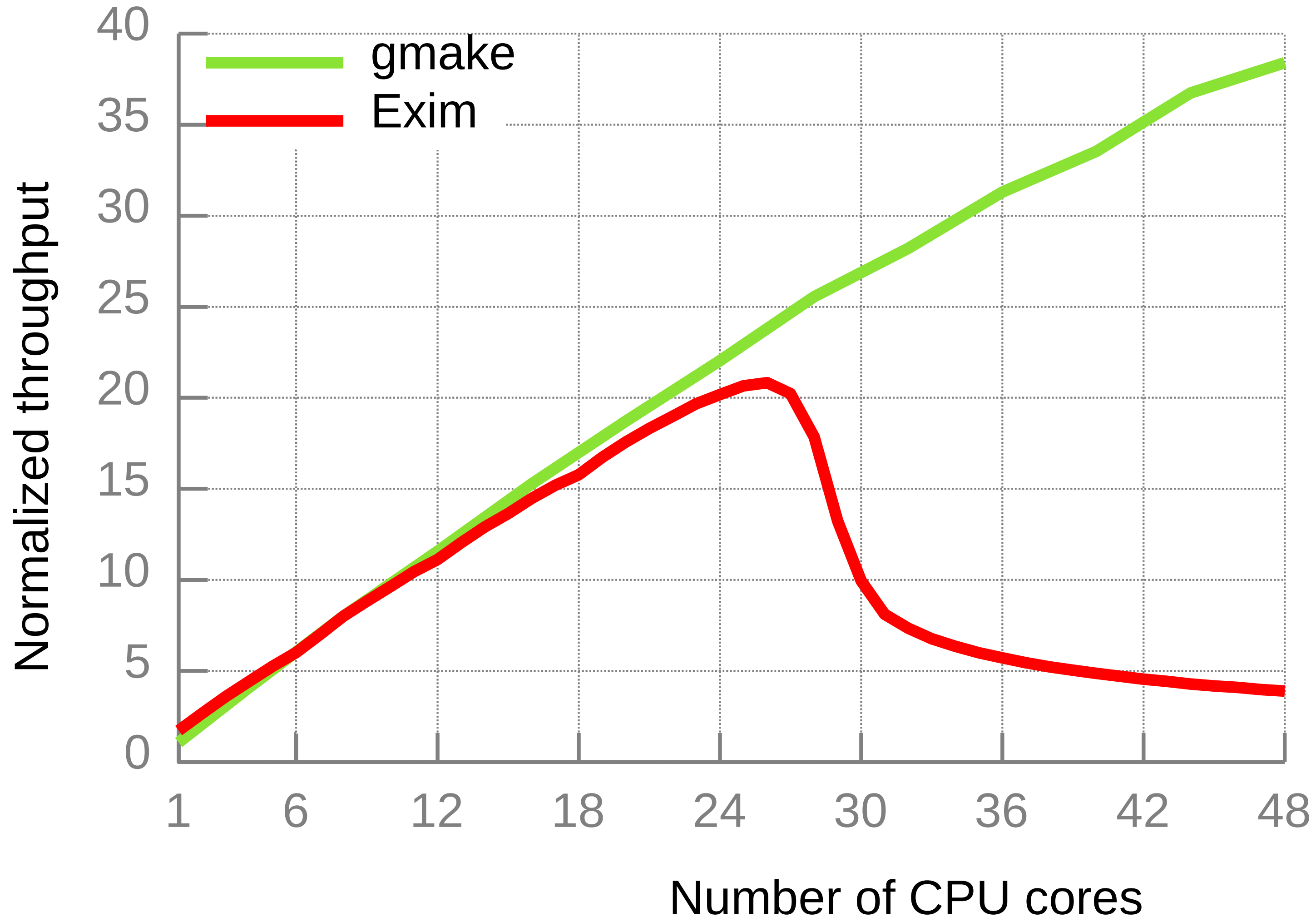
- bool __atomic_compare_exchange_n (type *ptr, type *expected, type desired, bool weak, int success_memmodel, int failure_memmodel)
- bool __atomic_compare_exchange (type *ptr, type *expected, type *desired, bool weak, int success_memmodel, int failure_memmodel)

C++11 Arithmetic and Logic Operations

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

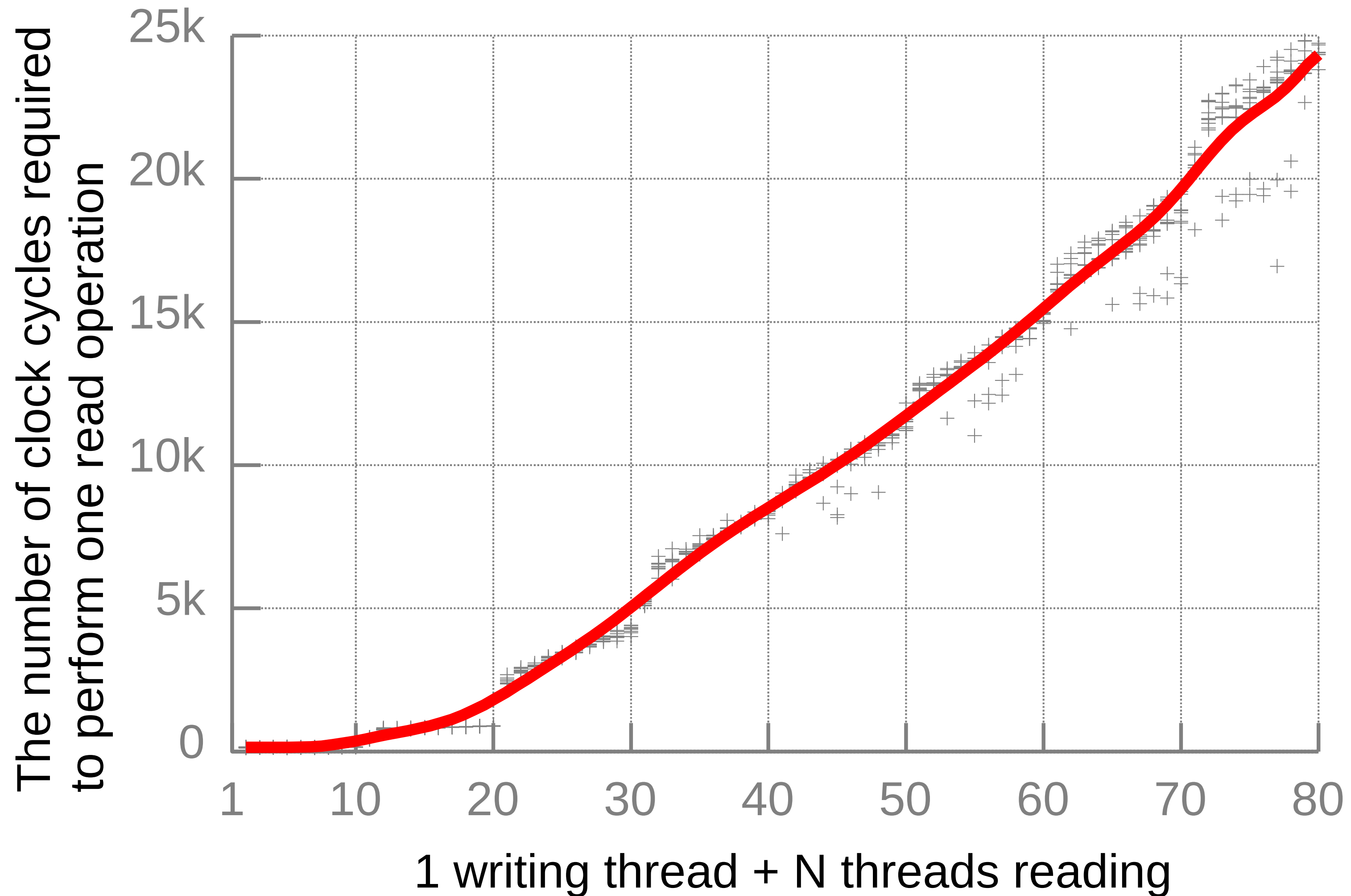
  add, sub, and, xor, or, nand
- type __atomic_fetch_add (type *ptr, type val, int memmodel)
- bool __atomic_test_and_set (void *ptr, int memmodel)
- void __atomic_clear (bool *ptr, int memmodel)
- void __atomic_thread_fence (int memmodel)
- void __atomic_signal_fence (int memmodel)
- bool __atomic_always_lock_free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

Core Y \ CPU core X	W	R	-
W	✗	✗	✓
R	✗	✓	✓
-	✓	✓	-

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**

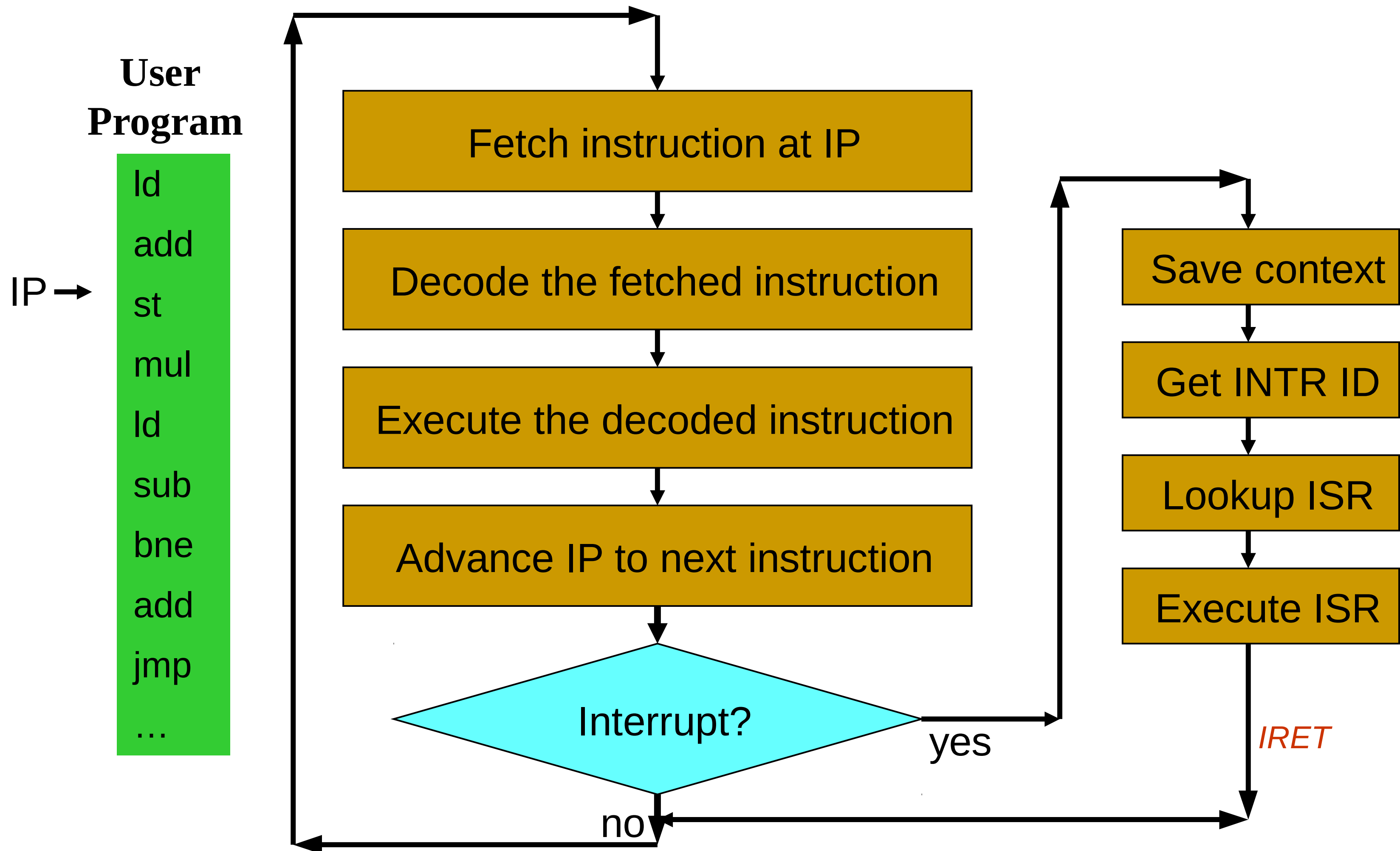
Exceptions and interrupts

- Exceptions – anomalous or exceptional situations (blocking further regular execution) requiring special processing
  - In a MIPS CPU case next main sources are recognized
    - Arithmetic overflow (result for integer/saturated arithmetic not fit)
    - Undefined instruction is to be executed (unknown opcode for IR type instruction or unknown function for R type)
    - 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

Block diagrams of exception processing



MIPS – registers for exceptions status and control

Register name	Register number	Usage
Status	12	Interrupt mask and enable bits
Cause	13	Exception type
EPC	14	Following address of the instruction where the exception occurred

Cause register

Number	Name	Description
00	INT	External Interrupt
01	IBUS	Instruction bus error (invalid instruction)
10	OVF	Arithmetic overflow
11	SYSCALL	System call

Status register - for disabling interrupts and exceptions

Bit	Interrupt/exception
3	INT
2	IBUS
1	OVF
0	SYSCALL

MIPS – exception/interrupt processing

CPU accepts interrupt request, exception or **syscall** opcode

```
EPC <= PC
Cause <= (cause code for event)
Status <= Status << 4
PC <= (handler address)
```

Interrupt service routine/exception handler startup is responsible for
- identification of request cause from co-processor 0 **mfc0 rd, rt**
- CPU state can be controlled by instruction **mtc0 rd, rt**
- **rd** is gen. purpose register, **rt** is one of co-processor 0 registers

The **rfe** instruction finalizes exception handling and returns to previous state

```
PC <= EPC
Status <= Status >> 4
```

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.

Evaluation of the exception source

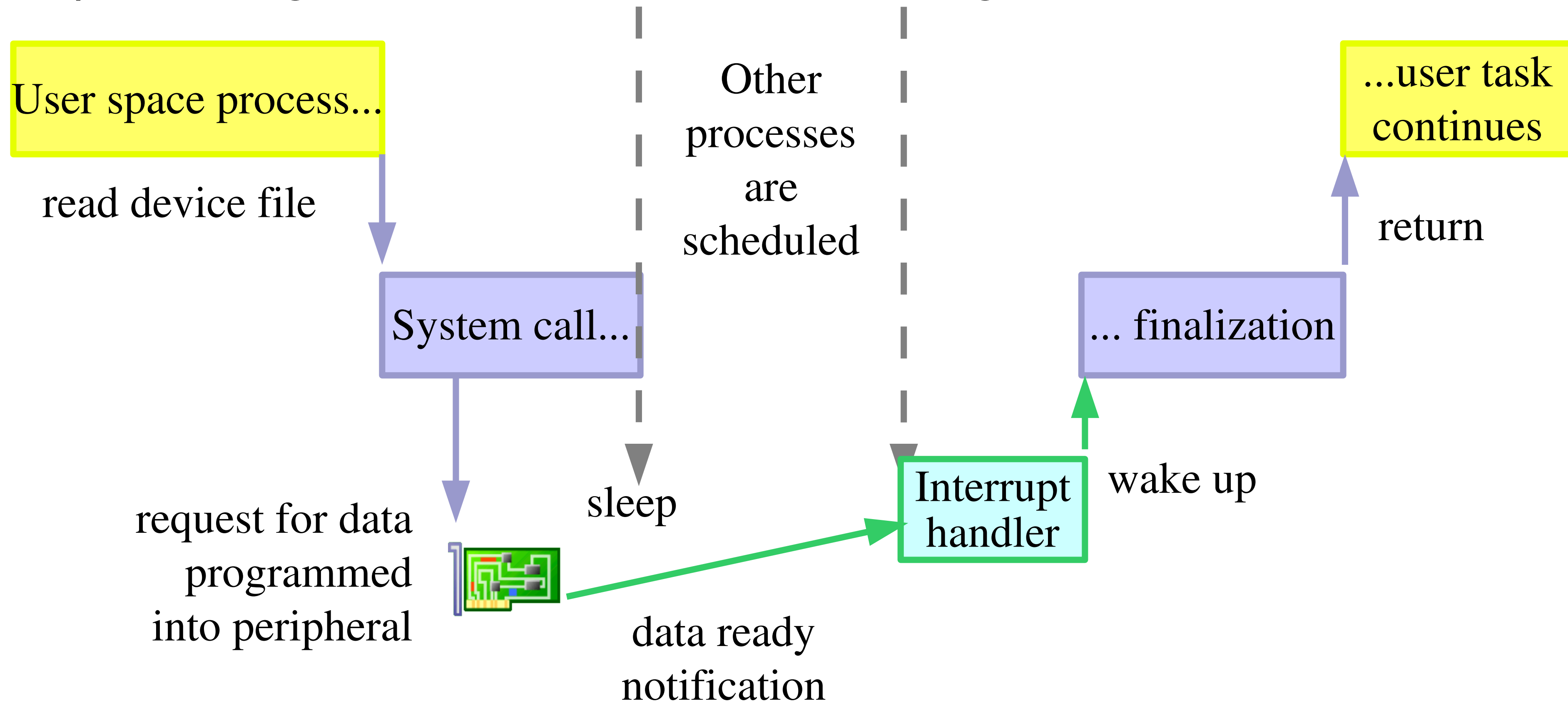
- Software cause evaluation (polled exception handling)
  - All exceptions/interrupts start same routine at same address – i.e. for MIPS that routine starts at 0x00000004 address
  - Routine reads source from status register (MIPS: cause registr)
- Vectored exception handling
  - CPU support hardware 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

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.)

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

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)