2

Hardware Fundamentals

Uwe R. Zimmer – International University Bremen

References for this chapter

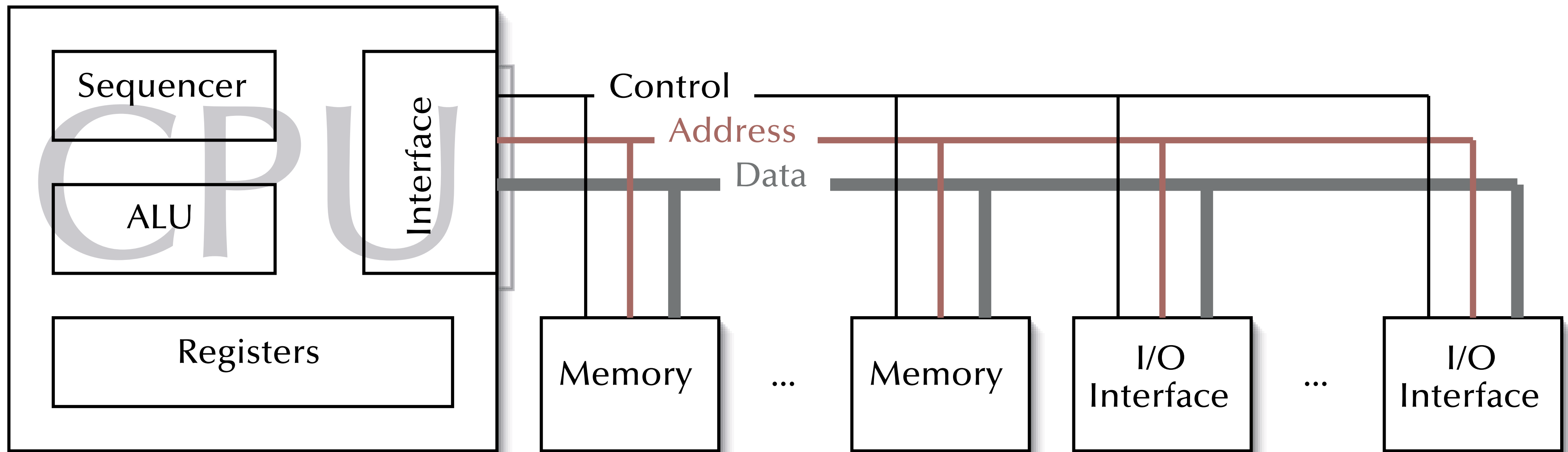
[Silberschatz01] – Chapter 2
Abraham Silberschatz, Peter Bear Galvin, Greg Gagne
Operating System Concepts
John Wiley & Sons, Inc., 2001

[Stallings2001] – Chapter 1
William Stallings
Operating Systems
Prentice Hall, 2001

all references and some links are available on the course page

Hardware Fundamentals

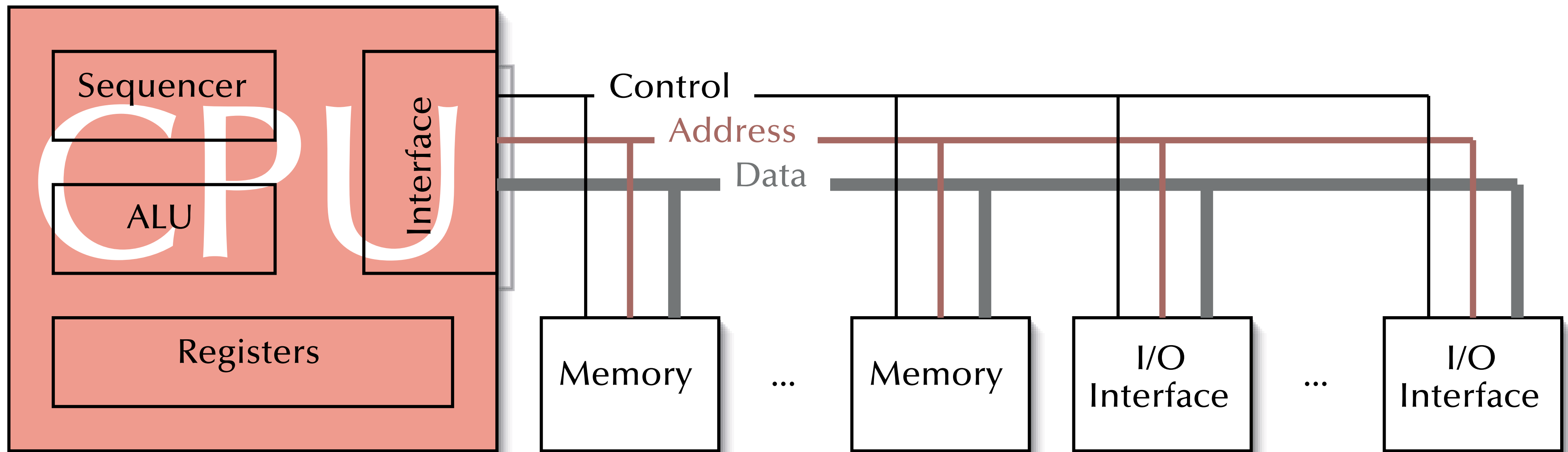
A common computer architecture:



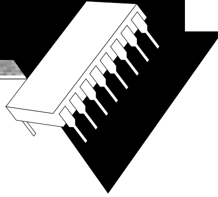
- Bus-systems carry device, address information and data (8-64 bit wide) as well as control lines in groups such as:
    - arbitration, synchronization, requests, interrupts, priorities

© 2003 Uwe R. Zimmer, International University Bremen

The CPU



- CPU components relevant for this course:
    - register-set, sequencer ('normal operation'), interrupt controller, protected modes

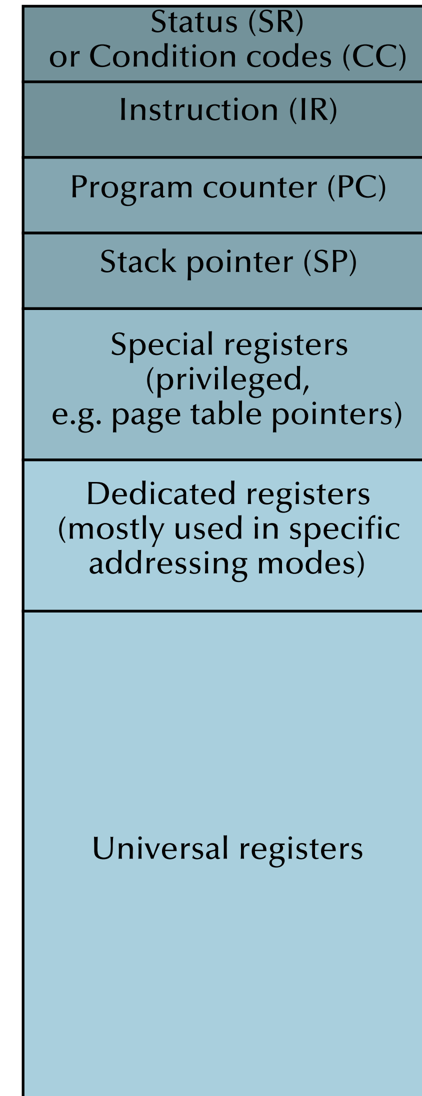


Hardware Fundamentals

Register set

- SR: Status / Condition codes (CC), e.g.:
  privilege level, interrupt level, result of last operation
- IR: current instruction
- PC: Address of current (next) instruction
- SP: Top of stack address
- Special privileged registers, e.g.:
  page table entries, memory protection maps
- Dedicated registers, e.g.:
  registers which can by employed in some contexts only
- Universal registers:
  registers, which can be employed for any purpose
  (addressing, storage, index, parameters, …)

Register structure



© 2003 Uwe R. Zimmer, International University Bremen

Hardware Fundamentals

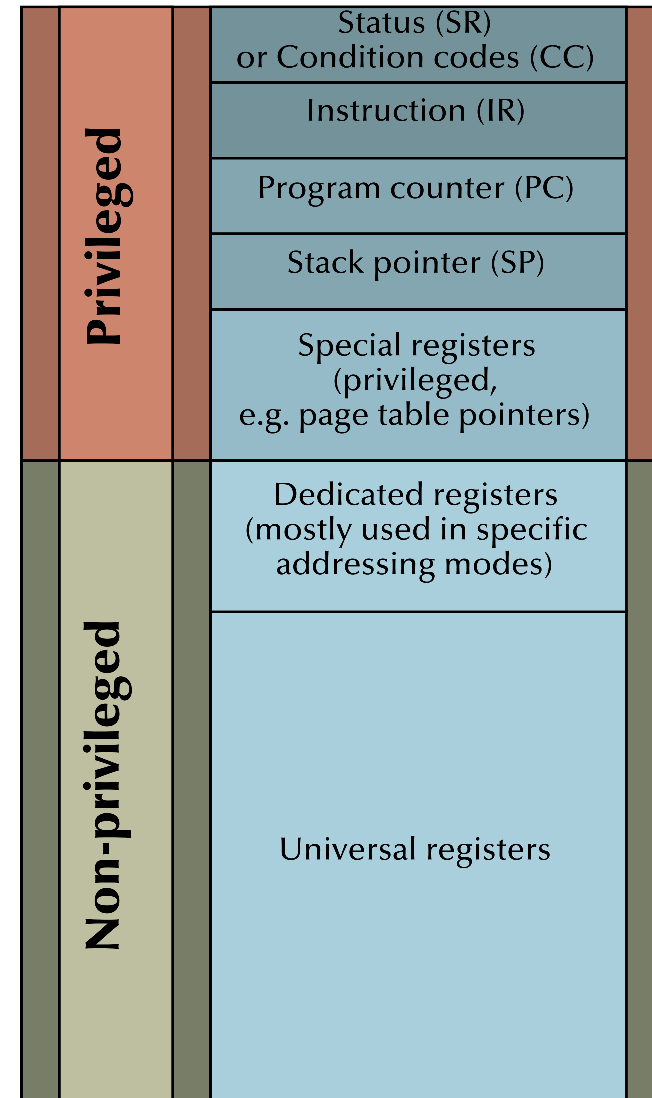
Register set

- Often divided into a privileged and non-privileged section
- Switch from non-privileged to privileged mode only via traps or interrupts (later in this chapter)

☞ SR, IR, PC, SP
+ some general registers (or at least one ‘accumulator’)
are found in all current processor designs

- Special and dedicated registers are not used in all architectures

Register structure



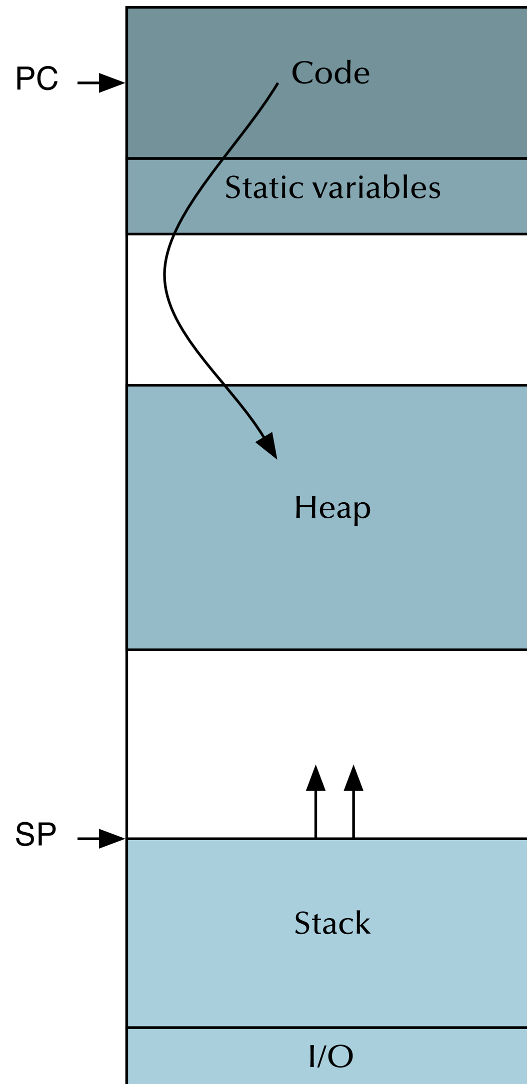
© 2003 Uwe R. Zimmer, International University Bremen

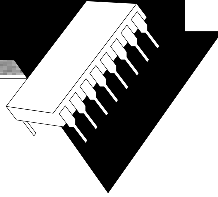
Main memory layout

Memory layout

- Classical usage of the RAM areas in most processors
- Main storage of data in
    - heap
    - stack
    - or local static

  depends on the usage of the programming language



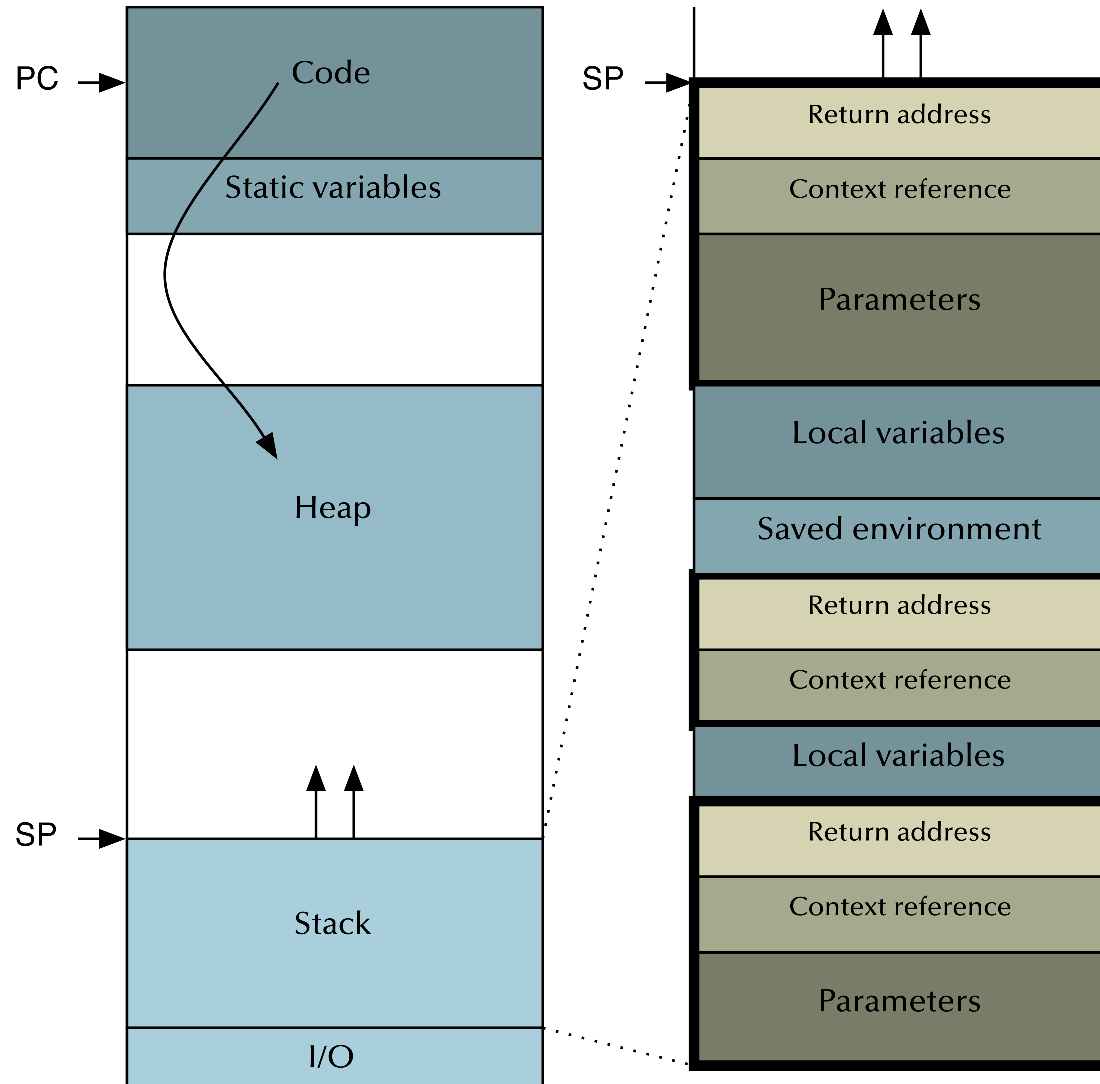


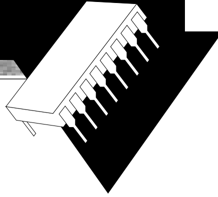
Hardware Fundamentals

Main memory layout

Stack frames

- Every sub-program call leaves an entry on the stack with all relevant information:
    - parameters
    - context (not in 'C')
    - return address
- Parameters may be removed by:
    - the calling routine ('C')
    - or the called routine
- Special architectures support faster parameter passing (e.g. register-bands)





Privileged instructions

Purpose:

- prevent user level tasks from by-passing the operating system
- restrict access form user-level tasks to resources, which are managed by the operating system:
    - Memory
    - I/O
    - Structures which are used to administer memory or I/O access (e.g. special registers, MMUs, etc.)

Implementation:

- declare some instructions privileged
- implement two (or more) protection levels in the CPU
- allow changes to a higher privilege level by means of traps/exceptions/interrupts only.

© 2003 Uwe R. Zimmer, International University Bremen

Interrupts

Required mechanisms for interrupt driven programming:

- **Interrupt control**: grouping, encoding, prioritising, and en-/disabling interrupt sources
- **Context switching**: mechanisms for cpu-state saving and restoring + task-switching
- **Interrupt identification**: Interrupt vectors, interrupt states

☞ hardware-supported

Interrupts

Interrupt control:

... at the individual device level

... at the system interrupt controller level

... at the operating system level
- beyond task-level (interrupt service routines)
- communicating interrupts to task
- transforming interrupts to signals

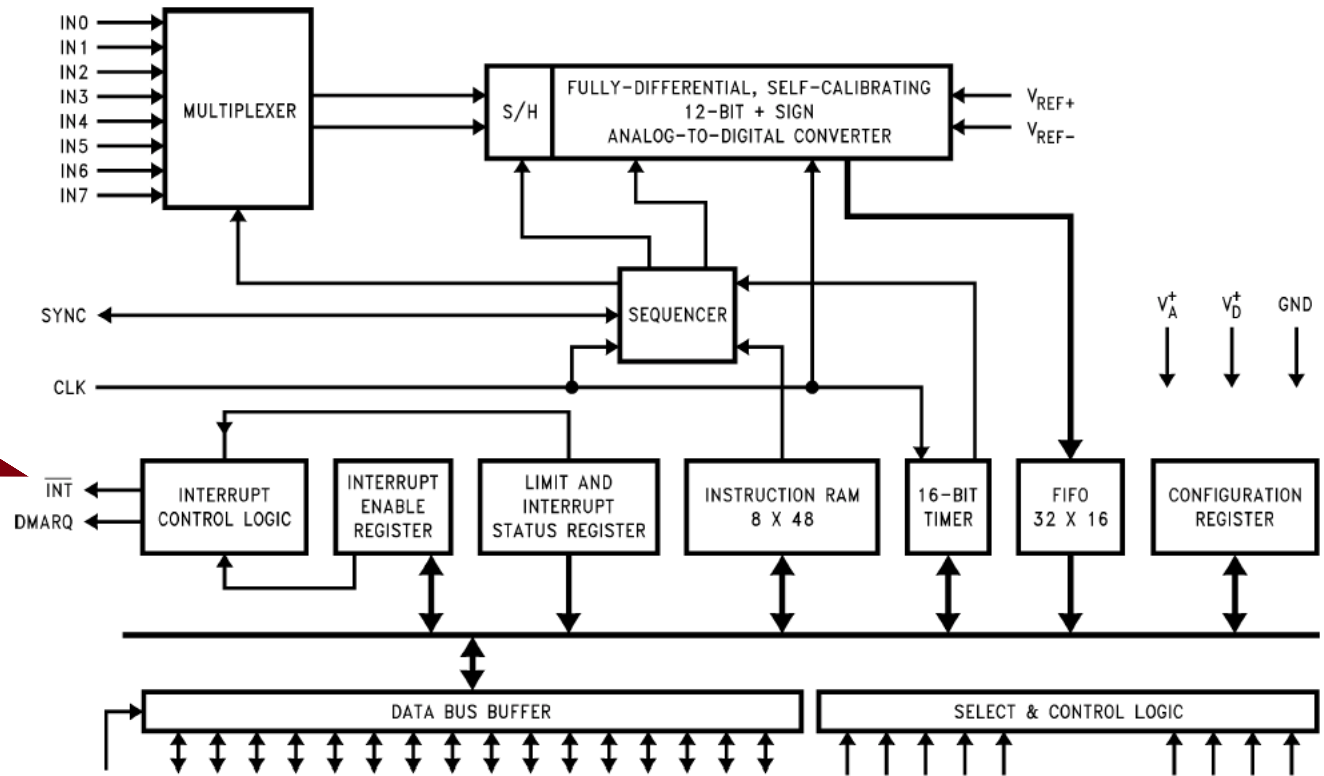
... at the language level

Interrupts

LM12L458

(National Semiconductor)

Interrupt signal



☞ only one interrupt signal line available!

☞ in order to identify the interrupt reason, an additional read cycle is required!

LM12L458

12-Bit + sign, 8 channel, A/D converter, controller and interface

Controller features:

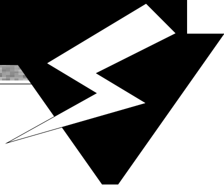
- Programmable acquisition times and conversion rates
- 32-word conversion FIFO
- Self-calibration and diagnostic mode
- 8- or 16-bit wide data bus microprocessor or DSP

Typ. applications:

- Data Logging
- Process Control

LM12L458 – accessible registers

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watch-dog	8/$\overline{12}$	Timer	Sync	V_{IN-}			V_{IN+}			Pause	Loop
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 01)	R/W	Don't Care						>/$\overline{<}$	Sign	Limit #1							
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 10)	R/W	Don't Care						>/$\overline{<}$	Sign	Limit #2							
1	0	0	0	Configuration Register	R/W	Don't Care				DIAG	Test = 0	RAM Pointer		I/O Sel	Auto $Zero_{ec}$	Chan Mask	Stand-by	Full CAL	Auto-Zero	Reset	Start
1	0	0	1	Interrupt Enable Register	R/W	Number of Conversions in Conversion FIFO to Generate INT2					Sequencer Address to Generate INT1			INT7	Don't Care	INT5	INT4	INT3	INT2	INT1	INT0
1	0	1	0	Interrupt Status Register	R	Actual Number of Conversion Results in Conversion FIFO					Address of Sequencer Instruction being Executed			INST7	"0"	INST5	INST4	INST3	INST2	INST1	INST0
1	0	1	1	Timer Register	R/W	Timer Preset High Byte								Timer Preset Low Byte							
1	1	0	0	Conversion FIFO	R	Address or Sign			Sign	Conversion Data: MSBs				Conversion Data: LSBs							
1	1	0	1	Limit Status Register	R	Limit #2: Status								Limit #1: Status							



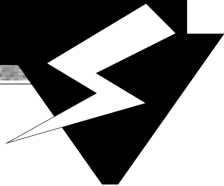
LM12L458 – *instruction RAM*

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watch-dog	$8/\overline{12}$	Timer	Sync	V_{IN-}			V_{IN+}			Pause	Loop
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 01)	R/W	Don't Care						$>/\overline{<}$	Sign	Limit #1							
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 10)	R/W	Don't Care						$>/\overline{<}$	Sign	Limit #2							

every entry in the **instruction RAM** consists of:

- **Loop** (1 bit): indicates the last instruction and branches to the first one.
- **Pause** (1 bit): halts the sequencer before this instruction.
- V_{IN+}, V_{IN-} (2*3 bit): select the input channels (000 selects ground in V_{IN-})
- **Sync** (1 bit): wait for an external sync. signal before this instruction.
- **Timer** (1 bit): wait for a preset 16-bit counter delay before this instruction.

© 2003 Uwe R. Zimmer, International University Bremen

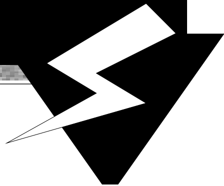


LM12L458 – *instruction RAM*

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watch-dog	$8/\overline{12}$	Timer	Sync	V_{IN-}			V_{IN+}			Pause	Loop
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 01)	R/W	Don't Care						$>/\overline{<}$	Sign	Limit #1							
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 10)	R/W	Don't Care						$>/\overline{<}$	Sign	Limit #2							

every entry in the **instruction RAM** consists of (cont.):

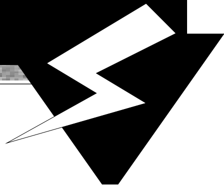
- **$8/\overline{12}$** (1 bit): selects the resolution (8 bit + sign or 12 bit + sign).
- **Watchdog** (1 bit): activates comparisons with two programmed limits.
- **Acquisition time** (D) (4 bit): the converter takes $9 + 2D$ cycles (12 bit mode) or $2 + 2D$ cycles (8 bit mode) to sample to input. Depends on the input resistance: $D \approx 0.45 \cdot R_S[k\Omega] \cdot f_{CLK}[MHz]$ for 12 bit conversions.
- **Limits** (including sign and comparator): used for Watchdog operation.



LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watch-dog	$8/\overline{12}$	Timer	Sync	V_{IN-}			V_{IN+}			Pause	Loop

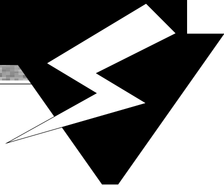
```
type ChannelPlus  is (Ch0, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7);
type ChannelMinus is (Gnd, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7);
type Resolutions  is (TwelveBit, EightBit);
type Aquisition_D is new Integer range 0..15; -- 9+2D (12bit), 2+2D (8bit)

for ChannelPlus  use (Ch0 => 0, Ch1 => 1, Ch2 => 2, Ch3 => 3,
                      Ch4 => 4, Ch5 => 5, Ch6 => 6, Ch7 => 7);
for ChannelMinus use (Gnd => 0, Ch1 => 1, Ch2 => 2, Ch3 => 3,
                      Ch4 => 4, Ch5 => 5, Ch6 => 6, Ch7 => 7);
for Resolutions  use (TwelveBit => 0, EightBit => 1);

type Instruction is record
       EndOfLoop, Pause, Sync, Timer, Watchdog : Boolean;
       Vplus                                   : ChannelPlus;
       Vminus                                  : ChannelMinus;
       Resolution                              : Resolutions;
       AquisitionTime                          : Aquisition_D;
    end record;
```



LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0 1	0 to 1	0 1	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watch-dog	8/$\overline{12}$	Timer	Sync	V_{IN-}			V_{IN+}			Pause	Loop

```
Units_Per_Word : constant Integer := Word_Size / Storage_Unit;

for Instruction use record
       EndOfLoop      at 0*Units_Per_Word range  0.. 0;
       Pause          at 0*Units_Per_Word range  1.. 1;
       Vplus          at 0*Units_Per_Word range  2.. 4;
       Vminus         at 0*Units_Per_Word range  5.. 7;
       Sync           at 0*Units_Per_Word range  8.. 8;
       Timer          at 0*Units_Per_Word range  9.. 9;
       Resolution     at 0*Units_Per_Word range 10..10;
       Watchdog       at 0*Units_Per_Word range 11..11;
       AquisitionTime at 0*Units_Per_Word range 12..15;
    end record;
```



LM12L458 – instruction RAM

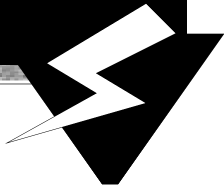
A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
	0	0	0	Instruction RAM	R/W	Acquisition				Watch-											
0		to		(RAM Pointer = 00)		Time				dog	8/$\overline{12}$	Timer	Sync		V_{IN-}			V_{IN+}		Pause	Loop
	1	1	1																		

```
for Instruction'Size      use 16; -- Bits
for Instruction'Alignment use 2;  -- Storage_Units (Bytes)
for Instruction'Bit_Order use High_Order_First;

type Instructions is array (0..7) of Instruction;
   pragma Pack (Instructions);

ADC_Instructions : Instructions;
for ADC_Instructions'Address use To_Address (16#0000132D#);
```

LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0 0 0 to 1 1 1			Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watch-dog	$8/\overline{12}$	Timer	Sync	V_{IN-}			V_{IN+}			Pause	Loop

```
ADC_Instructions (0) := (EndOfLoop       => False,
                         Pause           => False,
                         Vplus           => Ch0,
                         Vminus          => Gnd,
                         Sync            => True,
                         Timer           => False,
                         Resolution      => EightBit,
                         Watchdog        => False,
                         AquisitionTime  => 10);

ADC_Instructions (1) := (EndOfLoop       => True,  -- last instruction
                         Pause           => False,
                         Vplus           => Ch1,
                         Vminus          => Ch2,
                         Sync            => False,
                         Timer           => False,
                         Resolution      => TwelveBit,
                         Watchdog        => False,
                         AquisitionTime  => 0);
```

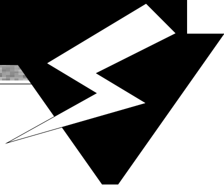


LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0 0 0 to 1 1 1			Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watch-dog	$8/\overline{12}$	Timer	Sync	V_{IN-}			V_{IN+}			Pause	Loop

Data structures in 'C':

```
enum ChannelPlus  {Ch0=0, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7};
enum ChannelMinus {Gnd=0, Ch1, Ch2, Ch3, Ch4, Ch5, Ch6, Ch7};
enum Resolutions  {TwelveBit=0, EightBit};

struct {
   unsigned int EndOfLoop       : 1;
   unsigned int Pause           : 1;
   ChannelPlus  Vplus           : 3;
   ChannelMinus Vminus          : 3;
   unsigned int Sync            : 1;
   unsigned int Timer           : 1;
   Resolutions  Resolution      : 1;
   unsigned int Watchdog        : 1;
   unsigned int AquisitionTime  : 4;
} Instruction;
```

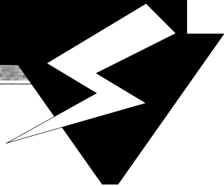


LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watch-dog	$8/\overline{12}$	Timer	Sync	V_{IN-}			V_{IN+}			Pause	Loop

Data structures in 'C':

```
struct {
   unsigned int EndOfLoop       : 1;
   unsigned int Pause           : 1;
   ChannelPlus  Vplus           : 3;
   ChannelMinus Vminus          : 3;
   unsigned int Sync            : 1;
   unsigned int Timer           : 1;
   Resolutions  Resolution      : 1;
   unsigned int Watchdog        : 1;
   unsigned int AquisitionTime : 4;
} Instruction;

Instruction    InstructionsA[8];
InstructionsA *Instructions;
Instructions = 0x0000132D;
```



LM12L458 – *instruction RAM*

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0 to 1	0 to 1	0 to 1	Instruction RAM (RAM Pointer = 00)	R/W	Acquisition Time				Watch-dog	$8/\overline{12}$	Timer	Sync	V_{IN-}			V_{IN+}			Pause	Loop

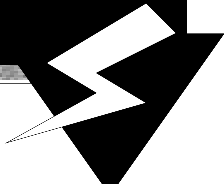
Data structures in 'C':

```
*Instructions (0).EndOfLoop       = 0;
*Instructions (0).Pause           = 0;
*Instructions (0).Vplus           = Ch0;
*Instructions (0).Vminus          = Gnd;
*Instructions (0).Sync            = 1;
*Instructions (0).Timer           = 0;
*Instructions (0).Resolution      = EightBit;
*Instructions (0).Watchdog        = 0;
*Instructions (0).AquisitionTime = 10;
```

don't!

If this works, you were lucky two times:

- The compiler implemented the struct-fields in the intended places and order.
- The bit ordering in your device is the way the compiler assumed it.



LM12L458 – instruction RAM

A4	A3	A2	A1	Purpose	Type	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
	0	0	0	Instruction RAM	R/W	Acquisition				Watch-											
0		to		(RAM Pointer = 00)		Time				dog	$8/\overline{12}$	Timer	Sync	V_{IN-}			V_{IN+}			Pause	Loop
	1	1	1																		

☞ *Macro-Assembler style programming:*

In order to produce portable code in 'C', it is necessary to set bits manually:

```
unsigned int setbits (unsigned int *r,
                      unsigned int n,          /* set n bits    */
                      unsigned int p,          /* at position p  */
                      unsigned int x)          /* to bitstring x */
{
   unsigned int mask;

   mask  = ~(~0 << n);
   *r   &= ~(mask << p);
   *r   |= (x & mask) << p;
   return (*r);
}
```

Interrupts

Interrupt control:

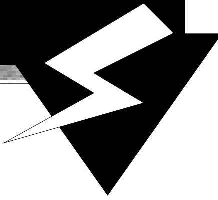
… at the individual device level

… at the system interrupt controller level

… at the operating system level

- beyond task-level (interrupt service routines)
- communicating interrupts to task
- transforming interrupts to signals

… at the language level



Interrupt service routines

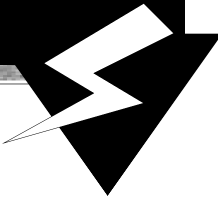
(available only in some OSs, e.g. VxWorks)

Purpose:

- Allow full access to the interrupt controller (interrupt vectors, priorities).
- Change to an interrupt service routine in a predictable amount of time.

☞ *Cannot* operate on the level of threads or tasks!

☞ Limitations regarding the accessibility of some OS-facilities (task level system calls).



Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Some VxWorks OS entries:

intConnect	Connect a routine to an interrupt vector
intLevelSet	Set the interrupt mask level
intLock	Disable interrupts (besides NMI)
intUnlock	Enable interrupts
intVecBaseSet	Set the interrupt vector base address
intVecBaseGet	Get the interrupt vector base address
intVecSet	Set an interrupt vector
intVecGet	Get an interrupt vector

these calls are employed by the language run-time environment or used directly from 'C'-code

Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Minimal hardware support (supplied by the cpu):

```
save essential CPU registers (IP, condition flags)
jump to the vectorized interrupt service routine
```

Minimal wrapper (supplied by the operating system):

```
save remaining CPU registers (or switch to another register set)
save stack-frame

--> execute user level interrupts service code

restore stack-frame
restore CPU registers (or switch back to the former register set)
restore IP
```

Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Interrupt service routine to task communication methods:

- **Shared memory and ring buffers**:
  most low level communication scheme (should be avoided)
- **Semaphore**: trigger a semaphore, where a task has been blocked before.
- **Monitors**:
  free a task, which is blocked at a monitor entry (standard Ada-method: protected object).
- **Message queues**: Send messages to a task (if queue is not full).
- **Pipes**: Write to a pipe (if pipe is not full).
- **Signals**: indicate an asynchronous task switch to the scheduler

Interrupt service routines

(available only in some OSs, e.g. VxWorks)

Interrupt service routine to task communication methods:

- **Shared memory and ring buffers**:
  most low level communication scheme (should be avoided)
- **Semaphore**: trigger a semaphore, where a task has been blocked before.
- **Monitors**:
  free a task, which is blocked at a monitor entry (standard Ada-method: protected object).
- **Message queues**: Send messages to a task (if queue is not full).
- **Pipes**: Write to a pipe (if pipe is not full).
- **Signals**: indicate an asynchronous task switch to the scheduler

☞ in all of the above: the interrupt service routines *cannot* block!

Interrupts ➪ 'Signals'

Interrupt control:

… at the individual device level

… at the system interrupt controller level

… at the operating system level

- beyond task-level (interrupt service routines)
- communicating interrupts to task
- transforming interrupts to signals

… at the language level

Interrupts ➪ 'Signals'

Some characteristics of signals:

- Involve a full task-switch operation

☞ Hard to predict timing behaviour

- Limited information about the interrupt-source
- Traditionally used to 'kill' processes
- Concept stems from a time before thread models,
  therefore the signal-to-thread propagation is implementation dependent and sometimes tricky.

Interrupts ➪ 'Signals'

Some common UNIX OS entries:

<table>
<tr><th>POSIX 1003.1b</th><th>BSD-UNIX</th><th></th></tr>
<tr><td>signal (…)</td><td>signal (…)</td><td>Specify the handler associated with a signal</td></tr>
<tr><td>sigaction (…)</td><td>sigvec (…)</td><td>Examine or set the signal handler for a signal</td></tr>
<tr><td>kill (…)</td><td>kill (…)</td><td>Send a signal (overwrite all other pending signals)</td></tr>
<tr><td>sigqueue (…)</td><td>N/A</td><td>Send a queued signal</td></tr>
<tr><td>sigsuspend (…)</td><td>pause (…)</td><td>Wait for a signal</td></tr>
<tr><td>sigwaitinfo (…)
sigtimedwait (…)</td><td></td><td>Wait for a signal, but do not involve the handler</td></tr>
<tr><td>sigemptyset (…)</td><td rowspan="2">sigsetmask (…)</td><td rowspan="2">Manipulate and
set the mask of blocked signals</td></tr>
<tr><td rowspan="2">sigprocmask (…)</td></tr>
<tr><td>sigblock (…)</td><td>Add to a set of blocked signals</td></tr>
</table>

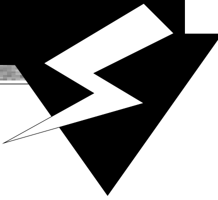
Interrupts ➪ 'Signals'

- Signals are originally process-level synchronization methods ('kill') and have been expanded to be used for everything from hardware-interrupts and timers to asynchronous task messaging.

☞ Signals are passed through a global task-scheduler.

☞ in many OSs: unpredictable 'work-arounds' for missing direct hardware interrupt propagation.

☞ make sure that you understand the attached strings in your OS, before employing any signals.



Interrupts

Interrupt control:

… at the individual device level

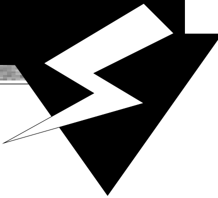
… at the system interrupt controller level

… at the operating system level

- beyond task-level (interrupt service routines)
- communicating interrupts to task
- transforming interrupts to signals

… at the language level

© 2003 Uwe R. Zimmer, International University Bremen



Exception/Trap/Interrupt indication

Four cases of modern exception indication:

raised: / from:	run-time environment	task
synchronously	run-time exceptions	exceptions or traps
asynchronously	interrupts / signals	asynchronous transfer of control

© 2003 Uwe R. Zimmer, International University Bremen

Exception/Trap/Interrupt indication

Ada95:

raised: / from:	run-time environment	task
synchronously	exceptions	
asynchronously	interrupt/signal handler	asynchronous transfer of control

Ada95: Interrupt handlers

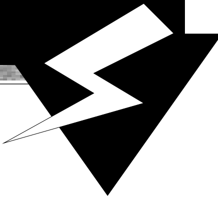
```
package Ada.Interrupts is
   type Interrupt_ID             is implementation-defined;
   type Parameterless_Handler is access protected procedure;

   function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
   function Is_Attached (Interrupt : Interrupt_ID) return Boolean;

   function  Current_Handler  (Interrupt : Interrupt_ID)
                                        return Parameterless_Handler;
   procedure Attach_Handler   (New_Handler : in  Parameterless_Handler;
                               Interrupt   : in  Interrupt_ID);
   procedure Exchange_Handler (Old_Handler : out Parameterless_Handler;
                               New_Handler : in  Parameterless_Handler;
                               Interrupt   : in  Interrupt_ID);
   procedure Detach_Handler   (Interrupt   : in  Interrupt_ID);

   function Reference (Interrupt : Interrupt_ID) return System.Address;

end Ada.Interrupts;
```

Ada95: Interrupt handlers

```
package Ada.Interrupts is
   type Interrupt_ID           is implementation-defined;
   type Parameterless_Handler is access protected procedure;

   function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
   function Is_Attached (Interrupt :

   function  Current_Handler   (Inter

   procedure Attach_Handler    (New_H
                                Inter
   procedure Exchange_Handler  (Old_H
                                New_H
                                Inter
   procedure Detach_Handler    (Inter

   function Reference (Interrupt : I

end Ada.Interrupts;
```

Protected procedures need to qualify as an interrupt handler:

1. use `pragma Interrupt_Handler`
2. let the compiler evaluate the suitability of the routine as an interrupt handler.



Ada95: Interrupt handlers

```
package Ada.Interrupts is
   type Interrupt_ID              is implementation-defined;
   type Parameterless_Handler is access protected procedure;
   function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
   function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
   function  Current_Handler  (Inter
   procedure Attach_Handler   (New_H
                               Inter
   procedure Exchange_Handler (Old_H
                               New_H
                               Inter
   procedure Detach_Handler   (Inter
   function Reference (Interrupt : I
end Ada.Interrupts;
```

Protected procedures can also be attached statically to an interrupt:

use `pragma Interrupt_Handler_Attach`

Ada95: Interrupt handlers

```
package Ada.Interrupts is
   type Interrupt_ID            is implementation-defined;
   type Parameterless_Handler is access protected procedure;

   function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
   function Is
   function  C
   procedure A
   procedure E
   procedure D
   function Re
end Ada.Interr
```

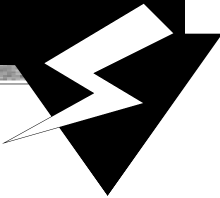
The mechanism to invoke an interrupt handler may be different from calling a protected procedure from a task.

Implementation advice: Whenever possible, the implementation should allow interrupt handlers to be called directly by the hardware.

Ada95: Interrupt handlers

```
package Ada.Interrupts is
   type Interrupt_ID              is implementation-defined;
   type Parameterless_Handler is access protected procedure;
   function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
   function Is_Attached (Interrupt : Interrupt_ID) return Boolean;
   function  C
   procedure A
   procedure E
   procedure Detach_Handler    (Interrupt   : in  Interrupt_ID);
   function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;
```

Metrics: *The implementation shall document the worst case overhead for an interrupt handler invocation (in clock cycles).*

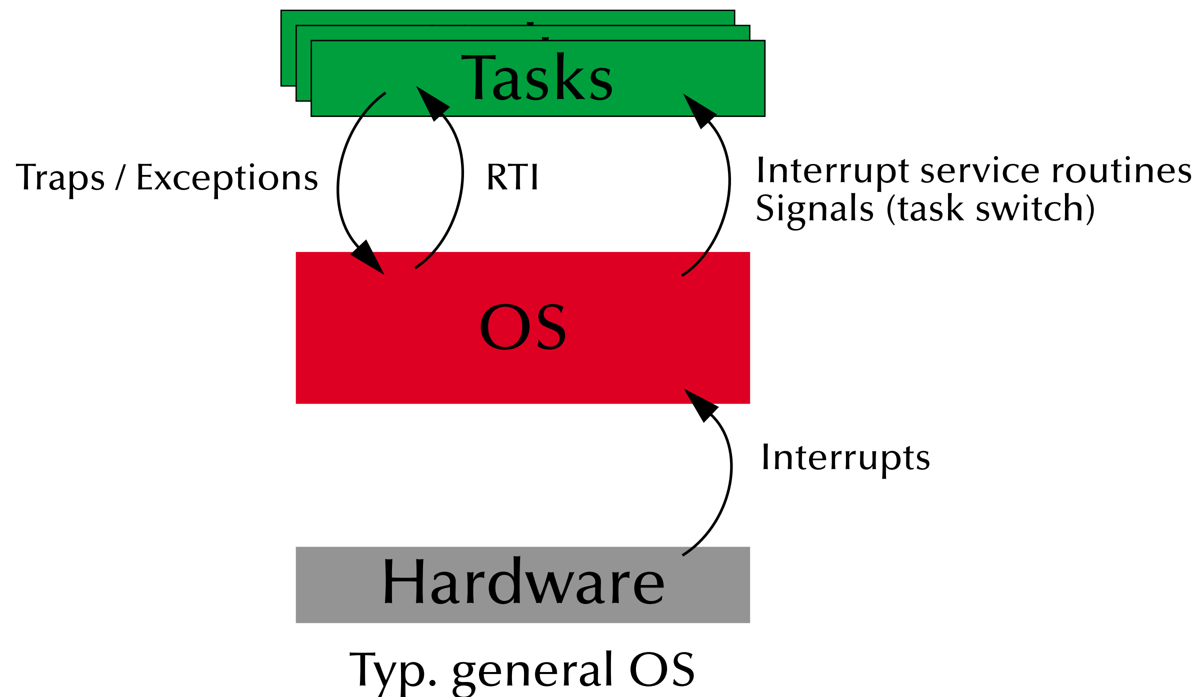


Ada95: Interrupt handlers

```
package Ada.Interrupts is
   type Interrupt_ID            is implementation-defined;
   type Parameterless_Handler is access protected procedure;

   function Is_Reserved (Interrupt : Interrupt_ID) return Boolean;
   function Is
   function  C
   procedure F
   procedure E
                                New_Handler : in   Parameterless_Handler;
                                Interrupt   : in   Interrupt_ID);
   procedure Detach_Handler    (Interrupt   : in   Interrupt_ID);

   function Reference (Interrupt : Interrupt_ID) return System.Address;
end Ada.Interrupts;
```

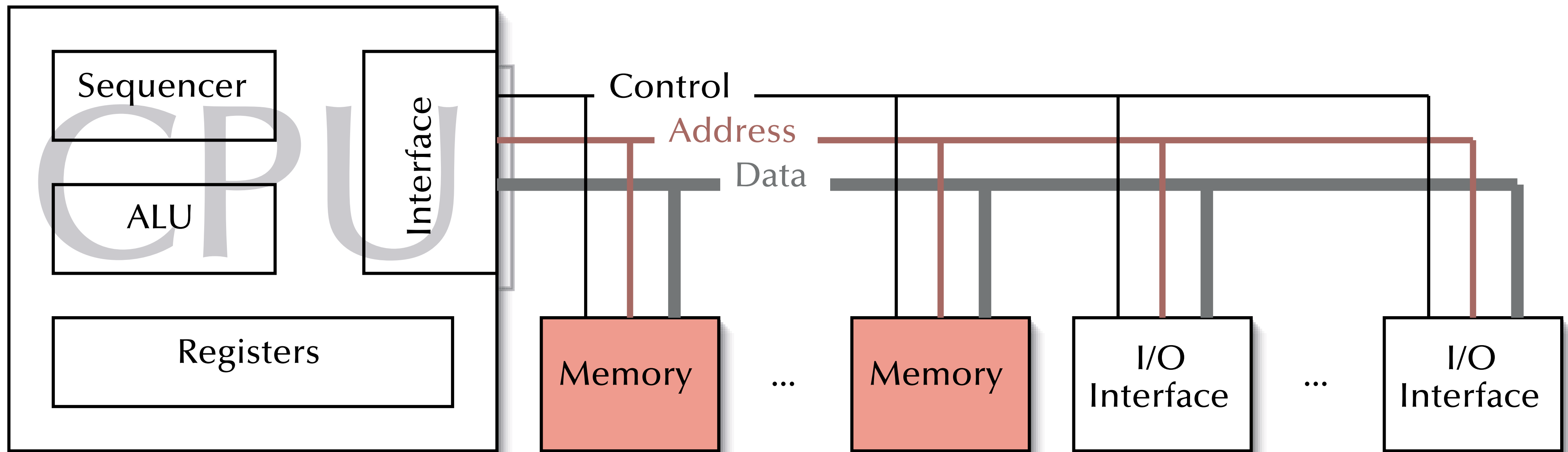
Direct access to the invocation address:
May be used to connect task-entries to interrupts
☞ *risky! — use with special care.*

3. A virtual machine, which is handling exceptions!

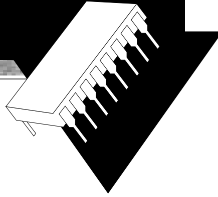


© 2003 Uwe R. Zimmer, International University Bremen

A common computer architecture:

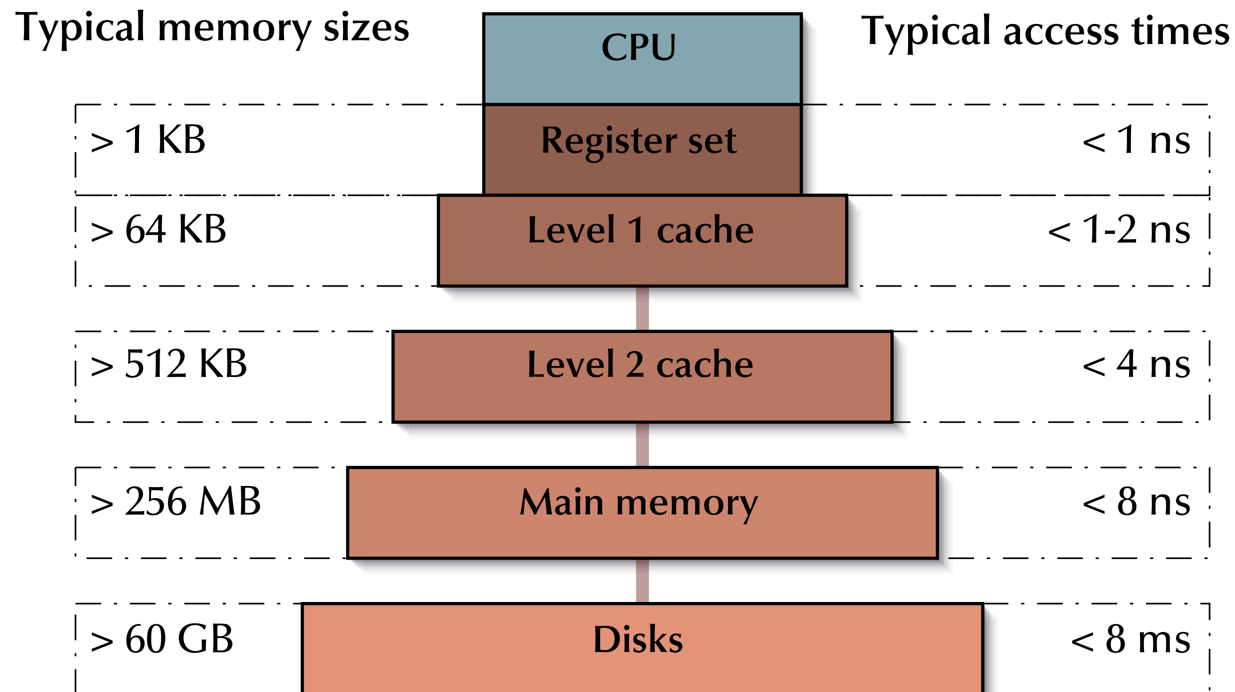


- Memory:
  - Hierarchy, Caching, Mapping

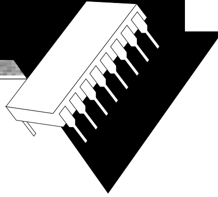


Memory sizes and access times: (typical workstation)

Basic memory hierarchy

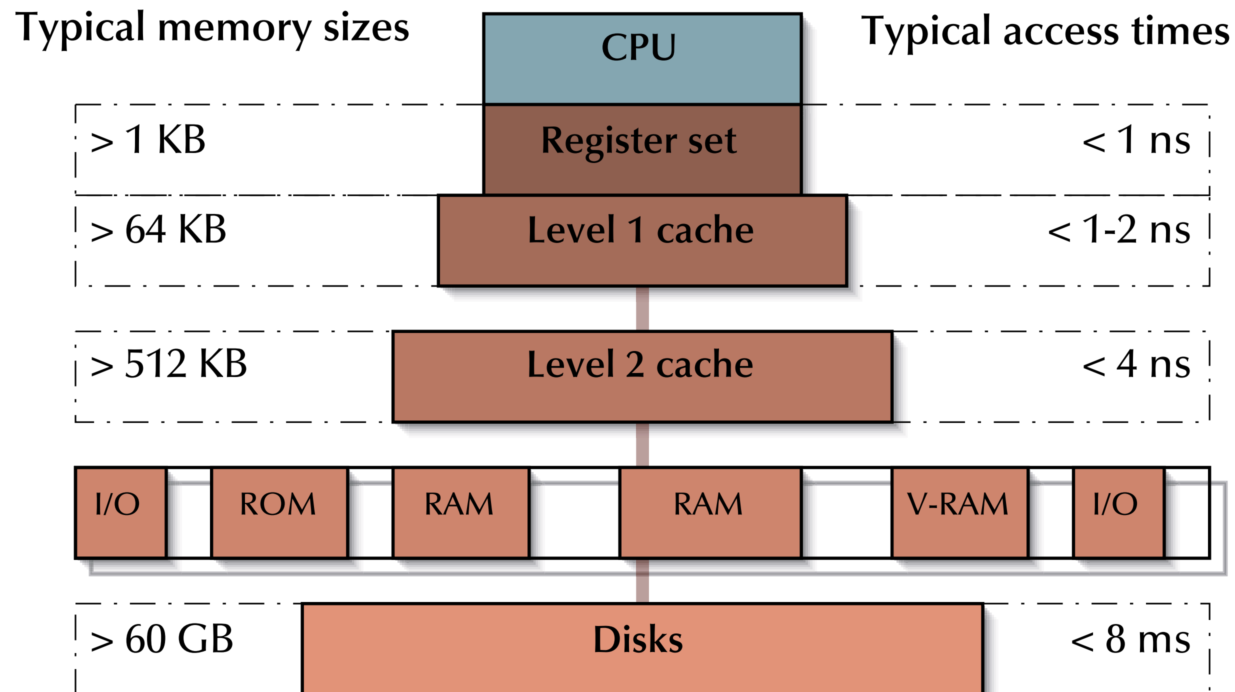


© 2003 Uwe R. Zimmer, International University Bremen



Main memory layout:

Basic memory hierarchy

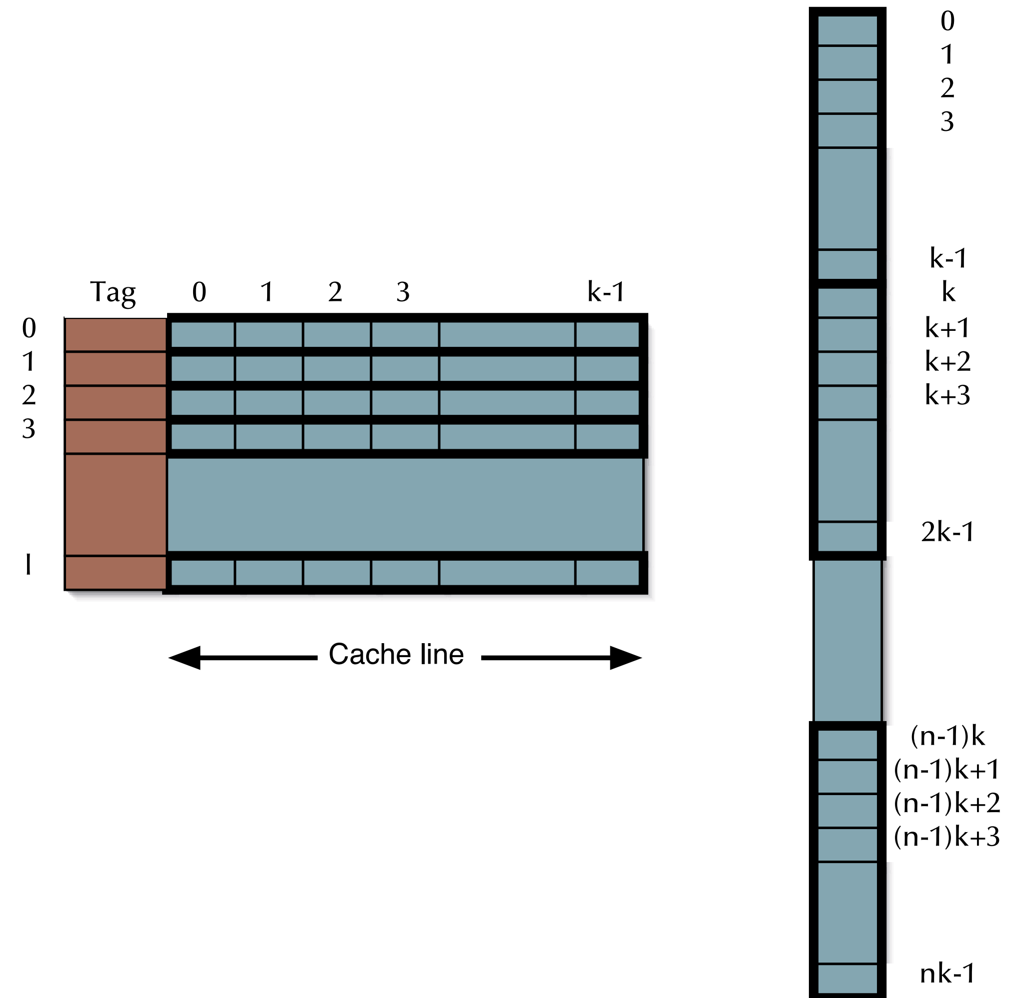


© 2003 Uwe R. Zimmer, International University Bremen

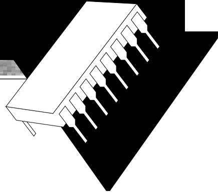
Caching

- Introduce a intermediate memory (cache), which is:
    - faster than the original memory
    - organized in 'cache lines'
    - addressed via tags and a fast matching hardware (e.g. associative memory)

Caché is actually French, meaning 'hidden', hence the cache memory is supposed to be 'invisible' to the user (the 'shadow memory').



© 2003 Uwe R. Zimmer, International University Bremen

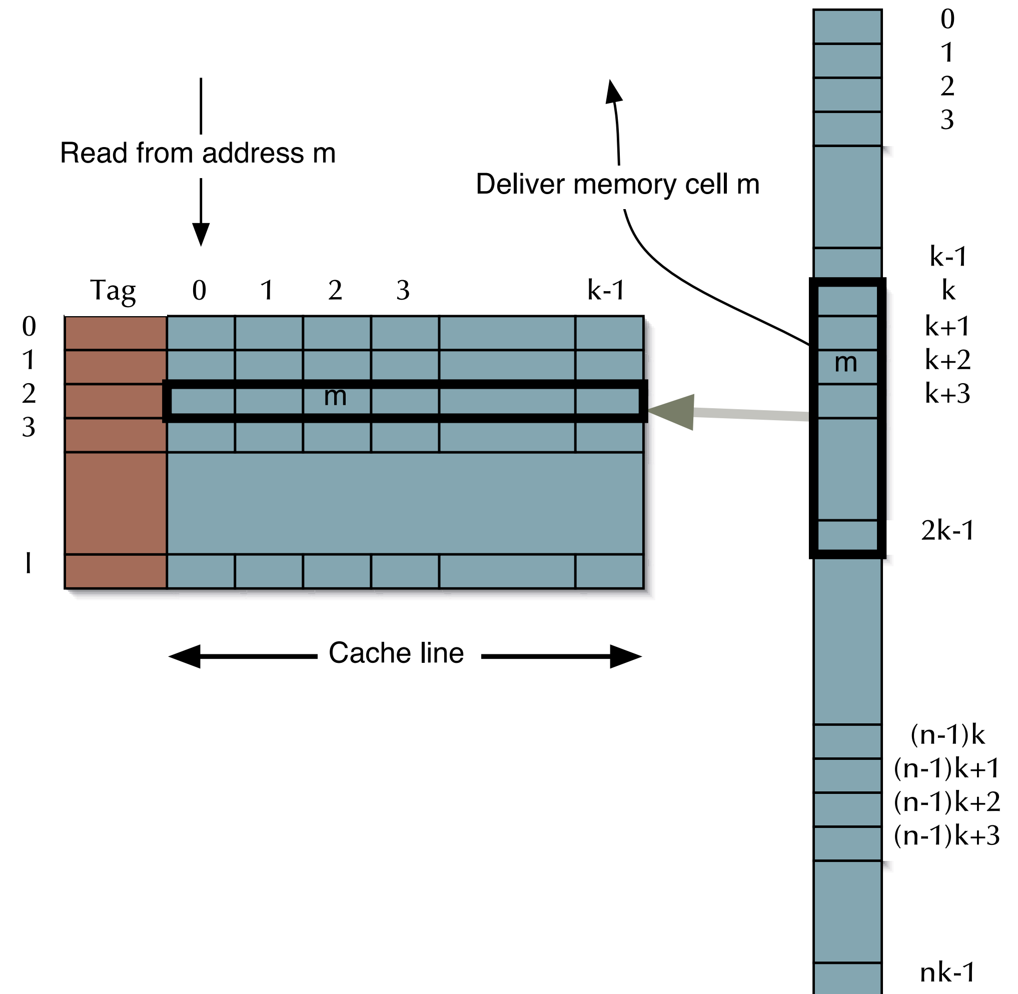


Cache misses

Memory read requests to cells, which are not currently stored in the cache, result in:

1. transfer of the full cache line into an empty of replaceable cache entry.
2. transfer of the data directly from the main memory to the requester.

Cache miss

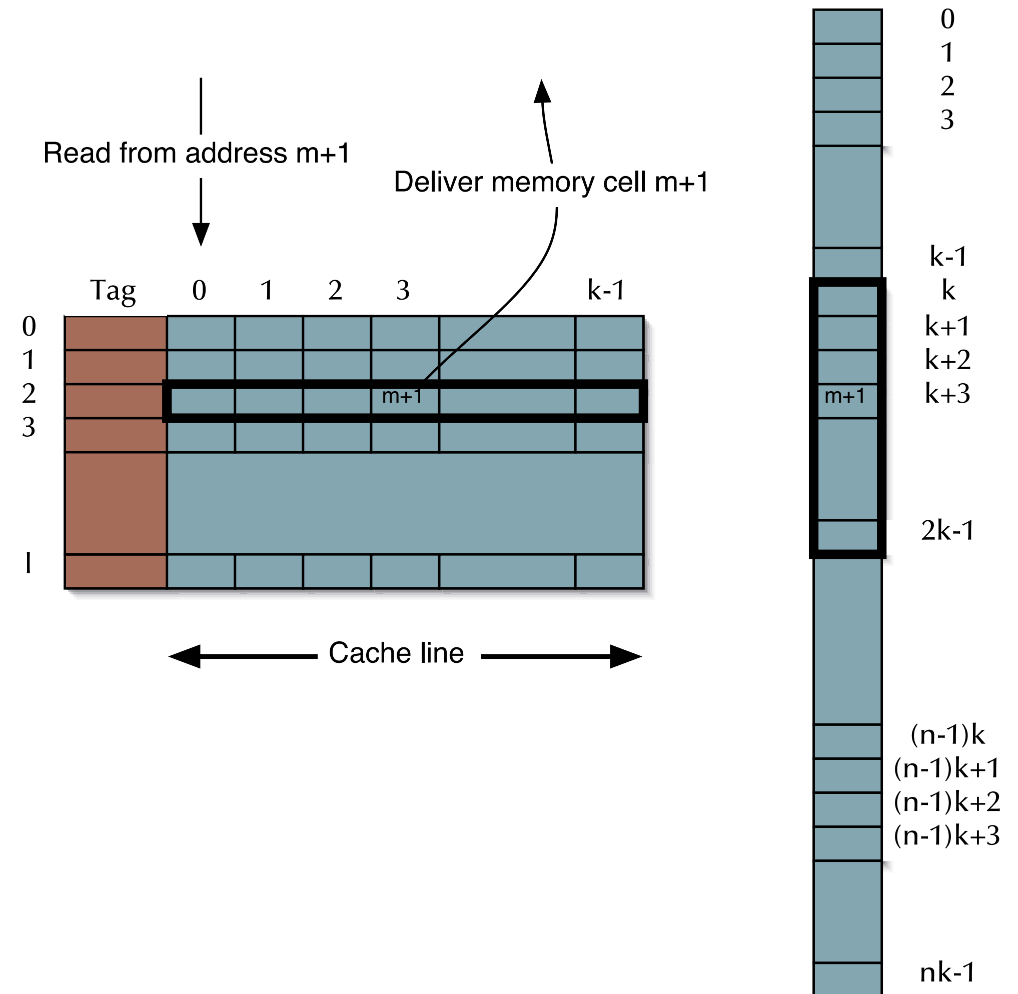


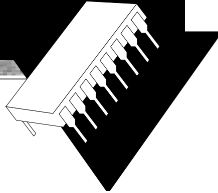
© 2003 Uwe R. Zimmer, International University Bremen

Cache hits

Memory read requests to cells, which are currently stored in the cache, result in:

- transfer of the requested data from the cache memory to the requester.
- *no access* to the main memory

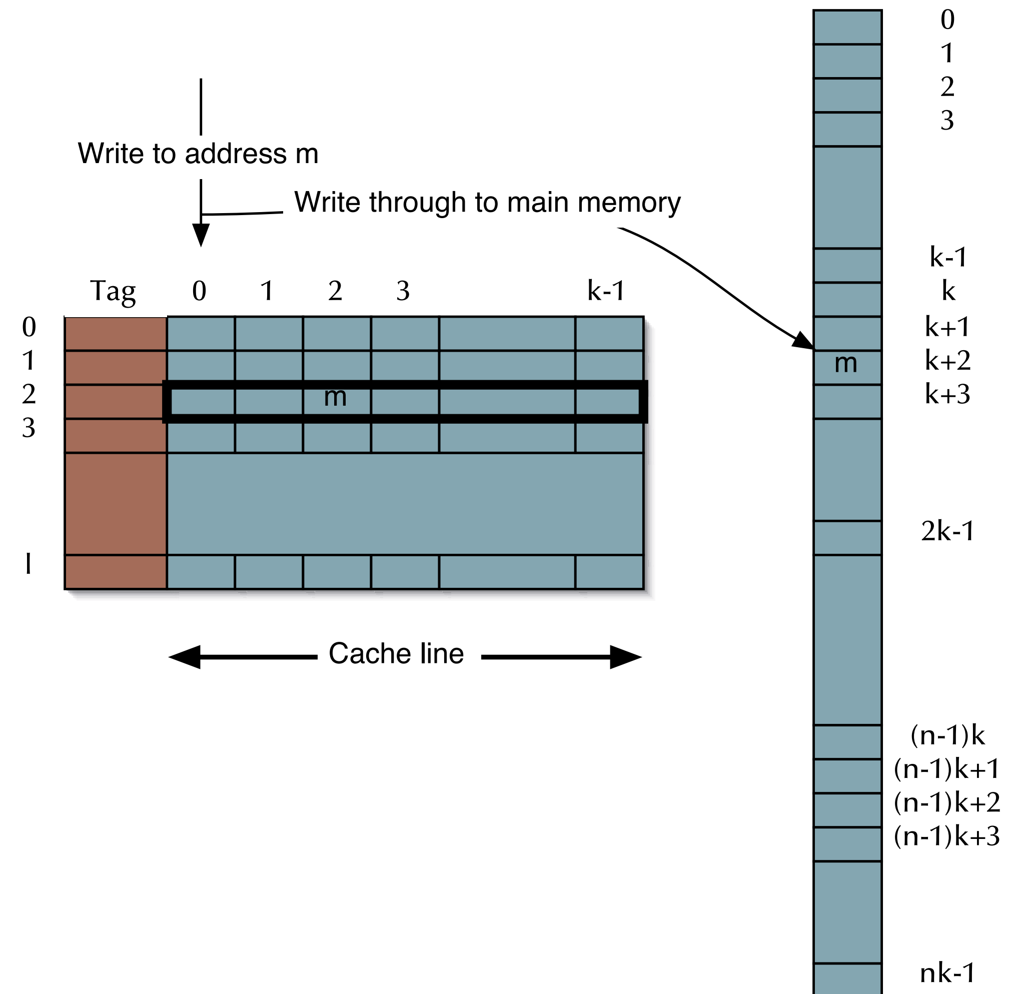


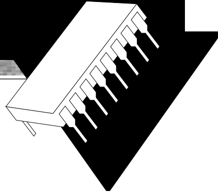


Cache write through

Write requests to cells, which are currently stored in the cache, result in:

1. update of the cache entry
2. update of the main memory cell





Hardware Fundamentals

Cache, delayed writes

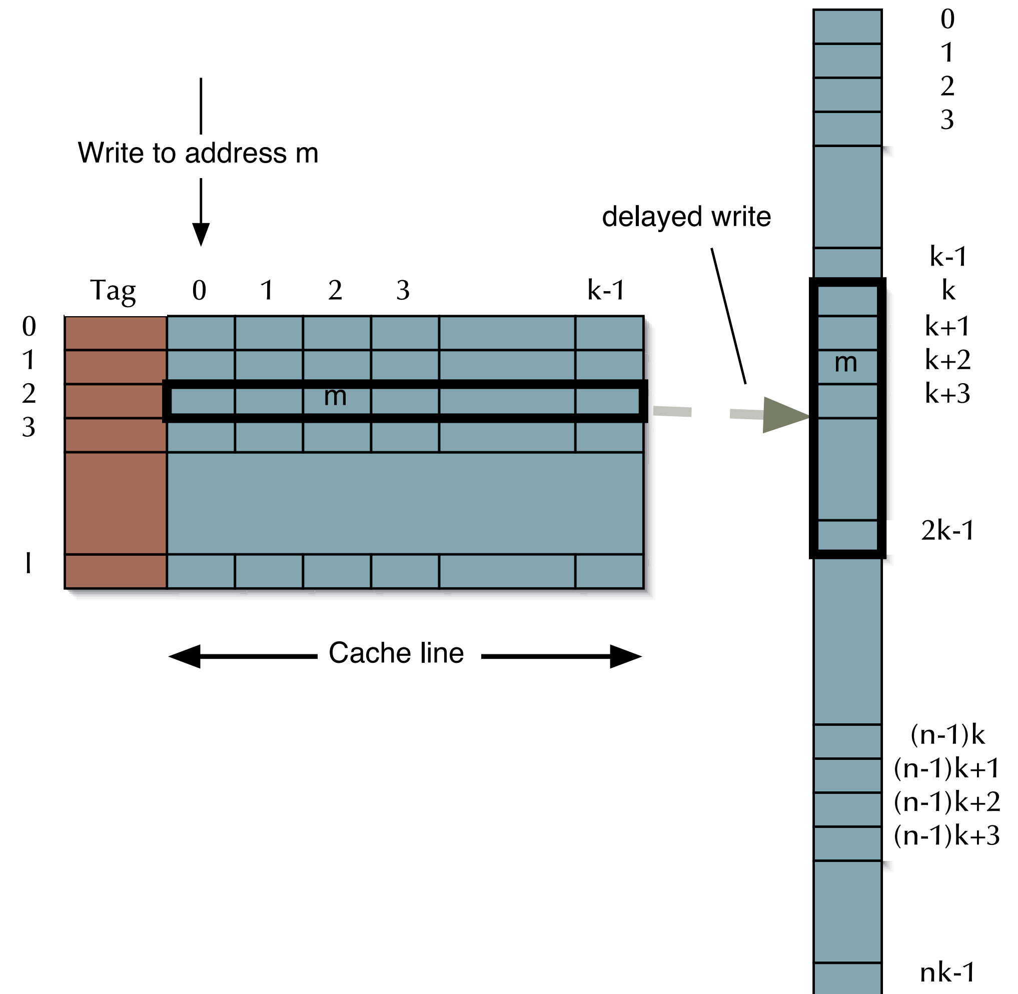
Write requests to cells, which are currently stored in the cache, result in:

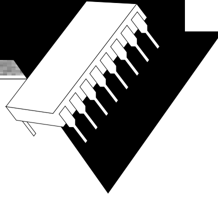
1. update of the cache entry

2. transfer of the full cache line (or the 'touched' entries) at a later point in time.

☞ **Critical in multi-processor / shared memory environments!**

Cache write (delayed)





Caching considerations

- Caches (two-level memories) are meant to maximize the **throughput –** not the predictability of a system.
- Cache performance is relying on:
    - **Spatial locality**: nearby memory cells are likely to be accessed soon
    - **Temporal locality**: recently addressed memory cells are likely to be accessed again soon

☞ The length of the cache lines are given by the relation between spatial and temporal locality

- According to some practical evaluations, the locality radius seems to be *independent* of the size of the main memory

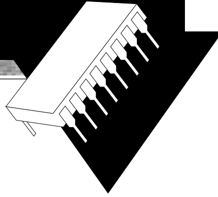
  ☞ thus there is an absolute maximum cache-size, beyond which the performance is no longer improving (memory caches of up to about 128KB are considered adequate in most cases).

More on memory locality

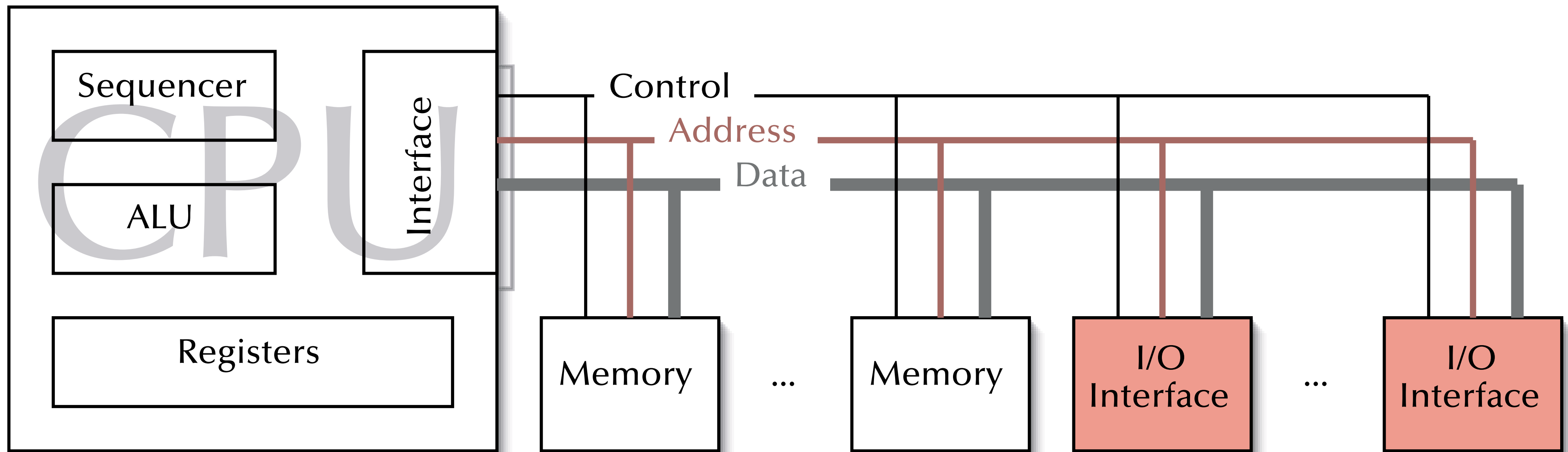
- *Imperative programming* will generate linear sequences of instructions mostly (☞ **spatial locality**).
- *Functional and declarative programming* turns out to generate more ‘jumpy’ code, but due to extensive usage of recursions it will show strong **temporal locality**.
- *Under all programming paradigms* CPU-time is often spent in relatively small loops/iterations (☞ **spatial & temporal locality**)
- Languages, which are using *explicit data structures* (like arrays and records) will store this data in a compact format (☞ **spatial locality**).

☞ The locality assumptions will thus be justified in the vast majority of all cases

… still it’s an heuristic.



A common computer architecture:



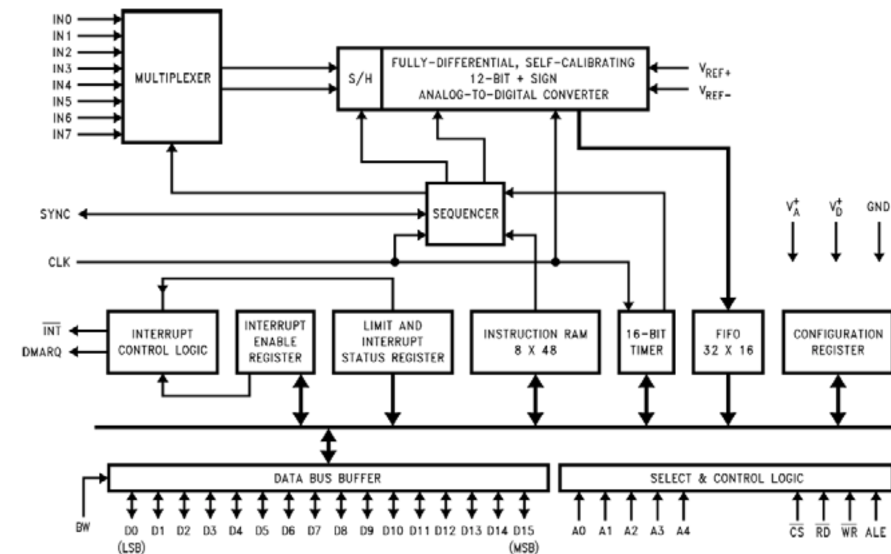
- I/O interfaces:
  - devices, controllers, communication with CPU, basic device programming

I/O devices

☞ the essential parts of a computer system,
which (may) make the computations meaningful.

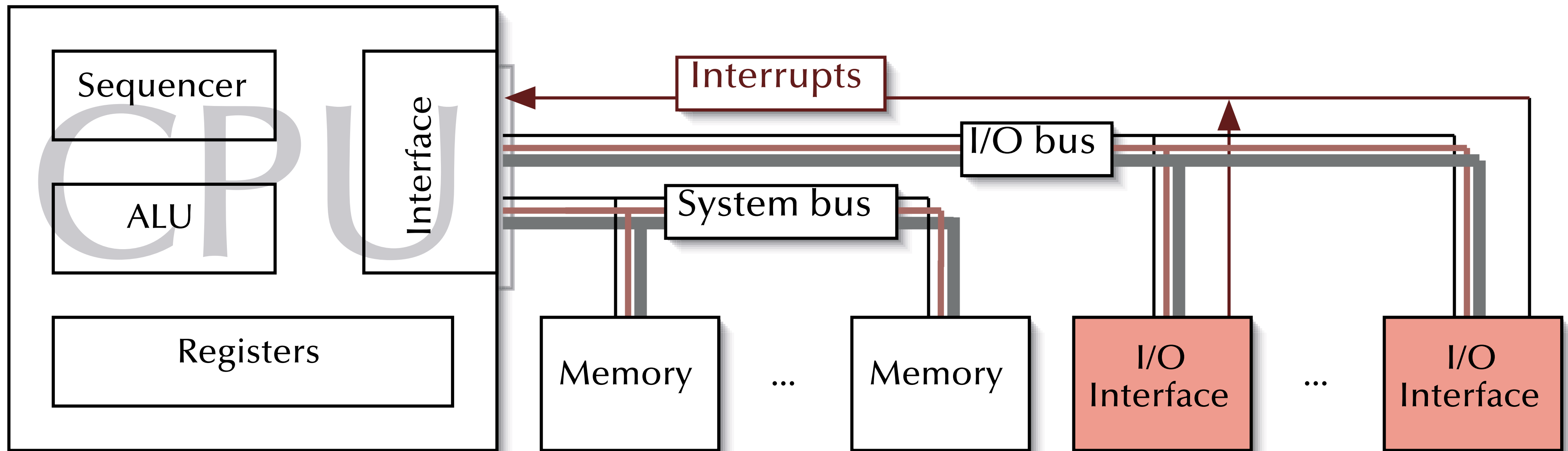
- Some typical classes of I/O devices:
    - clocks, timers
    - user-interface devices
    - document I/O devices (scanners, printers, …)
    - audio & video equipment
    - network interfaces
    - mass storage devices
    - all kinds of sensors and actuators in control applications

I/O controllers



- Interfacing between a local bus-system (system bus, peripheral bus) and an concrete hardware device
- Accessible from the CPU via control, status and data registers
- Major tasks:
    - convert electrical signals
    - buffer data in case of different signal speeds
    - multiplexing different channels
    - communicate with the external device independently of the CPU as far as possible
      ☞ often up to the level of a complete embedded µcontroller

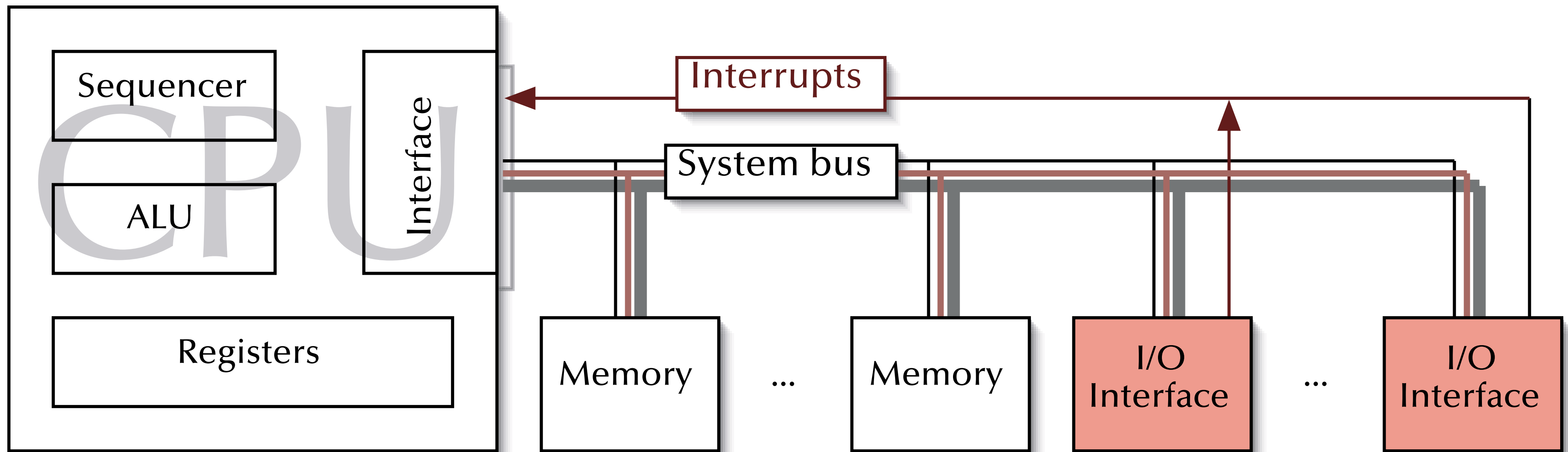
I/O interfaces via dedicated **I/O-buses**



- **I/O protection** is given by protected CPU instructions ☞ need to be done in protected mode.
- Potentially less efficient, since *all* I/O operations need to be done in the OS-kernel
  no obvious DMA - everything needs to be transferred via the CPU, I/O bus is processor specific

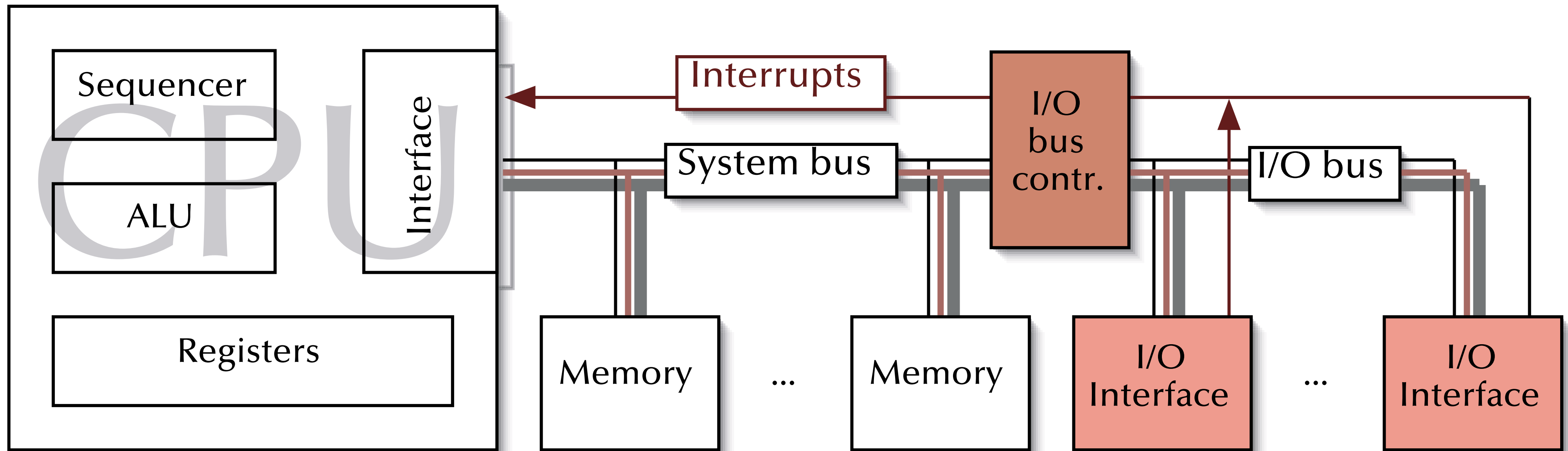
© 2003 Uwe R. Zimmer, International University Bremen

I/O interfaces via **system-bus**



- **I/O protection** requires / is identical with **memory protection,** DMA possibilities, expandible
- System bus can be a bottle-neck, I/O interfaces are processor dependent

I/O interfaces via **system-bus and I/O bus controller**



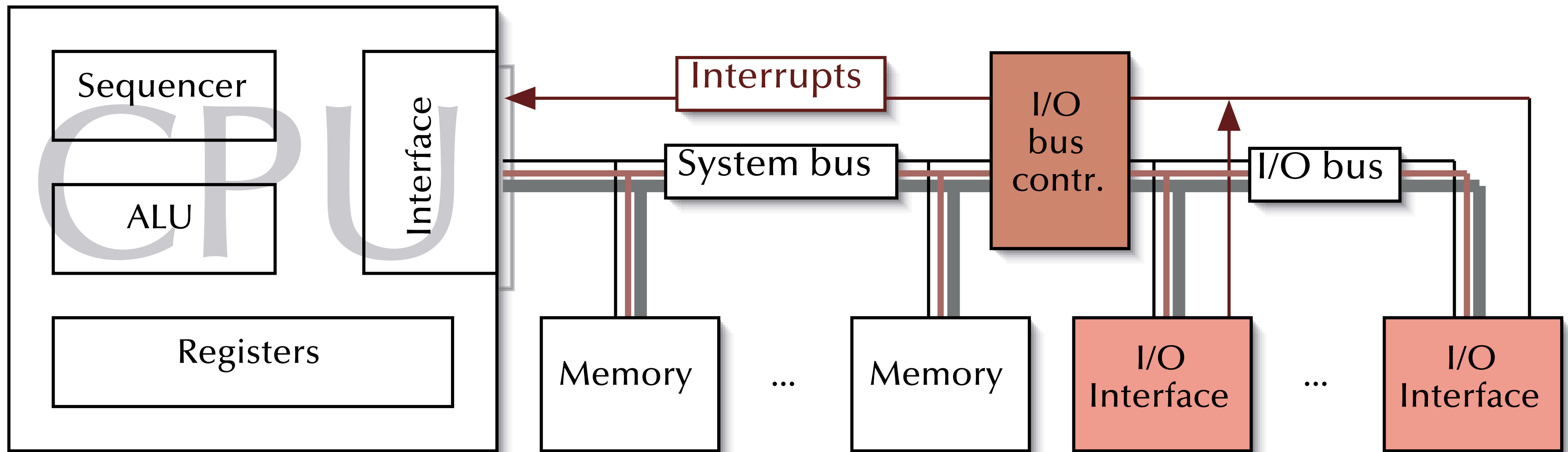
- **I/O protection** requires / is identical with **memory protection,** DMA possibilities, expandible
- System bus load can be reduced, I/O bus is platform independent, e.g. PCI, SCSI, …

Basic I/O device programming

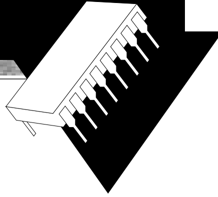
- **Status driven:** the computer polls for information (used in dedicated µcontrollers and pre-scheduled hard real-time environments)
- **Interrupt driven:** The data generating device may issue an interrupt when new data had been detected / converted or when internal buffers are full
  - **Program controlled:** The interrupts are handled by the CPU directly (by changing tasks, calling a procedure, raising an exception, free tasks on a semaphore, sending a message to a task, …)
  - **Program initiated:** The interrupts are handled by a DMA-controller. No processing is performed. Depending on the DMA setup, *cycle stealing* can occur and needs to be considered for the worst case computing times.
  - **Channel program controlled:** The interrupts are handled by a dedicated channel device. The data is transferred and processed. Optional memory-based communication with the CPU. ☞ the channel controller is usually itself a dedicated µengine / µcontroller.

Hardware Fundamentals

Concurrency is an intrinsic feature of real architectures!

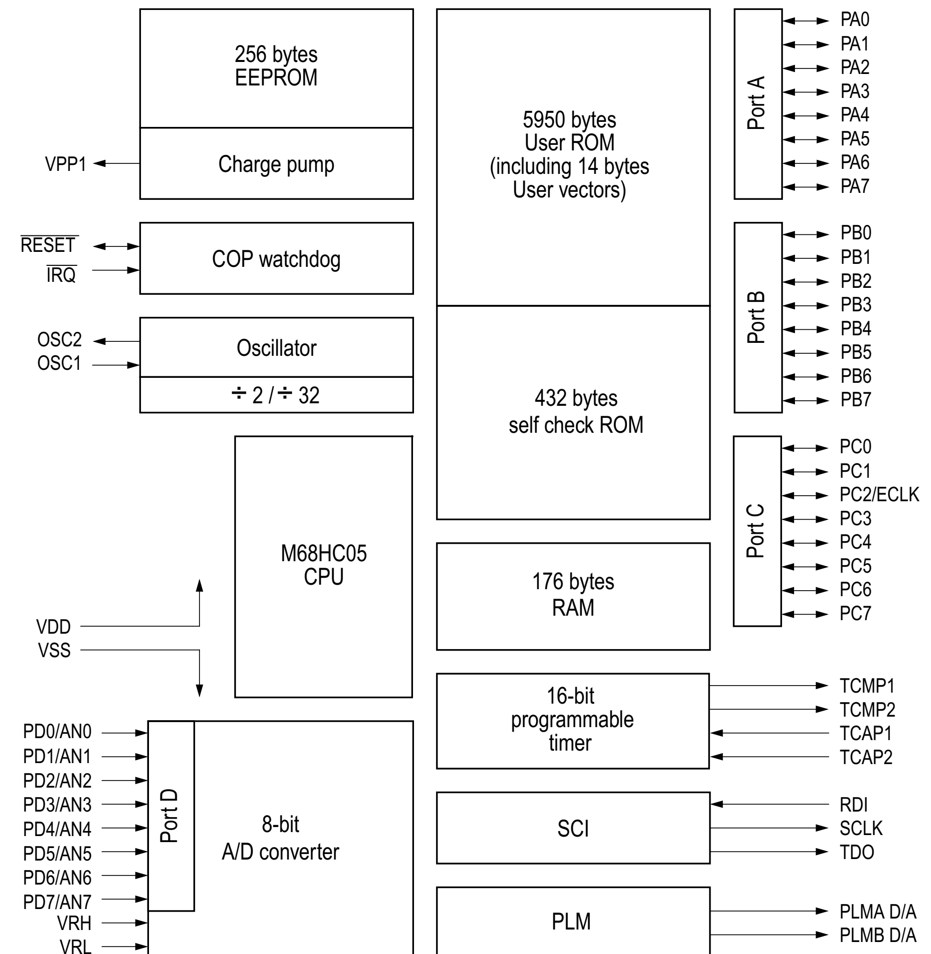


- ☞ Operating systems need to take care of all asynchronous and concurrent resources.
- ☞ ***Concurrency and synchronization are fundamentals of operating systems design!***

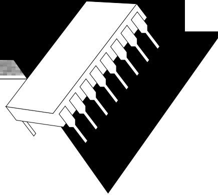


MC68HC05

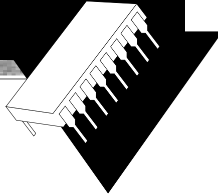
- **Clock**: max. 2.1 MHz internal (4.2 MHz external)
- **Registers**: PC, SP (16 bit); Accu, Index, CC (8 bit)
- **RAM**: 176 bytes
- **ROM**: 5936 bytes
- **EEPROM**: 256 bytes
- **Power saving modes** (stop, wait, slow)
- **Serial**: 46-76800 baud (at 2.4576 MHz)
- **Parallel** I/O: 3*8 bit; Parallel in: 1*8 bit
- **Timers**: 1*16 bit
- **A/D**: 8 channels, 8 bit
- **PWM**: 2 generators



© 2003 Uwe R. Zimmer, International University Bremen



```
MAIN    BRCLR  6,TSR,MAIN    ;Loop here till Output Compare flag set
        LDA    OCMP+1        ;Low byte of Output Compare register
        ADD    #$D4          ;Add Δt_l = (50ms/4µs)mod2^8 = $D4
        STA    TEMPA         ;Save till high half calculated
        LDA    OCMP          ;High byte of Output Compare register
        ADC    #$30          ;Add Δt_h = (50ms/4µs)div2^8 = $30 (+carry)
        STA    OCMP          ;Update high byte of Output Compare register
        LDA    TEMPA         ;Get low half of updated value
        STA    OCMP+1        ;Update low half and reset Output Compare flag
        LDA    TIC           ;Get current TIC value
        INCA                 ;TIC := TIC + 1
        STA    TIC           ;Update TIC
        CMP    #20           ;20th TIC?, 1 second passed?
        BLO    NOSEC         ;If not, skip next clear
        CLR    TIC           ;Clear TIC on 20th
NOSEC   EQU    *
        JSR    TIME          ;Update time-of-day & day-of-week
        JSR    KYPAD         ;Check/service keypad
        JSR    A2D           ;Check Temp Sensors
        JSR    HVAC          ;Update Heat/Air Cond Outputs
        JSR    LCD           ;Update LCD display
        BRA    MAIN          ;End of main loop
```



µControllers

JTAG

512 Kbytes Flash

512 Kbytes Flash

Burst Int.

U-BUS

USIU

E-BUS

PowerPC Core + FP

4-Kbyte CALRAM_B

4-Kbyte Overlay

READI

L2U I/F

L-BUS

32K CALRAM_A

28 Kbytes SRAM (No Overlay)

4-Kbyte Overlay

QADC64E w/AMUX

QADC64E w/AMUX

QSMCM

QSMCM

UIMB I/F

DLCMD2

IMB3

TPU3

6 Kbytes DPTRAM

TPU3

TPU3

4 Kbytes DPTRAM

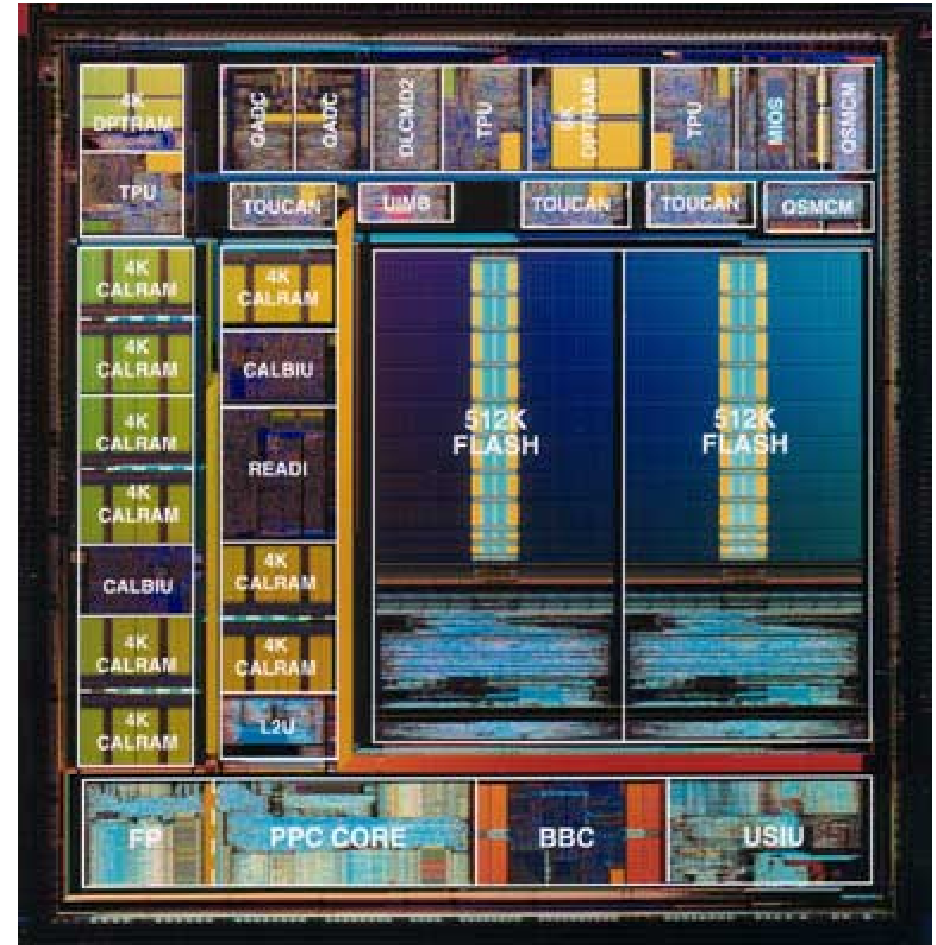
Tou CAN

Tou CAN

Tou CAN

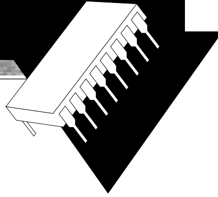
MIOS14

MPC565



µControllers MPC565

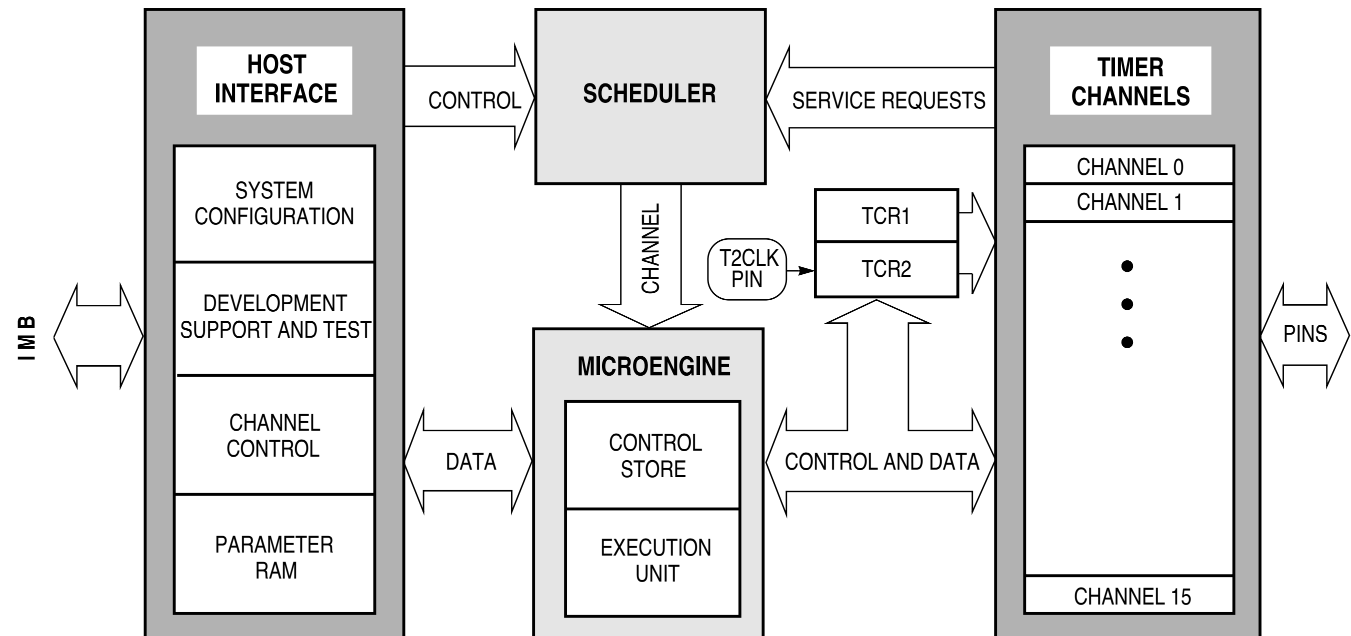
- -40° - +125°C, power dissipation: 0.8 - 1.12W
- **CPU**: PowerPC core (incl. FPU & BBC), 40/56MHz
- **Memory**: flash: 1M, static: 36K, 32 32-bit registers
- **Time processing units**: 3 (via dual-ported RAM)
- **Timers**: 22 channels (PWM & RTC supported)
- **A/D convertors**: 40 channels, 10bit, 250kHz
- **Can-bus**: 3 TOUCAN modules
- **Serial**: 2 interfaces
- **Interrupt controller**: 48 sources on 32 levels
- **Data link controller**:
  SAE J1850 class B communications module
- **Real-time embedded application development interface**: NEXUS debug port (IEEE-ISTO 5001-1999)
- **Packing**: 352/388 ball PBGA



Time processing unit

a special-purpose μcontroller:

- Independent μengine.
- 16 digital I/O channels with independent *match* and *capture* capabilities.
- Meant to operate these I/O channels for timing control purposes.
- Predefined μengine command set (ROM functions in control store).
- 2 16-bit time bases



Hardware Fundamentals

- **General computer architecture**
- **CPU**
    - Registers
    - Traps/Interrupts & protected modes
- **Memory**
    - General memory layout
    - Caching
- **I/O systems**
    - I/O controllers, I/O buses, device programming
- **Some examples of µprocessors**
    - Small scale µcontroller (68HC05)
    - Full scale integrated processor (MCP565)