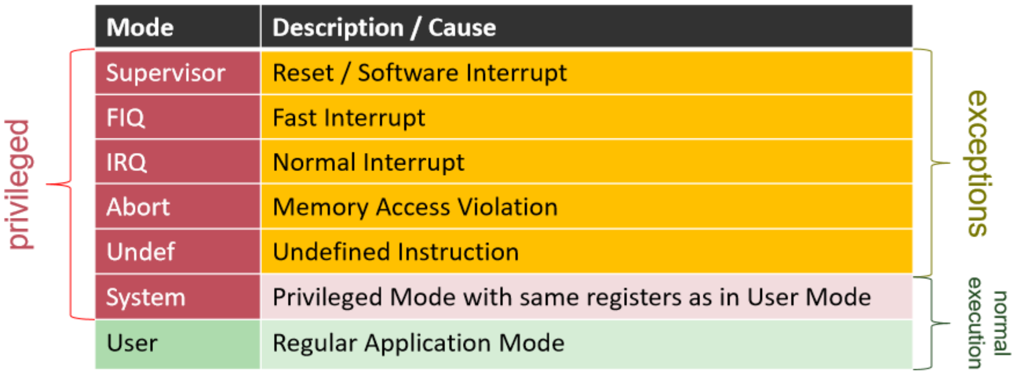
SYSTEM CONSTRUCTION

Case Study 1: Minos

ARM

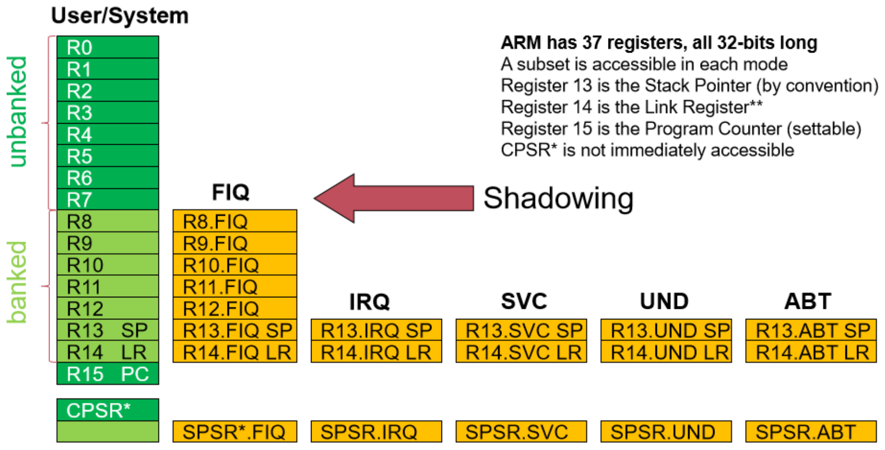
- 32-bit RISC instruction set, 3 operands
- ARM Architecture Version: high Level, defines instructions (see here)
- ARM Processor Family (Microarchitectures): Implementation but version doesn't match (see here)
- ARM Cortex Microarchitectures (Newest Implementation with different profiles)
- Manuals:
  - ARM Architecture Reference Manuals (describes Architecture e.g. build compiler)
  - ARM Technical System Reference Manuals (describes Implementation of ARM Arch.)
  - System on Chip Implementation Manuals (implementation of SoC e.g. address map of GPIO)
- Thumb Instruction Set: Subset of ARM Instruction Set (16-bit, 2 operands, better code density)
- ARM Processor Modes (see here)
  - Each mode has own stack and different subset of registers



Mode	Description / Cause
Supervisor	Reset / Software Interrupt
FIQ	Fast Interrupt
IRQ	Normal Interrupt
Abort	Memory Access Violation
Undef	Undefined Instruction
System	Privileged Mode with same registers as in User Mode
User	Regular Application Mode

privileged
exceptions
normal execution

- ARM Register Set (each mode gets own registers, some are shared)



ARM has 37 registers, all 32-bits long
A subset is accessible in each mode
Register 13 is the Stack Pointer (by convention)
Register 14 is the Link Register**
Register 15 is the Program Counter (settable)
CPSR* is not immediately accessible

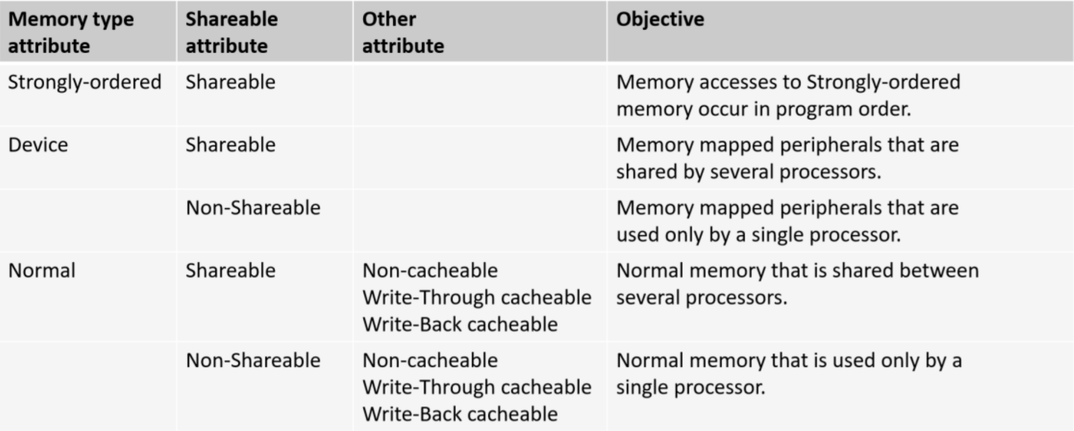
* current / saved processor status register, accessible via MSR / MRS instructions
** more than a convention: link register set as side effect of some instructions

- Processor Status Register (PSR): contains condition codes (e.g. negative result from ALU), one can disable IRQ and set processor mode

- Procedure Call:
    1. Caller: push param., bl #address (stores PC of next inst. into link register)
    2. Callee: save LR and old FP on stack, execute procedure, reset SP, restore FP and jump back to LR address
    3. Caller: clean up parameters from stack
- Raspberry Pi 2: ARMv7, 900 MHz, 1GB RAM
- ARM System Boot
    - ARM Processors usually starts executing code at address 0x0
    - RPI is booted from Video Core and RPI firmware does a lot of things beforehand (e.g. copies kernel-image to 0x8000)
- RPI 2 Memory Map:
    - Initially MMU is switched off
    - System memory is divided in ARM and VC part (partially shared e.g. frame buffer)

Cache

- ARM supports multiple levels of cache for each processor
- Cache Coherency
    - Multiple agents have cache between same memory
    - e.g. multiprocessor system, data and instruction cache of same CPU
    - Methods:
        - disable cache
        - for some memory regions (e.g. memory mapped devices) we can configure hardware supported mechanisms (e.g. strongly ordered memory not cached and memory barrier added)
        - manage cache coherence in software (e.g. special instructions, barriers, protocol to mediate several processors)
- Memory types in ARM

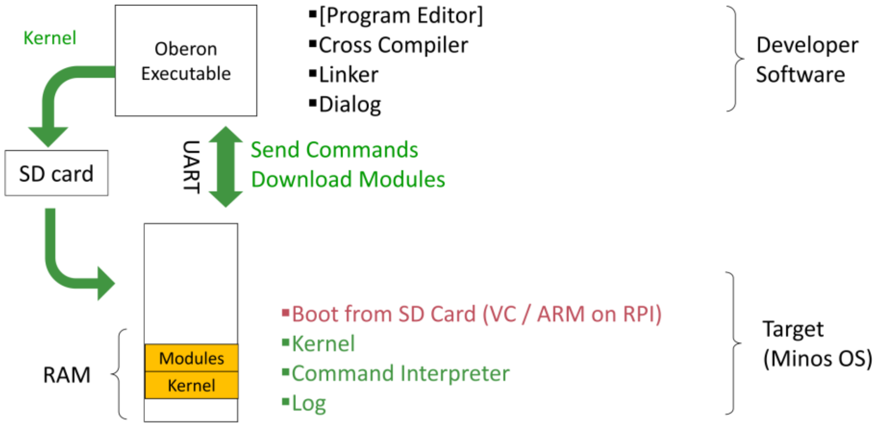


Memory type attribute	Shareable attribute	Other attribute	Objective
Strongly-ordered	Shareable		Memory accesses to Strongly-ordered memory occur in program order.
Device	Shareable		Memory mapped peripherals that are shared by several processors.
	Non-Shareable		Memory mapped peripherals that are used only by a single processor.
Normal	Shareable	Non-cacheable Write-Through cacheable Write-Back cacheable	Normal memory that is shared between several processors.
	Non-Shareable	Non-cacheable Write-Through cacheable Write-Back cacheable	Normal memory that is used only by a single processor.

- ARM cache properties
    - data memory cache: read/writes from observer happens always in order (even for different virtual addresses) -> no barriers necessary
    - instruction cache: guarantees from above doesn’t apply -> barriers necessary
    - changes in page table requires also cache maintenance
- Barriers
    - **ISB** (Instruction Synchronisation Barrier)
        - completes when all instructions before ISB have been completed (i.e. pipeline is flushed)
        - All inst. after ISB are fetch from cache/memory
        - for MMU reload, self-modifying code (module loading)

    - **DMB** (Data Memory Barrier)
        - All memory accesses before DMB are complete before memory accesses after DMB (but barrier doesn’t affect reordering of other instructions)
        - for Spinlocks, before writing to MMU control register
    - **DSB** (Data Synchronisation Barrier)
        - Stronger as DMB as no instruction after DSB is executed before all memory accesses have completed
        - when page tables changes, waking up processes waiting for mutex
    - Details

Cross Development



- [Program Editor]
- Cross Compiler
- Linker
- Dialog

- Boot from SD Card (VC / ARM on RPI)
- Kernel
- Command Interpreter
- Log

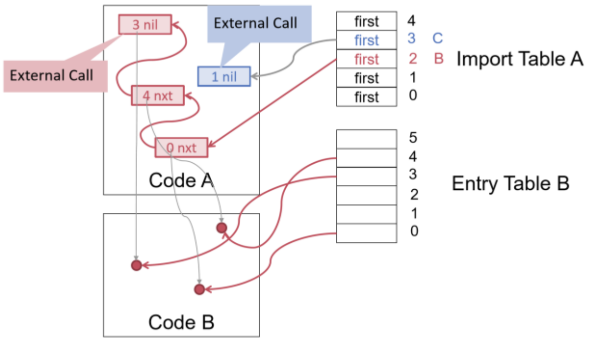
Oberon Language

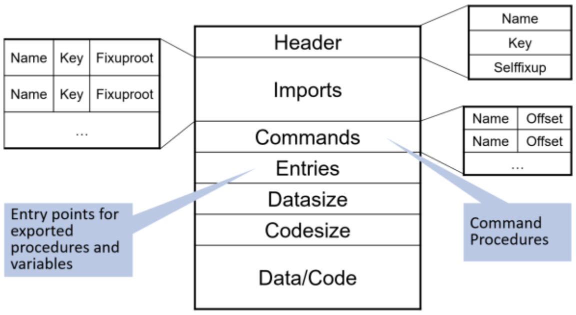
- Pascal family, strongly typed (static check compile time, runtime checks with type guards)
- There is no program in Oberon (Only modules which contain commands)
- Modules can be statically linked (form kernel) or dynamically linked (see here)
- Modules:
    - module body is executed only once when the module is loaded
    - * exports a procedure (no param. can act as a command)
- Module loading:
    - Statically linked modules are loaded on start-up
    - Modules are loaded on demand
    - Modification only visible after reloading
    - Unloading not possible if module still in use or statically linked
- Module SYSTEM allows unsafe operations for system programming
    - Direct memory and register access (e.g. Link Register, Stack Pointer)
    - Special flags are available to mark procedure e.g. as interrupt handler
    - SET type for bit manipulation, inline-assembly, unsafe pointers
- Supports inheritance, type test and type guards

Linking

- See Project Oberon Kap. 6
- Linking: Map relative addresses of symbols into absolute addresses
- Loading: Map the absolute addresses into memory
- Normally linking takes a lot of time so an object file is pre-linked with absolute addresses and a loader places these addresses into memory. In Oberon both steps are combined which allows dynamic loading of existing modules.
- Dynamic linking:
    - Defer linking of some symbols just before executing a program
    - Allow usage of shared libraries (e.g. save space, fix bugs without relinking)

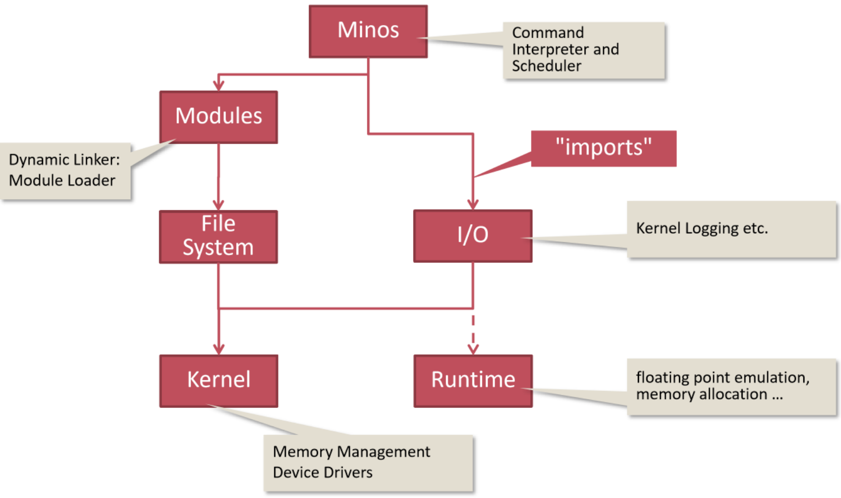
- Static linking:
  - Copy library procedures in executable
  - More portable as library don’t have to be present on system
- Compilation:
  - Compiler generates symbol file (interface) and object file (code&data)
  - Each object file has a fingerprint to detect changes
  - If module B imports module A fingerprint of A is stored in object file of B
- Linking process:
  - Link A = Link B, Link C, Fixup A, Execute module body
  - Each module provides a table with addresses to its exported elements
  - Each module has also a table with its imports. Each entry is the beginning of a fixup chain for each imported module
  - on module loading we have to fixup A i.e. we have to go through the import table and follow for each module the fixup chain. Each node of the fixup chain has an entry which is pointing to the entry table of the respective module (entry table contains addresses of exported elements)
  - with this approach we have to fixup fewer addresses which is faster

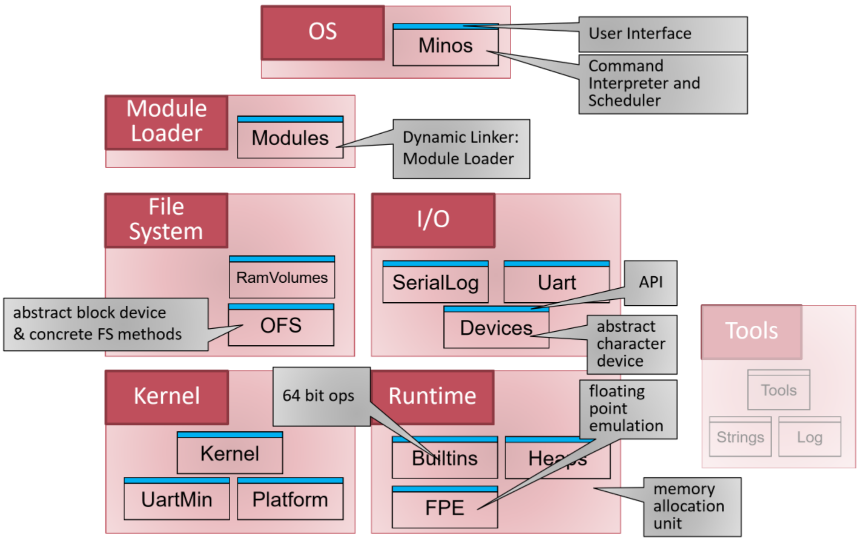




MINOS Kernel

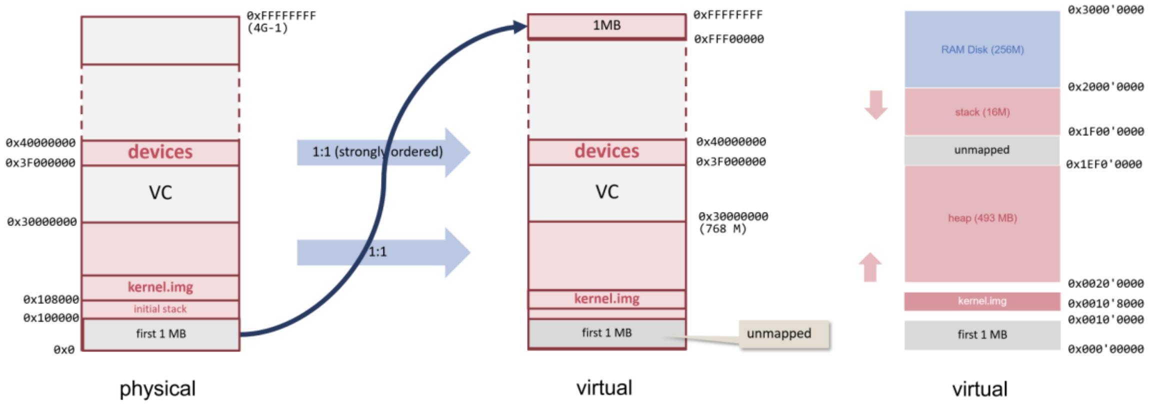
- Hardware starts completely uninitialized (RPI copy boot image to RAM and jump to start)
- From the Kernel we have to initialize:
  - stack registers (i.e. stack pointer) for each CPU mode
  - initialize MMU and page table
  - setup interrupt handlers
  - start timer & enable interrupts
  - initialize UART, RAM disk etc.
  - enter OS scheduling loop

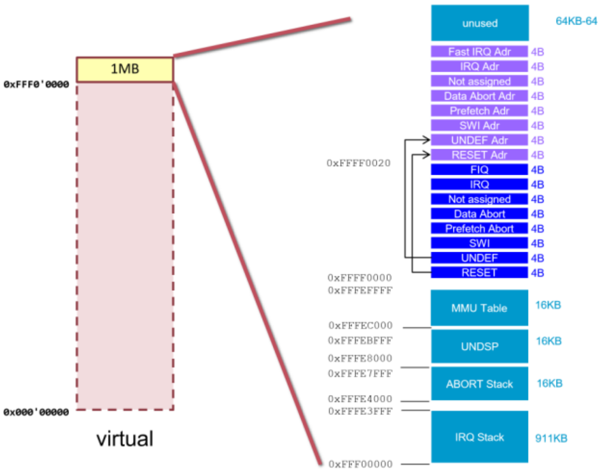

Minos
Command Interpreter and Scheduler
Modules
Dynamic Linker: Module Loader
"imports"
File System
I/O
Kernel Logging etc.
Kernel
Runtime
floating point emulation, memory allocation ...
Memory Management Device Drivers



Memory Management

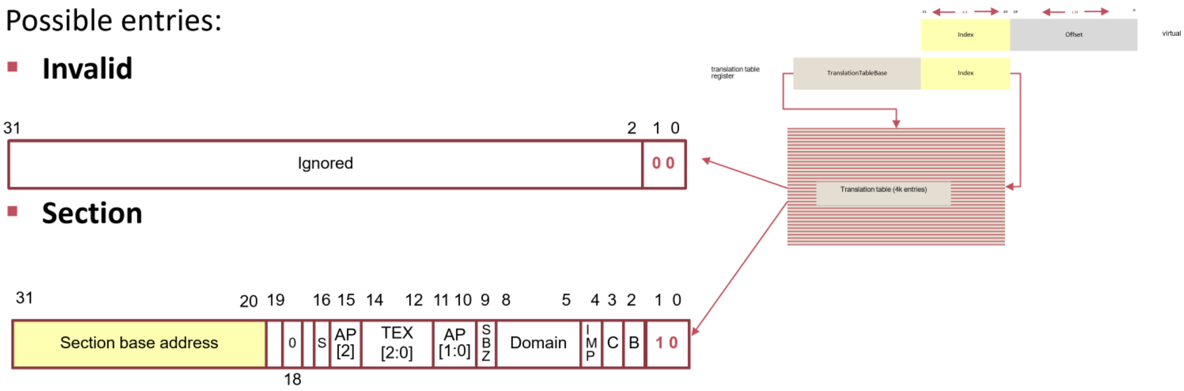
- MMU:
  - Virtual and physical addresses are split up in pages
    (4KB yields 16 pages for 16-bit addresses)
  - Each address has a page number and an offset
    (4-bit page num to represent all 16 pages and 12-bit offset for addresses in a page)
  - TLB: Translation Lookaside Buffer (caches page table entries)
  - See Tannenbaum Section 3.3 for details
  - In ARM MMU is coprocessor 15 (control with MCR instruction)
- Memory Layout Minos:
  - Mapping is 1:1 (1MB with Stacks, MMU table and IRQ vector is special Why?)
  - first page unmapped to trap NIL pointer, first 1MB unmapped by convention
  - Stack/Heap have fixed sizes (in exercise we extend them dynamically, use unmapped page for page fault to extend stack/heap)
  - RAM Disk behaves like a disk (e.g. hard-drive) but is fast (can map /tmp to it)





- Paging Minos:
  - 1MB Pages (i.e. 32-bit address, 12-bit page number, 20-bit offset, 1st level)
  - 4KB as 2nd level pages (they are determined via translation table register)
  - if entry is not in TLB a translation table walk is performed then section base address (physical address) and a set of properties is returned (memory type and sharing)

Possible entries:



(Properties define memory type and sharing attributes)

IRQ/Scheduling

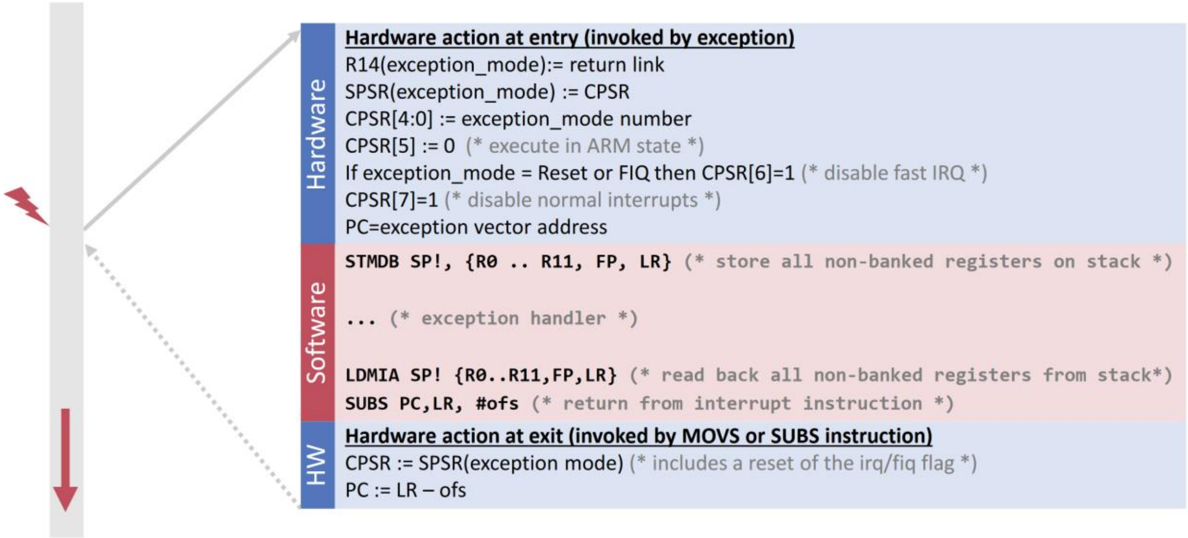
- Exceptions:
  - Interrupt: Async event triggered by a device signal
  - Trap/Syscall: intentional exception (see here)
  - Fault: error condition that a handler might be able to correct
  - Abort: error condition that cannot be corrected

Type	Mode	Address*	return link(type)**
Reset	Supervisor	0x0	undef
Undefined Instruction	Undefined	0x4	next instr
SWI	Supervisor	0x8	next instr
Prefetch Abort	Abort	0xC	aborted instr +4
Data Abort	Abort	0x10	aborted instr +8
Interrupt (IRQ)	IRQ	0x18	next instr +4
Fast Interrupt (FIQ)	FIRQ	0x1C	next instr +4

* alternatively High Vector Address = `0xFFFF0000 + adr` (configurable)
** different numbers in Thumb instruction mode

- Exception handling:
  1. Hardware saves processor state, disable interrupts and set the PC to a defined exception vector address

2. Now the code at the exception vector address is executed but as we cannot fit the whole handler into this memory location, we initialize it with `LDR pc, pc + 18` i.e. set the PC to address above ("jump" to real handler)
3. Oberon saves current processor state on stack or in special registers (behind the scene with {INTERRUPT})
4. Assign to each exception an exception number (with that number we can identify handler code from a table)
5. Switch to exception handler which is running in a different processor mode



Hardware action at entry (invoked by exception)

R14(exception_mode):= return link
SPSR(exception_mode) := CPSR
CPSR[4:0] := exception_mode number
CPSR[5] := 0 (* execute in ARM state *)
If exception_mode = Reset or FIQ then CPSR[6]=1 (* disable fast IRQ *)
CPSR[7]=1 (* disable normal interrupts *)
PC=exception vector address

```
STMDB SP!, {R0 .. R11, FP, LR} (* store all non-banked registers on stack *)

... (* exception handler *)

LDMIA SP! {R0..R11,FP,LR} (* read back all non-banked registers from stack*)
SUBS PC,LR, #ofs (* return from interrupt instruction *)
```

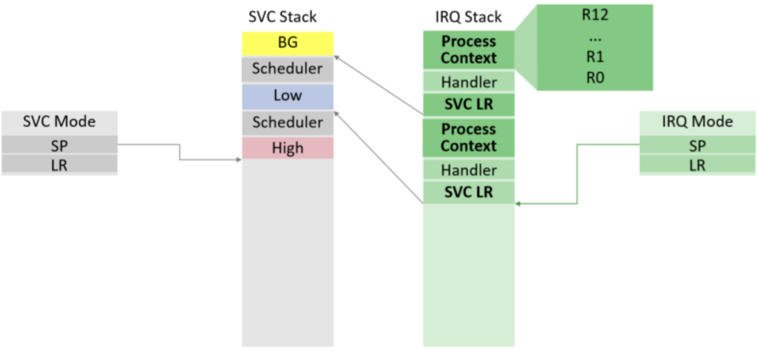
Hardware action at exit (invoked by MOVS or SUBS instruction)

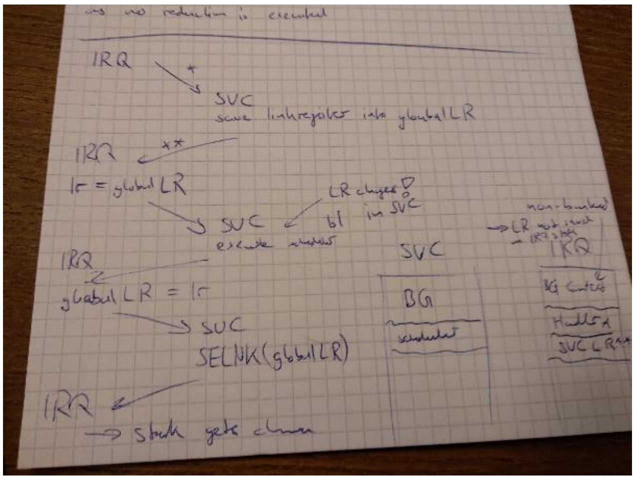
CPSR := SPSR(exception mode) (* includes a reset of the irq/fiq flag *)
PC := LR – ofs

- Exception initialization:
  - Write LDR pc, pc + 18 into interrupt vector address
  - Write address of handlers into vector address + 18 (InstallHandler)
- Exception handlers:
  - **IRQTrap** (async signal by device/timer):
    1. Read pending bits from special register
    2. Disable interrupts
    3. Call handlers according to pending register state
  - **DataAbort** (page fault): Find trapped location and call handler
  - **SWITrap** (e.g. assert): find trapped location (stack trace is possible with old FP as state of trapped program is on stack)

Task Scheduling

- Schedule Strategy:
  - Three task categories (high prio. each 5ms, low prio. Each 20ms, background always)
  - One stack for all task (as high prio. must finish before low prio. can continue)
  - Tasks are stored in linked list sorted by period/priority (priority set with rate monotonic scheduling)
  - Background tasks get started once and are then executed one by one in a loop
  - Scheduler is installed as timer interrupt handler (problem with LR)
  - Every time a timer interrupt occurs the scheduler goes through all periodic tasks and executes them if they are in the correct interval (assume that task doesn't take longer than his interval)
- On an interrupt the SVC context get transferred to the stack but the LR is banked (i.e. doesn't get copied to stack) and doesn't get saved. When we call the scheduler/tasks in SVC mode the LR changes, but we need it later to restore the original context (before IRQ). The trick is to save the SVC LR on the IRQ stack and set it then manually on return.



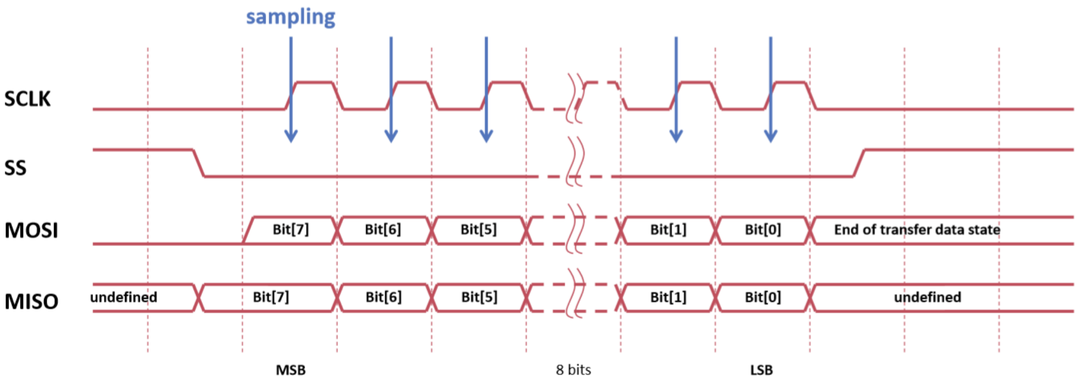

as no reduction is executed
IRQ
*
SVC
save linkregister into globalLR
IRQ
**
lr = globalLR
LR changes?
bl in SVC
SVC
execute scheduler
non-banked
-> LR not saved on IRQ stack
IRQ
SVC
BG
scheduler
BG Context
Handler
SVC LR
IRQ
globalLR = lr
SVC
SELNK(globalLR)
IRQ
-> stack gets clean

SPI

- simplex (->), half-duplex (<-> but not same time), duplex (<-> same time)
- Master-Slave, Master-Multi-Slave (select-line), Multi-Master-Multi-Slave (bus)
- sync (same clock for master/slave) / async (different clocks for master/slave)

	Wires (+Gnd)	Directionality	Synchrony	Distance typ.	Speed typ.	Remarks
RS-232	2/4 –7	full duplex	asynchronous +synchronous	10 m	115kbps / 1Mbps	Point-to-Point Interference prone
RS-485	2/4	half/full duplex	asynchronous	1000 m	Mbps	Differential Signalling
SPI [aka SSP, Microwire]	4 [+Vcc]	full duplex	synchronous	few cm	10 Mbps	Master-Multi-Slave with Slave select
I²C [SMBus]	2 [+Vcc]	half duplex	synchronous	few m	100kbps-3Mbps	Addressed Multi-Master
1-Wire	1	half duplex	time-slot based, synchronous	tens of m	15kbps/ 125kbps	Master-Multi-Slave Parasitic power
USB 2.0	2 [+ Vcc]	half-duplex	asynchronous	few m	12Mbits/ 480 MBits	isochronous/ bulk/ interrupt transfers Differential signalling
USB 3.0	2+4 [+DGnd + Vcc]	full-duplex	asynchronous	few m	5/10/20 GBits (USB 3.0/3.1/3.2)	

- SPI is Master-Slave/Master-Multi-Slave, synchronous, duplex with 4 wires:
  - SCLK: Clock shared between master/slave
  - MOSI: Master-Output-Slave-Input
  - MISO: Master-Input-Slave-Output
  - SS/CS: Slave Select
- Data transfer with two shift registers (one for master/slave), no ack, no interrupts:
  1. Master pulls CS low followed by waiting period (if required by slave)
  2. Clocks starts toggling -> full duplex data transmission in each cycle (MSB first)
  3. When all data is transmitted master stops toggling clock (data send from slave can get lost)



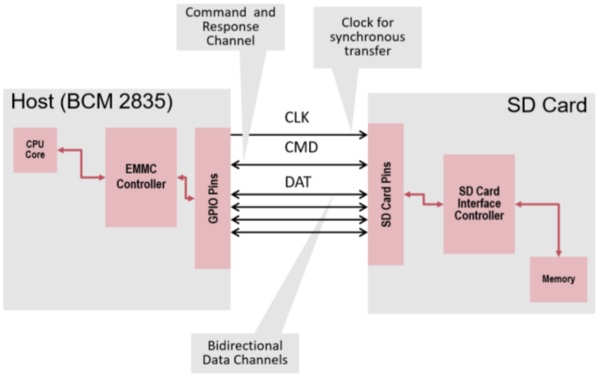
- Polarity = 0 (clock idle is low) Polarity = 1 (clock idle is high)
- Phase = 0 (sample on rising edge) Phase = 1 (sample in falling edge)
- Bit banging: transfer one byte in software instead of dedicated controller. Transfer bit i:
  1. Shift right by i (MSB first)
  2. If bit 1 is one set MOSI otherwise clear MOSI
  3. Set SCLK and clear SCLK
- SPI controller: Raspi has Control/Status register and FIFO
  1. Set transition start bit in CS register
  2. Wait until TXD is high in CS register
  3. Write data to FIFO
  4. Read data from FIFO
  5. Wait until DONE is high in CS register

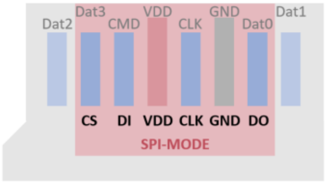
LED Display

- Register address and data is written into a 16-bit shift register
- If CS becomes high then data is latched into register and command gets processed

Memory Cards

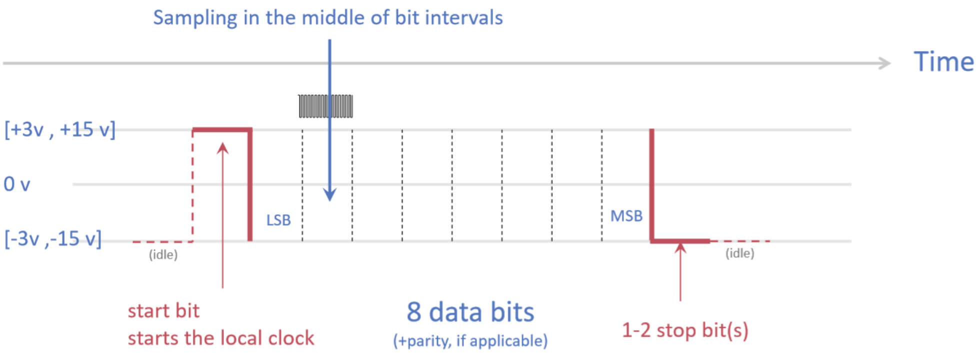
- MMC cards are not used often (SD cards are now used) but controller eMMC is still used
- serial&synchronous transfer of commands and data
- Can also run in SPI mode but then pins are different


Command and Response Channel
Clock for synchronous transfer
Host (BCM 2835)
CPU Core
EMMC Controller
GPIO Pins
CLK
CMD
DAT
SD Card
SD Card Pins
SD Card Interface Controller
Memory
Bidirectional Data Channels



RS232

- Wires: TxD (Device ->Terminal) / RxD (Terminal -> Device) / GND
- if hardware flow control (prevents that data is sent too fast): RTS/RTR/CTS
- async so each participant has own clock (sync over start/stop bit and data rate, fixed length)



- UART[1]: serial transmission of ind. bits in byte package (data bits per byte, parity, # of stop bits, transfer rate configurable)
- UART driver implemented with ring buffer and IRQ if ready

[1] Universal Asynchronous Receiver Transmitter

Case Study 2: A2

- A2 is the Oberon system for multi-processor systems
- SMP[2] architecture (i.e. processor share bus and memory -> same addresses for each processor)

APIC

- APIC[3] is a sophisticated interrupt controller for multiprocessor systems
- Each processor has a local APIC which handles processor specific interrupts (e.g. timers, thermal sensors)
- Interrupts between processors and device interrupts (I/O APICS) are handled over an APIC Bus (integrated into System Bus in newer versions)
- A x86 multiprocessor system starts with a boot processor which enables further processors
- Messages to processors: Start processor, Halt processor, Halt process & schedule new process

Multiprocessor Specification

- Intel standard for memory map, APIC and interrupt modes
- OS can read information about MP system from MP configuration table (e.g. local APIC ID, is boot processor)
- A special data structure is used to get physical address of MP conf. table (data structure search by key)
- Special instructions can also be used to configure APIC (e.g. RDMSR – read machine specific register)

PCI Local Bus

- Standardized configuration address space for all PCI devices (with interrupt routing configuration)
- PCI BIOS provides functionality such as “find device by class code” (presence determined like MP table)
- Devices are addressable via in/out instructions on separate I/O memory address space

Active Oberon

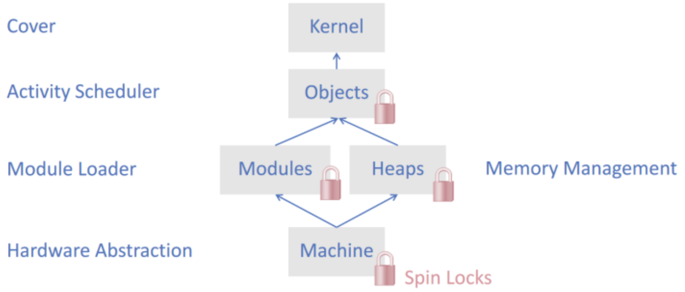
- Oberon + concurrency
- Locks guarantees exclusive access to a resource (e.g. part of page table)
- Monitors are modules/classes that allow safe access of its variable or methods by more than one thread (e.g. Java `synchronized` key word on method)
- Active Objects decouple method execution from method invocation (a scheduler can then handle requests better) -> adds threads to monitor
- Active Oberon provides the following concurrency mechanisms
  - Protection `{EXCLUSIVE}`: Methods tagged exclusive run under mutual exclusion
  - Synchronisation `AWAIT`: Wait until a condition becomes true (only in exclusive section)
  - Parallelism `{ACTIVE}`: module body marked as active is executed as thread (if `NEW(o)` is called)
- AWAIT implementation:
  - Monitors are implemented using queues (e.g. entry queue for threads waiting for entering the monitor and condition queues for threads waiting on condition)
  - When a signal changes a condition on which a thread is waiting on then either the running thread or the waiting thread can enter the monitor -> we have to make a choice
  - Signal-And-Pass (favour waiting thread): If running thread generates a signal, context is switched to waiting thread and he enters the monitor -> many context switches
  - Signal-And-Continue (favour running thread): Running thread sends signal but continues execution (stays in monitor and add waiting thread to entry queue) -> less context switches but condition can change between notifying and reschedule the thread (check condition again with while-loop)
  - Signal-And-Exit (Oberon implementation): signal is sent after signalling thread finished execution and then signalled thread is scheduled (await queues have priorities) -> prevents while-problem (condition is re-evaluated if thread leaves exclusive section and lock is then granted to waiting thread)

[2] Symmetrical Multiple Processors

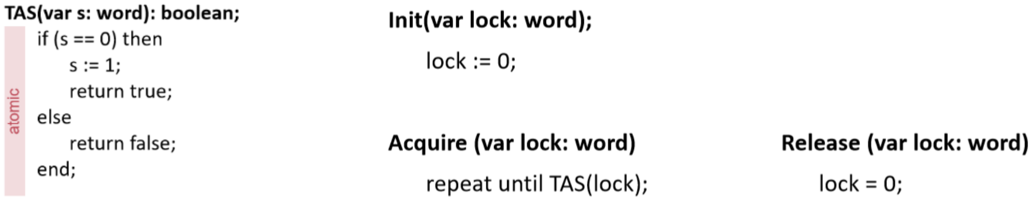
[3] Advanced Configuration and Power Interface

A2 Overview

- modular kernel (above Objects we can use Active Oberon features):



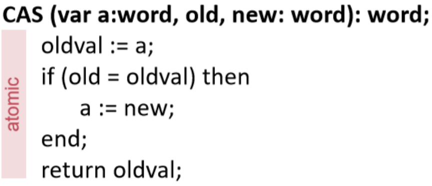
  - Machine: implemented by different architectures (I386, AMD64 etc.), low level locks, processor management., virtual memory management., IRQs
  - Heap: portable, heap allocation, garbage collector, finalizers
  - Modules: portable, load/unload module, termination handlers
  - Objects: conceptually portable, scheduler, timer interrupt, process synchronisation (implements monitor)
  - Kernel: cover for underlying modules
- A2 implements only non recursive locks (process cannot enter a critical section twice even if it holds lock)
- Hardware supports atomic operations with special instructions (e.g. LDREX load register exclusive in ARM):
  - These instructions are executed atomically and thus slower
  - Test-And-Set (TAS)
  - Compare-And-Swap (CAS): e.g. Intel x86
  - Load Linked / Store Conditional: e.g. ARM
- TAS Semantics & Spin Lock



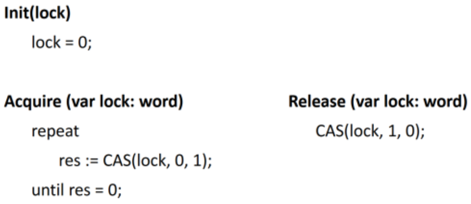
```
TAS(var s: word): boolean;
    if (s == 0) then
        s := 1;
        return true;
    else
        return false;
    end;
```

```
Init(var lock: word);
    lock := 0;

Acquire (var lock: word)
    repeat until TAS(lock);

Release (var lock: word)
    lock = 0;
```

- CAS Semantics & Spin Lock



```
CAS (var a:word, old, new: word): word;
    oldval := a;
    if (old = oldval) then
        a := new;
    end;
    return oldval;
```



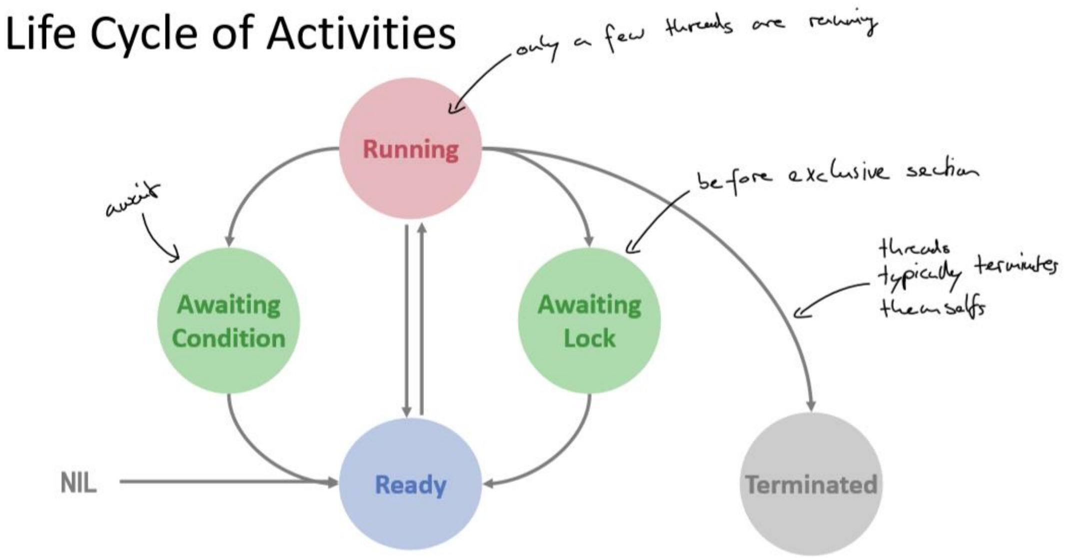
```
Init(lock)
    lock = 0;

Acquire (var lock: word)
    repeat
        res := CAS(lock, 0, 1);
    until res = 0;

Release (var lock: word)
    CAS(lock, 1, 0);
```

- Boot procedure (executed only on boot processor):
  - Start BIOS firmware
  - Load A2 bootfile
  - Init modules: Machine, Heaps, ..., Objects (scheduler), Kernel (start all processors), ..., Bootconsole (read config and execute boot commands)
- Processor start-up:
  - Start processor P (boot processor)
    1. Setup boot program (Machine.InitProcessors)
    2. Enter processor Ids into table (Machine.InitBootPage)
    3. Send *startup* message to P via APIC
    4. Wait with timeout on *started* flag by P
  - Boot program (for each processor)
    1. Set 32-bit runtime environment

2. Init control reg., memory management, interrupt handling, APIC
3. Set *started* flag
4. Setup scheduler

- Boot processor proceeds with boot console

Activity Management


Life Cycle of Activities

- global running array stores which process (thread) is currently running
- global ready queues array holds ready processes in different queues (acc. to their priority)
- per (monitor) object condition and lock queues are maintained in the object header
- Process description (representation of process, state contains PC, SP, registers etc.)

```
TYPE
  Process = OBJECT
    …
    stack: Stack;
    state: ProcessState;
    preempted: BOOLEAN;
    condition: PROCEDURE (slink: ADDRESS);
    conditionFP: ADDRESS;
    priority: INTEGER;
    obj: OBJECT;
    next: Process
  END Process;
```

- Process dispatching: insert/select process from the ready queue with highest priority
- Process creation: acquire lock for module Objects, add process to ready queue, release lock

Memory Management

- single shared virtual address space for all processes
- heap is single shared area in virtual address space directly mapped to physical addresses
- each process has its own stack (stack addresses are stored in process descriptor)
- Easy to achieve with virtual memory:
  - allocate stack in pages
  - use page fault to allocate more
  - deallocate stack with garbage collection (in process finalizer)
  - stack frames in virtual memory need to be contiguous (in virtual address space)

Context Switch

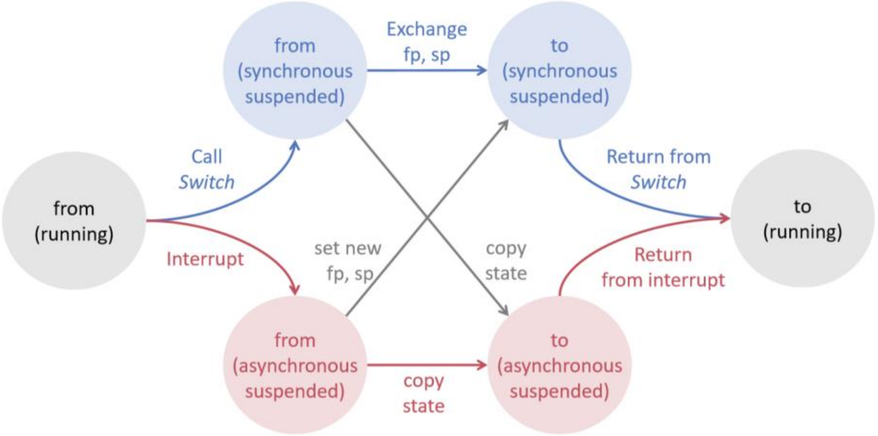
- There is an idle activity which is scheduled if no activity is in queue
- Two types of context switches:
  - Synchronous (system expects these switches):
    - Explicit: Terminate, Yield
    - Implicit: waiting on condition, mutual exclusion

  - Asynchronous:
    - Preemption (process get interrupted without cooperation): priority handling, time slicing

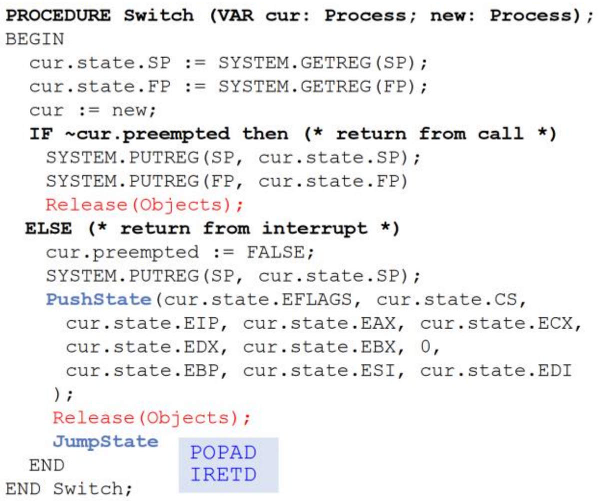
- If we want to switch to a process which was synchronous suspended, we just set the SP and FP and return from the method (as the process knows where he left execution we don't have to store much, actually it is enough to just store the SP and BP as the rest of the process state is saved by the calling convention of a function)

  (as this process was synchronously suspended it had to call SwitchTo once, so the stack looks like before a function call, if we now set the correct stack pointers and return from a function, the suspended process continue execution)
- If we want to switch to a process which was asynchronously suspended, we have to push the whole process state to the stack and then return from an interrupt (we don't know where the process left execution so we have to save the whole state just to make sure we can restore the process properly)

  (this process was suspended by an interrupt, so we have to prepare the stack such that it looks like if an interrupt has prepared the stack -> we have to copy the whole state)

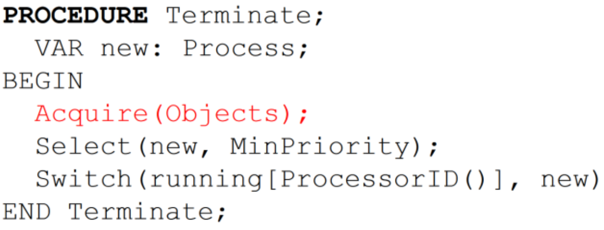


- Code for switching from a sync suspended process to a sync (first if-branch) or async (second if-part)

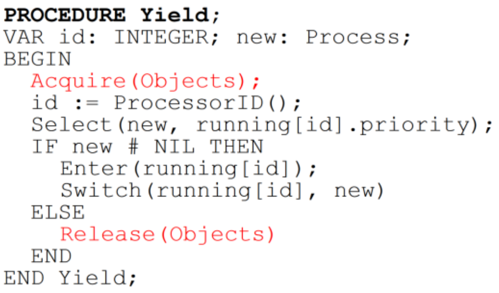


```
PROCEDURE Switch (VAR cur: Process; new: Process);
BEGIN
  cur.state.SP := SYSTEM.GETREG(SP);
  cur.state.FP := SYSTEM.GETREG(FP);
  cur := new;
  IF ~cur.preempted then (* return from call *)
   SYSTEM.PUTREG(SP, cur.state.SP);
   SYSTEM.PUTREG(FP, cur.state.FP)
   Release(Objects);
  ELSE (* return from interrupt *)
   cur.preempted := FALSE;
   SYSTEM.PUTREG(SP, cur.state.SP);
   PushState(cur.state.EFLAGS, cur.state.CS,
     cur.state.EIP, cur.state.EAX, cur.state.ECX,
     cur.state.EDX, cur.state.EBX, 0,
     cur.state.EBP, cur.state.ESI, cur.state.EDI
    );
    Release(Objects);
    JumpState      POPAD
  END              IRETD
END Switch;
```

- Examples that use Switch function

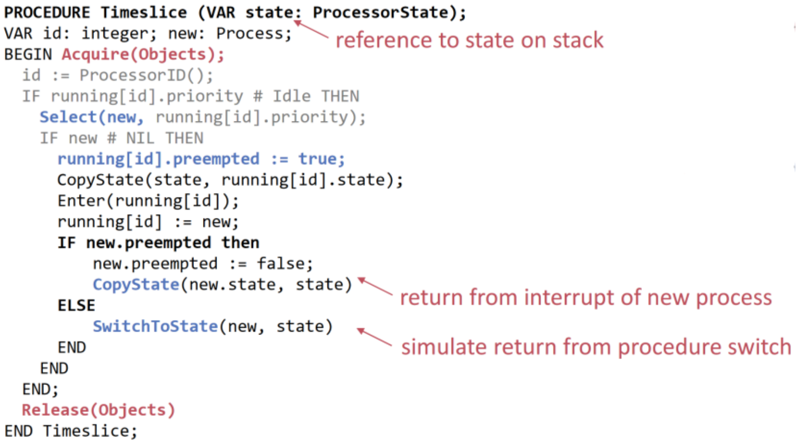


```
PROCEDURE Terminate;
  VAR new: Process;
BEGIN
  Acquire(Objects);
  Select(new, MinPriority);
  Switch(running[ProcessorID()], new)
END Terminate;
```



```
PROCEDURE Yield;
VAR id: INTEGER; new: Process;
BEGIN
  Acquire(Objects);
  id := ProcessorID();
  Select(new, running[id].priority);
  IF new # NIL THEN
    Enter(running[id]);
    Switch(running[id], new)
  ELSE
    Release(Objects)
  END
END Yield;
```

- Code for switching from async suspended process to sync or async



```
PROCEDURE Timeslice (VAR state: ProcessorState);
VAR id: integer; new: Process;
BEGIN Acquire(Objects);
  id := ProcessorID();
  IF running[id].priority # Idle THEN
    Select(new, running[id].priority);
    IF new # NIL THEN
      running[id].preempted := true;
      CopyState(state, running[id].state);
      Enter(running[id]);
      running[id] := new;
      IF new.preempted then
          new.preempted := false;
          CopyState(new.state, state)
      ELSE
          SwitchToState(new, state)
      END
    END
  END;
  Release(Objects)
END Timeslice;
```

reference to state on stack

return from interrupt of new process

simulate return from procedure switch

Synchronisation

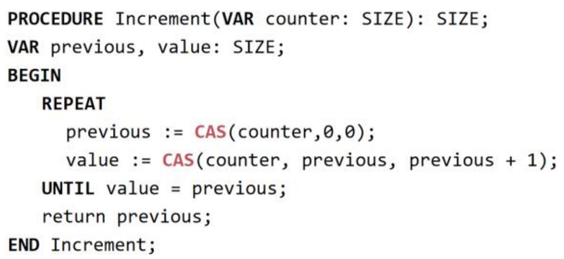
- System adds special fields to object header to support object locking and condition management:
  - headerLock: lock which we have to acquire if we want to change object header
  - lockedBy: process that holds lock of object (not object header)
  - awaitingLock: queue for processes that want object lock
  - awaitingCondition: queue for processes that wait for condition
- Lock object:
  1. Acquire header lock (with Spin Lock)
  2. if lockedBy is NIL we can acquire the object lock and set the running process as owner
  3. else we put running process in awaitingLock queue and switch to new process
- Unlock object:
  1. Acquire header lock
  2. Check if condition is fulfilled and if so, take process from condition queue
  3. else take process from lock queue
  4. enter the new process into ready queue
- Implementation of AWAIT
  1. Acquire header lock
  2. find a waiting process (either from lock or condition queue), give him object lock and enter it into ready queue
  3. Add condition (from await param) to running process and put it into condition queue
  4. Switch to new thread
- each AWAIT is replaced by a syscall to await and a helper procedure
  - helper procedure takes frame pointer and all variables are access relatively to this frame pointer
  - await-syscall takes helper procedure, FP and self reference (used to grant his lock to another process)
  - await-syscall does the following things
    - release object lock (take process from lock queue, give lock to him, enter him into ready queue)
    - suspend running process (save procedure & FP into process description, add into condition queue)
  - at the end of a EXCLUSIVE block a call to unlock is performed
    - call to FindConditions which goes through condition queue and return a process which condition is true (check with helper procedure and old FP)
    - if a condition was true grant this process the lock and insert it into ready queue (otw. pick one from lock queue)

Lock Free Kernel

- Problems with locks:
    - Deadlock: Threads block each other because every thread holds a lock that another need to continue
    - Livelock: You realize a deadlock can occur -> try to resolve (but this happens again and again)
    - Starvation: Only one thread accesses a resource and the other never
    - Waiting processes depend heavily on cooperative of competing processes
- Lock-freedom: at least one algorithm makes progress, even if other algorithms run concurrently, fail or get suspended (implies system-wide progress bit not freedom of starvation)
- Wait-freedom: each algorithm eventually makes progress (implies lock- and starvation-freedom)
- Another view:
    - someone makes progress: deadlock-free (blocking) or lock-free (non-blocking)
    - everybody makes progress: starvation-free (blocking) or wait-free (non-blocking)
- lock-free programming: scheduler provides non-blocking atomic operations instead of blocking synchronisation primitives (e.g. EXCLUSIVE, AWAIT)
- lock-free describes a property of a non-blocking algorithm
- Implementation goals:
    - lock-freedom: guarantee progress and re-entrant algorithms (we can suspend algorithm and run it later)
    - portability: hardware independent, simple
- Principle: exclusively employ non-blocking algorithms
    - use implicit cooperative multitasking (all processes have to cooperate to guarantee system progress)
    - no virtual memory
    - limits in optimization

Active Oberon Implementation

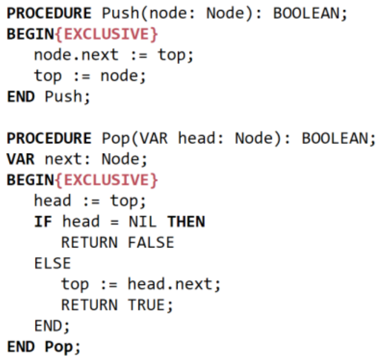
- Locks in Kernel:
    - Scheduling queues / heaps in object header and ready queue array
    - Memory management
- CAS is implemented wait-free by hardware (i.e. hardware guarantees some fairness)
- Memory Model:
    - defines behaviour of concurrent programs that share data
    - enables compiler optimisations and provide guarantees for programmer (Java has one, C++11 too)
    - In A2 only two rules:
        - Data shared between two or more activities has to be protected using exclusive blocks (unless CAS is used)
        - Changes to shared data are visible to other activities after leaving exclusive block or executing CAS
- CAS instruction is a statement of Oberon:
    - Operation is executed atomically -> result visible instantaneously to other processes
    - CAS(variable, x, x) is a atomic read
    - Compiler is required to implement CAS as a synchronisation barrier
        - performance suffers with increasing numbers of contenders to the same variable
        - CAS with back off (sleep for a while) has better performance
- Non-blocking counter (not wait-free, ABA-Problem occurs if process reads previous, other threads overflow counter and previous still holds old value but is not same state)



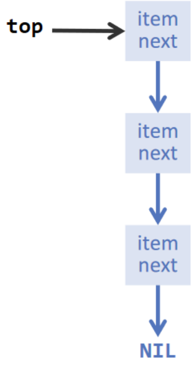
```
PROCEDURE Increment(VAR counter: SIZE): SIZE;
VAR previous, value: SIZE;
BEGIN
    REPEAT
      previous := CAS(counter,0,0);
      value := CAS(counter, previous, previous + 1);
    UNTIL value = previous;
    return previous;
END Increment;
```

Stack

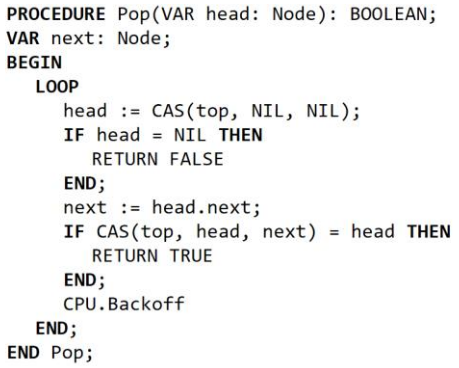
- Blocking stack:



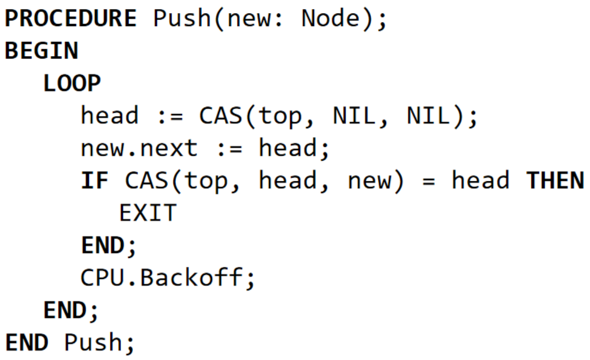
```
PROCEDURE Push(node: Node): BOOLEAN;
BEGIN{EXCLUSIVE}
   node.next := top;
   top := node;
END Push;

PROCEDURE Pop(VAR head: Node): BOOLEAN;
VAR next: Node;
BEGIN{EXCLUSIVE}
   head := top;
   IF head = NIL THEN
      RETURN FALSE
   ELSE
      top := head.next;
      RETURN TRUE;
   END;
END Pop;
```



- Non-blocking:



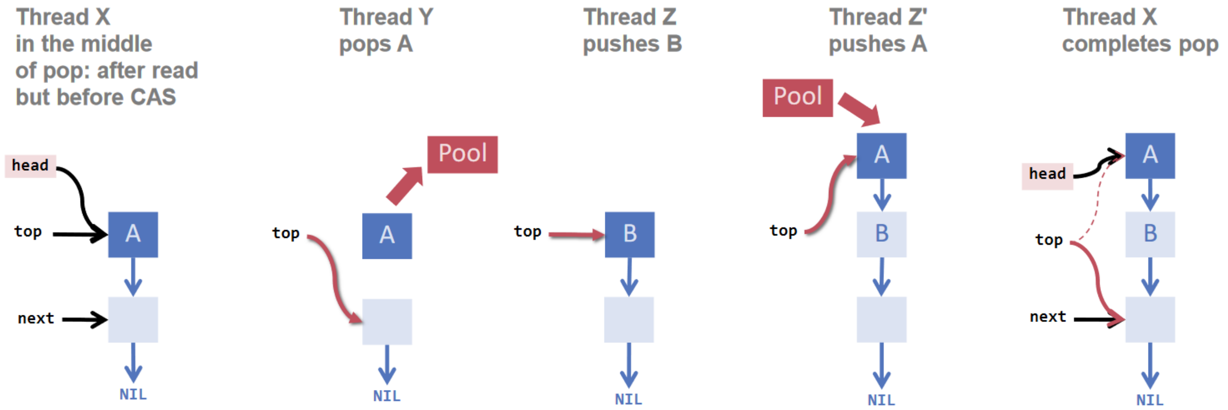
```
PROCEDURE Pop(VAR head: Node): BOOLEAN;
VAR next: Node;
BEGIN
   LOOP
      head := CAS(top, NIL, NIL);
      IF head = NIL THEN
         RETURN FALSE
      END;
      next := head.next;
      IF CAS(top, head, next) = head THEN
         RETURN TRUE
      END;
      CPU.Backoff
   END;
END Pop;
```



```
PROCEDURE Push(new: Node);
BEGIN
   LOOP
      head := CAS(top, NIL, NIL);
      new.next := head;
      IF CAS(top, head, new) = head THEN
         EXIT
      END;
      CPU.Backoff;
   END;
END Push;
```

(if top has not changed replace top)

- Node reuse doesn't work with this implementation as other processes could push the same node again and then we don't recognize it -> ABA Problem



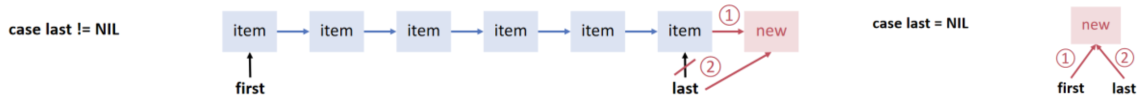
ABA Problem

- ABA problem occurs when one activity fails to recognise that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overall state has not been changed

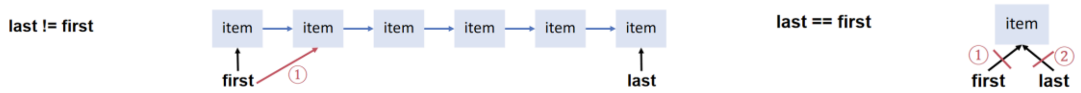
- Solutions:
  - DCAS (double compare and swap): not available on most platforms
  - hardware transactional memory (like DB transactions): not available on most platforms
  - Garbage Collection: relies on existence of GC (e.g. inner kernel runtime has no GC)
  - Pointer Tagging: does not cure the problem but can be practical
  - Hazard Pointers
- Pointer Tagging:
  - pointers are aligned 32-bit to fit better in memory
  - pointer address is 16-bit long; this leaves the 5 LSB to zero to get a multiple of 32
  - We can use these 5 bits as a tag and increase a counter each time a pointer is used
  - this makes ABA problem less likely because we have 32 versions of each pointer
- Hazard Pointers
  - ABA problem stems from reuse of pointer P
  - Process X reads P but hasn’t yet written, while process Y modifies the state
  - Solution:
    - Before X reads P, it marks P as hazardous (insert it into array of size # of threads)
    - When finished (after CAS), X removes it from array
    - Before Y reuses P it checks the array

Unbounded Queue

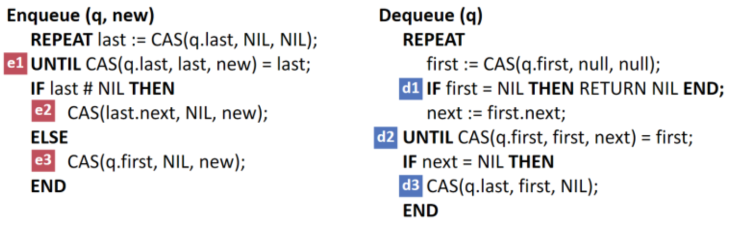
- Enqueue



- Dequeue

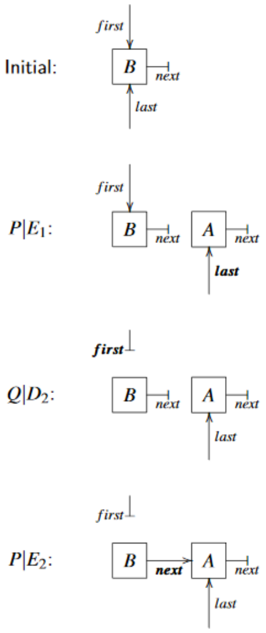


- Naive approach



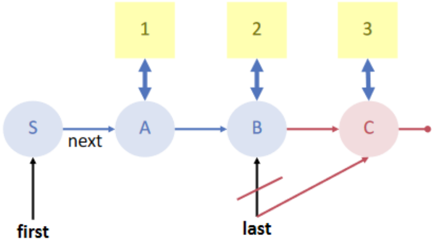
```
Enqueue (q, new)
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
e2      CAS(last.next, NIL, new);
    ELSE
e3      CAS(q.first, NIL, new);
    END
```

```
Dequeue (q)
    REPEAT
        first := CAS(q.first, null, null);
d1      IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
d3      CAS(q.last, first, NIL);
    END
```

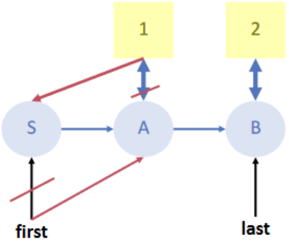


- Problem:
  - we treat empty queues differently from non-empty queues
  - advantages of two pointers is that they don’t interfere but if e.g. queue is empty we have to set the last pointer and the first pointer (update two references with CAS doesn’t work)
  - this can lead to permanent corruption of the data structure (see image above)

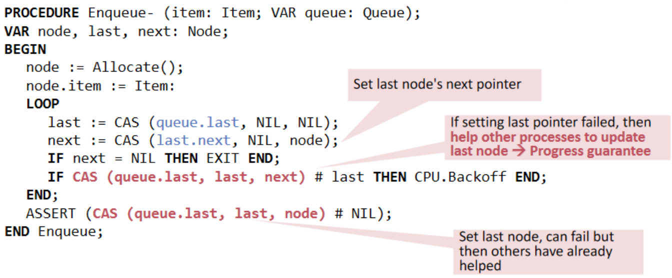
- Sentinel
    - A sentinel is a dummy which is always the first element
    - Its purpose is to distinguish between an empty/non-empty list
    - first and last point always to an existing element and we don’t have to invalidate them
    - Problem: Intermediate nodes needs memory allocation which is not memory-efficient



Item enqueued together with associated node.



A becomes the new sentinel. S associated with free item.

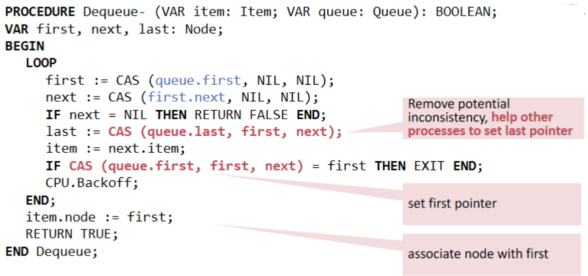


```
PROCEDURE Enqueue- (item: Item; VAR queue: Queue);
VAR node, last, next: Node;
BEGIN
   node := Allocate();
   node.item := Item:
   LOOP
      last := CAS (queue.last, NIL, NIL);
      next := CAS (last.next, NIL, node);
      IF next = NIL THEN EXIT END;
      IF CAS (queue.last, last, next) # last THEN CPU.Backoff END;
   END;
   ASSERT (CAS (queue.last, last, node) # NIL);
END Enqueue;
```

Set last node's next pointer

If setting last pointer failed, then help other processes to update last node → Progress guarantee

Set last node, can fail but then others have already helped



```
PROCEDURE Dequeue- (VAR item: Item; VAR queue: Queue): BOOLEAN;
VAR first, next, last: Node;
BEGIN
   LOOP
      first := CAS (queue.first, NIL, NIL);
      next := CAS (first.next, NIL, NIL);
      IF next = NIL THEN RETURN FALSE END;
      last := CAS (queue.last, first, next);
      item := next.item;
      IF CAS (queue.first, first, next) = first THEN EXIT END;
      CPU.Backoff;
   END;
   item.node := first;
   RETURN TRUE;
END Dequeue;
```

Remove potential inconsistency, help other processes to set last pointer

set first pointer

associate node with first

- Invariant: linked list of nodes is always up to date (i.e. the next pointers are always correct)
- last pointer can intentionally lag behind (we can identify last node by next equal NIL)
- enqueue doesn’t have to update first pointer (when queue is empty) thanks to sentinel
  -> just update next pointer of last node (identified by NIL) and use queue.last to find last node faster
- Remaining problem: assign nodes needs dynamic memory allocation (not efficient) and reusing nodes not possible because of ABA problem
- We could use Hazard Pointers but time to check hazard pointers grow with # of threads (unbounded)
- Use guarantees of cooperative multitasking to implement lock-free queues efficiently

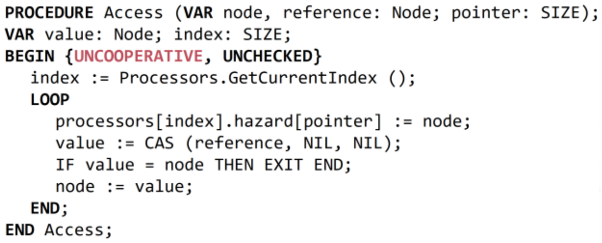
Cooperative Multitasking

- Pre-emptive multitasking disadvantages:
    - depends on timer interrupts (hardware dependent) and runs in kernel mode
    - does a lot of expensive asynchronous context switches (we have to save whole context)
    - non-parallel as scheduler has to lock queues
- Cooperative multitasking:
    - can be implemented in software (hardware independent)
    - fewer steps and all in user mode
        - pick new process to schedule from lock-free queue
        - switch base pointer
        - return from function call
    - Processes have only two (implicit) states:
        - running on a processor with a given index
        - suspended (in lock-free queue)
    - All context switches are synchronous, so processes need only represent a stack as processor state (registers etc. are already stored by compiler)
    - To ensure cooperation the compiler inserts code to decrease quantum (how long can the process use CPU) and switch if the quantum is too low
    - {UNCOOPERATIVE} flag tells the compiler not to insert these instructions there
    - Pros:

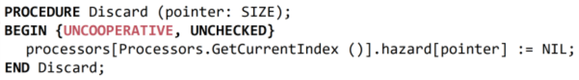
        - lightweight (cost of function call)
        - allow global optimization (calls to scheduler known to compiler)
    - Cons:
        - overhead inserted by scheduler code
        - currently sacrifices one general purpose register (not hardware independent)
        - requires special compiler
- Cooperative MT guarantees that no more than M threads are executing inside an uncooperative block (M = # of processors)
- Now we can associate hazard pointers with the processor and the number of hazard pointers limited by M -> queue is efficient with hazard pointers (constant search time)
- Interrupts:
    - number of contenders is limited by # of processor so an interrupt handler cannot be handled immediately
    - model interrupt handlers as virtual processors -> M = # processor + # of pot. concurrent interrupts

Lock-free queue efficient & correct – HURRA!

- Now we can build a queue which reuses nodes with hazard pointers and doesn't corrupt (sentinel)
    - Marking/Unmark node as hazardous (pointer is just a constant per "pointer" e.g. last = 0)

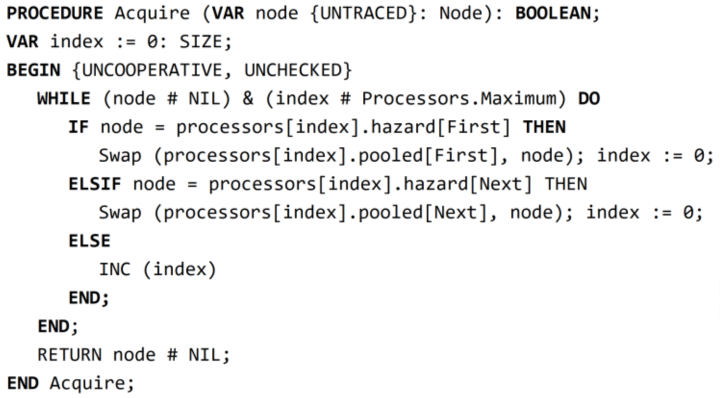


```
PROCEDURE Access (VAR node, reference: Node; pointer: SIZE);
VAR value: Node; index: SIZE;
BEGIN {UNCOOPERATIVE, UNCHECKED}
   index := Processors.GetCurrentIndex ();
   LOOP
      processors[index].hazard[pointer] := node;
      value := CAS (reference, NIL, NIL);
      IF value = node THEN EXIT END;
      node := value;
   END;
END Access;
```



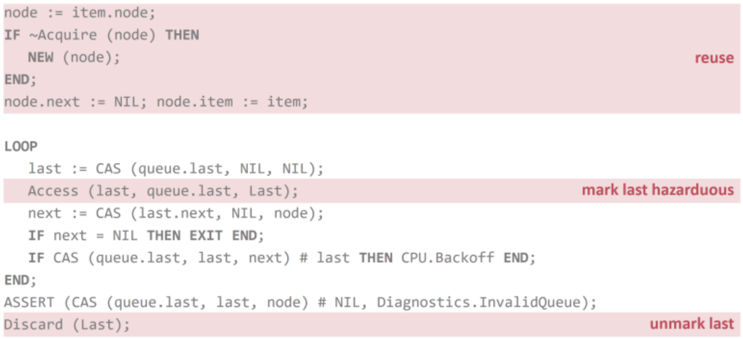
```
PROCEDURE Discard (pointer: SIZE);
BEGIN {UNCOOPERATIVE, UNCHECKED}
   processors[Processors.GetCurrentIndex ()].hazard[pointer] := NIL;
END Discard;
```

    - Find node to reuse (we need only for first and next a new node)
        - go through all processors and check if node is hazardous
        - if node is hazardous swap with pooled node and check again (pooled node could still be hazardous)
        - do this until not hazardous



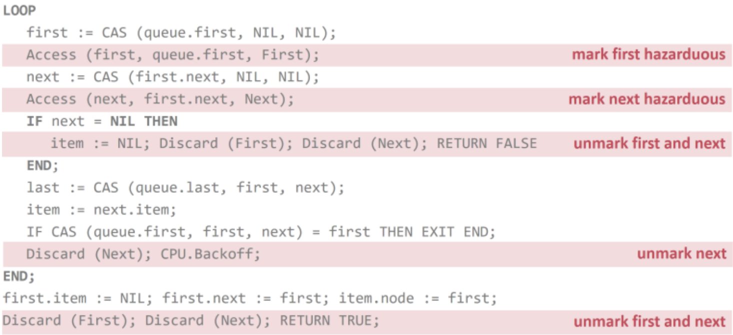
```
PROCEDURE Acquire (VAR node {UNTRACED}: Node): BOOLEAN;
VAR index := 0: SIZE;
BEGIN {UNCOOPERATIVE, UNCHECKED}
   WHILE (node # NIL) & (index # Processors.Maximum) DO
      IF node = processors[index].hazard[First] THEN
         Swap (processors[index].pooled[First], node); index := 0;
      ELSIF node = processors[index].hazard[Next] THEN
         Swap (processors[index].pooled[Next], node); index := 0;
      ELSE
         INC (index)
      END;
   END;
   RETURN node # NIL;
END Acquire;
```

    - Enqueue



```
node := item.node;
IF ~Acquire (node) THEN
   NEW (node);                                                    reuse
END;
node.next := NIL; node.item := item;

LOOP
   last := CAS (queue.last, NIL, NIL);
   Access (last, queue.last, Last);                mark last hazarduous
   next := CAS (last.next, NIL, node);
   IF next = NIL THEN EXIT END;
   IF CAS (queue.last, last, next) # last THEN CPU.Backoff END;
END;
ASSERT (CAS (queue.last, last, node) # NIL, Diagnostics.InvalidQueue);
Discard (Last);                                               unmark last
```

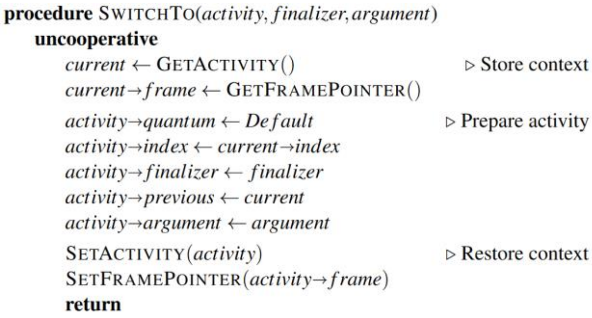
- Dequeue



```
LOOP
   first := CAS (queue.first, NIL, NIL);
   Access (first, queue.first, First);                          mark first hazarduous
   next := CAS (first.next, NIL, NIL);
   Access (next, first.next, Next);                             mark next hazarduous
   IF next = NIL THEN
      item := NIL; Discard (First); Discard (Next); RETURN FALSE   unmark first and next
   END;
   last := CAS (queue.last, first, next);
   item := next.item;
   IF CAS (queue.first, first, next) = first THEN EXIT END;
   Discard (Next); CPU.Backoff;                                  unmark next
END;
first.item := NIL; first.next := first; item.node := first;
Discard (First); Discard (Next); RETURN TRUE;                   unmark first and next
```

Lock-free scheduling

- Use non-blocking queues and discard coarse locking
- A process is represented with a simple task descriptor:
  - inherits from Item to use it with ready queue
  - following fields:
    - object (associated active object)
    - quantum
    - priority
    - processor index where process is running on
    - stack (pointer to array)
    - frame (frame pointer)
- The stack is represented as an array of bytes which contains the whole call stack
- Stacks corresponds to memory blocks on the heap (can grow dynamically, # processes only limited by memory)
- Stack stores also process context (most work done by compiler, we just save address to stack -> SP/FP)
- SwitchTo is used to switch to another process:
  - First save frame pointer (address to stack) into task descriptor (rest was already pushed onto stack before SwitchTo was called)
  - prepare the new process
  - SetActivity/SetFramePointer restore context of new process
  - the actual switch is performed if we return from the SwitchTo procedure



```
procedure SwitchTo(activity, finalizer, argument)
    uncooperative
        current ← GetActivity()                         ▷ Store context
        current→frame ← GetFramePointer()
        activity→quantum ← Default                      ▷ Prepare activity
        activity→index ← current→index
        activity→finalizer ← finalizer
        activity→previous ← current
        activity→argument ← argument
        SetActivity(activity)                           ▷ Restore context
        SetFramePointer(activity→frame)
        return
```

- We get race-conditions:
  - Current process enqueue himself into ready queue
  - Another processor could dequeue the current process and run it immediately
  - Now the same process could run on the same stack frame -> corruption of stack
  - Solution: Use finalizer to run code of “old” process in context of new process and enqueue old process there into ready queue

Lock-Free Stack Management

- Three ideas:
  - reserve consecutive pages in virtual memory but only associate few physical pages (if more is needed -> detect with page fault and extend [like A2])

    - statically examine call graph of process to determine required stack size (difficult to find size e.g. recursion)
    - compiler instrument program to allocate more space if needed
- We use combination of this idea:
    - instead of examining complete call graph -> find required stack memory per procedure call
    - compiler adds stack check at beginning of function prologue (callee side)
    - process descriptor contains pointer to beginning of stack and limit of stack
    - when a new stack activation frame exceeds limit -> call procedure ExpandStack
- Stack can either be expanded by copying the old to the new stack or by link old/new stack together
    - when copying stack must keep track of all pointers, call-by-reference parameters etc. -> expensive
    - use linked stack which is better (how does linked stack work?)

Lock-Free Runtime

- consequent use of lock-free algorithms in the kernel
- Oberon synchronisation primitives (e.g. AWAIT) implemented on top
- efficient unbounded lock-free queues (ABA-Problem solved using hazard pointers)
- implicit cooperative multi-tasking (can switch of scheduling locally with {UNCOOPERATIVE})

Case Study 3: FPGA

Introduction

Build from Scratch

- Pros:
    - clear design (easy to see where to extend or fix)
    - flexible and based solely on problem domain and experience
    - reduce complexity (less baggage and things that you don't need)
    - increase control (as less dependencies)
    - more choices of implementation (tailor made for customer)
    - eliminate surprises and thus delivery on time and budget
- Cons:
    - duplication of effort (reinventing the wheel)
    - more fundamental knowledge required
    - may be more actual work (the first time)
    - risky: tendency to underestimate
    - restricted component choices
    - not for short term

Configurable Hardware (FPGA)

- first we had PALs[4] (often one-time programmable), then GALs[5] emerged (eraseable, more functionality). There are also CPLD[6] with complexity between PALs and FPGAs
- FPGA loads configuration (bitstream) (not fixed like VLSI (combine mio. of transistor into IC) or ASIC (Application Specific Integrated Circuit)
- Application: telecommunications, industry, banking (crypto mining)
- flexible, but not best in performance or power
- now big and fast enough for entire SoC

Hardware / Software Boundary

- traditionally only firmware gave hardware its personality
- this led to strong interfaces and standards (separation between hardware/software dev)

[4] Programmable Array Logic
[5] Generic Logic Array
[6] Complex Programmable Logic Device

- innovation was only possible for a few big companies
- now, configurable hardware democratises which opened the new field of hardware / software co-design

Resources inside FPGA

- look-up tables implement a logic function (truth table)
- D-type flip-flops remember a binary value from clock to clock
- routing resources connect the various elements
- global clocks
- block RAM for holding organised data
- SPI, I2C or memory interface are often pre-implemented (incl. JTAG interface for testing board)
- DSP[7] functions like multiply or multiply-and-accumulate
- basic logic (like LUT and flip-flops) organised in logic blocks

Hardware Description Language

- used to describe circuits textually which is more precise, scalable and formal than schematic capture
- same source code used for both simulation and synthesis
- commercial examples: Verilog, VHDL (at ETH: Lola, Active Cells)
- very different from software programming languages as everything is parallel (different notion of time) and resources are limited
- still not a perfect description (e.g. timing, metastability[8])

Toolchain

- Programming a FPGA requires the following steps:
  1. synthesis: create logical netlist of components and connections (Yosys)
  2. technology mapping: against a physical chip family
  3. placement: cells onto a particular target chip (Arachne)
  4. routing: of connections between placed cells (Arachne)
  5. bitstream generation: encoding used to configure the chip (IceStorm)
  6. timing analysis: check if all delays are within requirements (icetime calculates max. frequency for design but actual frequency is part of design not set by tools)
- simulation is possible at synthesis level (even power and electrical models e.g. IBIS[9])
- usually proprietary integrated HDL-driven environment ($$$) and also pre-fabricated blocks available
- integrated software/hardware co-design (see Active Cells)

Implement Logic Functions

- often several choices for familiar functions
- example functions:
  - multiplexers (lots of these)
  - shift-registers/barrel-shifters (shift multiple bits in one clock cycle)
  - ripple-carry vs. carry-save vs. carry-lookahead adders
  - ripple counters vs. synchronous counter (ripple counter move bits forward, syn. counter changes all flip-flops on clock cycle)
  - composition (e.g. multiplier, divider, ALU)
  - simple external interfaces (e.g. RS232, SPI)
  - complex external interfaces (e.g. memory controllers, video processing)

Our FPGA-Board

- Lattice iCE40 low-power, low-cost FPGA with 7680 LUTs & 128Kb BRAM
- 1MB external fast (10ns) asynchronous static RAM
- SPI flash for non-volatile FPGA bitstream and data storage

[7] digital signal processing

[8] behaviour between two defined states

[9] manufacturers provide information about power consumption without leaking intellectual property

- microcontroller for system management
- 2.5V & 3.3V GPIO, VGA connector
- 100MHz oscillator and 3.3V/2.5V/1.2V regulators

RISC

Introduction

- follows successful reduced-instruction-set design philosophy
- registers instead of stack machine
- Harvard (RISC0) or Von Neumann (RISC5) memory architecture
- hardware floating-point option
- defined in Verilog (developed originally for Xilinx Spartan 3 FPGA)

Overview

- computer consists of ALU, control unit and a store
- ALU contains 16 general purpose 32-bit registers
- control unit consists of instruction register (holds currently executing instruction) and a program counter (points to next instruction)
- 4 flags (NZCV[10]) for conditions
- Three types of instructions:
  - Register instructions:
    - e.g. ADD a, b, n (n could be register or immediate)
    - Result is in register a
    - affect flag registers
  - Memory instructions:
    - e.g. LD a, b, off (load b+off into a)
    - word or byte access
  - Branch instructions:
    - LT reg (if less than jump to reg)
    - branch and link (if a bit is set read PC from register otw. with offset)

RISC0 Implementation on Lattice FPGA

- Harvard RISC0 CPU, 2K words data RAM, separate instruction memory
- complete ALU (simple multiplier and divider)
- Module top in Verilog is interface to outside world
- memory-mapped I/O port decoding (e.g. timer, LED, SPI, GPIO)

RISC0 SPI Communication

- shift registers implemented in hardware (used for both in and out)
- control signals manipulated by software (e.g. CS)
- poll status bit for operation complete (or Interrupt)
- input available at end of operation
- RISC0 SPI simpler than e.g. typical ARM SoC implementation

Project Oberon

Features

- fast file system, module loader and garbage-collector
- high-level, type-safe language for applications and implementation
- graphical, tiled window interface
- mouse oriented:
  - command (middle button), e.g. execute command Edit.Open

[10] Negative, Zero, Carry, Overflow

    - select (right button), e.g. select parameter (^)
    - point (left button), e.g. set caret
- vertical user and system tracks for placing views
- *tool* views for organizing commands
- on-screen diagnostic LEDs

Structure

- inner core: memory, files and module loader
    - Kernel.Mod: memory allocation and garbage collection, disk sectors
    - FileDir.Mod: flat file directory and garbage collection
    - Files.Mod: files as byte streams, "riders" for access
    - Modules.Mod: (recursively) load object code, execute commands
- outer core: viewer and task management
    - Input.Mod, Display.Mod: drivers for keyboard, mouse and screen
    - Viewers.Mod: division of screen into *tracks* and *viewers*
    - Fonts.Mod: font file and glyph management
    - Texts.Mod: text file manipulation and scanning
    - Oberon.Mod: task loop and command execution
    - MenuViewers.Mod: viewers with title and menu of commands
    - TextFrames.Mod: rendering and editing of text
    - System.Mod, Edit.Mod: commands for viewers/system, text editing
- applications: compiler, graphics editor, networking
    - OR[SBGP].Mod: self-hosted Oberon compiler
    - Graphics.Mod, GraphicFrames.Mod etc.: graphics editor
    - Net.Mod, Print*.Mod: optional network and printing system
    - file xfer, mail, instant messaging, time synch, remote printing uses SCC.Mod: twisted-pair network device driver (now 2.4GHz wireless)
- simple user application stars
    - bounce stars around a menu viewer
    - module body: create background task with handler (handler is called on each interval)
    - `Open`: create viewer and install into display hierarchy (menu etc.)
    - `Handle`: interpret viewer messages (and draws stars if step message is received)
    - `Step1`: broadcast *Step* message to advance viewer display (gets called by background task)
    - `Run` and `Stop`: install and remove background task

Project Oberon on RISC

- revival of the original system described in the book
- RISC5 processor replaces defunct NS32032
- complete system on 1MB RAM Spartan-3 board + daughterboard (SD-Card)
- simple 2.4GHz Nordic SPI wireless module replaces twisted-pair network
- now on compact custom-designed FPGA board
- From RISC0 to RISC5
    - optional floating-point
    - external asynchronous static RAM interface
    - enhanced fast SPI for SD-card/flash and network
    - PS/2 keyboard & mouse
    - 1024x768x1bpp video output via VGA connector
    - extra features only around 250-350 more lines of Verilog (RISC0 is around 265 lines)

- Oberon Memory Map and I/O

```
[000000H - 0000FFH] 256 bytes system area incl. module table
[000100H - 07FFFFH] 512K module block area and stack
[080000H - 0E7EFFH] 425K heap
[0E7F00H - 0FFEFFH]  96K video framebuffer
[0FFFC0H - 0FFFFFH]   64 bytes memory-mapped I/O
FFFFC0 (-64)  milliseconds (rd) / -- (wr)
FFFFC4 (-60)   -- / "LEDs" (last line of on-screen diagnostics!)
FFFFD0 (-48)  SPI data / SPI data (start)
FFFFD4 (-44)  SPI status / SPI control
FFFFD8 (-40)  PS2 mouse data, keyboard status / --
FFFFDC (-36)  keyboard data / --
FFFFE0/4 (-32/-28) general-purpose I/O data/ctrl
```

- FPGA Board USB DFU *Load-and-go*
  - uses standard USB Downloadable Firmware Upgrade (DFU) protocol 1.1
  - USB device interface on FPGA board is Microchip PIC16LF1459
  - PICs are simple microcontrollers similar to Atmel AVR in Arduino
  - 8-bit data, 14-bit instruction set, ~40 instructions
  - on-chip peripherals including UART, SPI etc, A/D, D/A, timers
  - programmed using Wirth's PICL language, DFU handler is 325 lines
  - receives bitfile from USB, reverses bits & sends directly into FPGA
  - flash upload facility (e.g. to copy Oberon filesystem back to PC)
  - RISC program (and/or other data) can be appended to bitfile
  - captured by FPGA and written directly to SRAM before startup
- Booting/Installing the Oberon System
  1. bitfile either comes over USB via DFU, or is loaded from start of flash
  2. FPGA loads appended inner core from DFU, or from fixed area of flash
     a. (ROM bootloader in Project Oberon no longer needed - simplifies CPU)
  3. system starts at 000000H, e.g. with jump to inner core init
  4. initialises Kernel, Files, Modules
  5. loads Oberon and dependent modules and runs Oberon.Loop
- Nordic nRF24L01+ 2.4GHz SPI Wireless Network Transceiver
  - 1Mb/s wireless to replace low-cost 230kb/s twisted-pair wiring
  - nRF24L01 replaces original serial comms controller (SCC)
  - simple packet frame format with up to 32 bytes of payload
  - straightforward SPI command interface, FIFOs for rx and tx packets
  - optional hardware acknowledge and re-transmit of dropped frames

FPGA Mouse SW/HW Co-design Example

- typical PS/2 (or USB) mouse operation involves separation of SW and HW duties
- hardware receives data serially into shift register
- movement packet transmitted byte-by-byte with start, parity and stop bits
- byte reception interrupts processor, which stores bytes in a buffer
- software interprets packets and updates mouse pointer position
- software needs to ensure buffer does not become full (seen in practice)
- is there a better way? Yes, PS/2 Mouse Driver in Hardware
  - reports current mouse position to software
  - receive entire PS/2 packet into a single shift register
  - pick out the needed bits, discard others
  - later even enhanced to handle Microsoft wheel button enable

Video (VGA) Interfacing - Worked Example

- how is a video signal produced? VGA (analogue) vs HDMI (digital)
- worked example of an application implemented solely in hardware
- introducing Lola, Niklaus Wirth's HDL (demo)

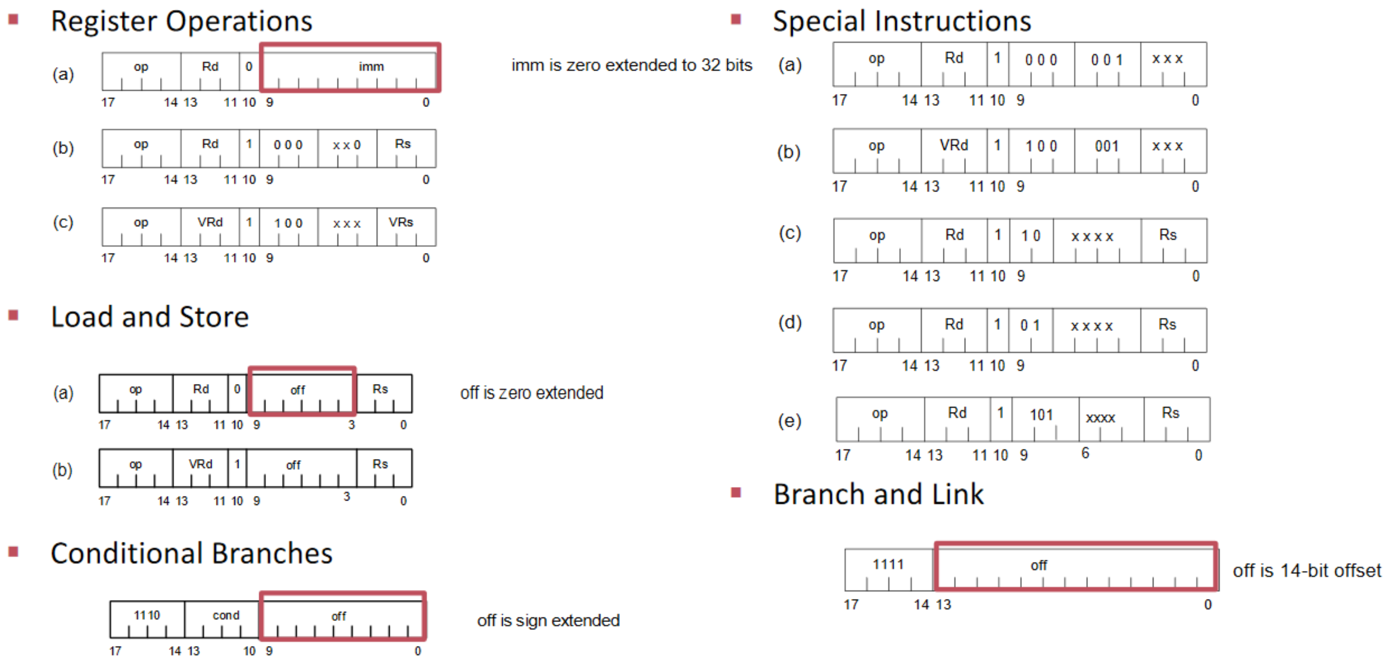
1. generate horizontal & vertical sync and fixed pattern
2. create testbed for simulation - VIDExTest.v
3. add border and cursor
4. input I2S data and store in BRAM (block ram included in FPGA, can be access every cycle)
5. display BRAM data

Case Study 4: Active Cells

- In a general-purpose computer, each core uses shared bus/memory and processes are assigned to a core
- This involves several challenges:
  - Multicore:
    - cache coherence
    - shared memory is bottleneck (difficult to scale design to massive multi-core architecture)
    - thread synchronisation overhead (hard to predict program performance)
  - Operating System:
    - process time sharing (interrupts, context switches, thread synchronisation)
    - memory sharing (inter-process with paging, intra-process/inter-thread with monitor)
- An application specific multicore network on chip can solve these problems
- We focus on streaming applications
  - given data stream a series of operations is applied to each element in the stream
  - with this model we can execute tasks in parallel, use pipelining and data parallelism (vector computing)

Hardware Building Blocks

- The central building block is the Tiny Register Machine (TRM)
  - extremely simple processor on FPGA with Harvard architecture (code/data separated)
  - two-stage pipeline (fetch, execute)
  - no caches
  - 4 conditional register (C, N, V, Z)
  - Register H for storing the high 32 bits of a product
  - 7 general purpose registers (32-bit)
- TRM Machine Language:
  - 18/16-bit instructions
  - Three instruction types:
    - arithmetical and logical operations
    - load and store instructions
    - branch instructions
  - As we have a small instruction size we cannot jump to all addresses with the offset (we have to make two jumps to cover larger distance with second offset)



- Register Operations

imm is zero extended to 32 bits

- Load and Store

off is zero extended

- Conditional Branches

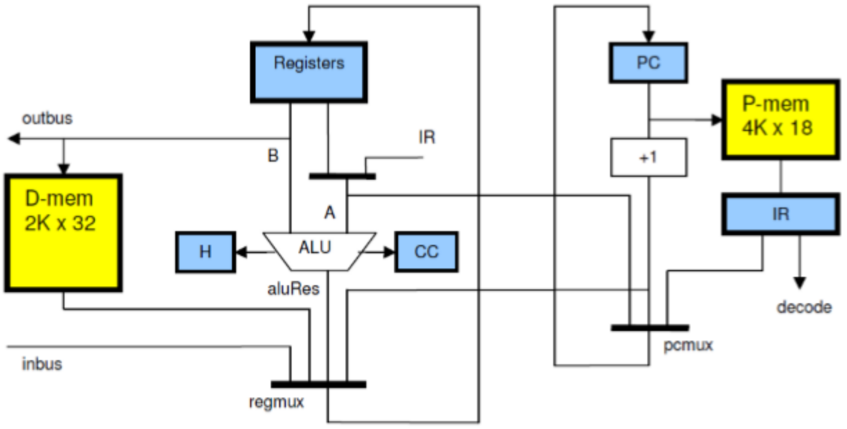
off is sign extended

- Special Instructions
(a)
op | Rd | 1 | 0 0 0 | 0 0 1 | x x x
17 14 13 11 10 9 0
(b)
op | VRd | 1 | 1 0 0 | 001 | x x x
17 14 13 11 10 9 0
(c)
op | Rd | 1 | 1 0 | x x x x | Rs
17 14 13 11 10 9 0
(d)
op | Rd | 1 | 0 1 | x x x x | Rs
17 14 13 11 10 9 0
(e)
op | Rd | 1 | 101 | xxxx | Rs
17 14 13 11 10 9 6 0

- Branch and Link

off is 14-bit offset

- TRM architecture



- Variants of TRM:
  - FTRM (includes floating-point unit)
  - VTRM (includes vector processing unit, with/without FP unit)
  - TRM with software configurable instruction width (smaller instructions more code density)
- One can connect TRMs with different topologies
  - Bus is faster but also more complicated (fills the while FPGA)
  - Ring is slower but architecture much simpler

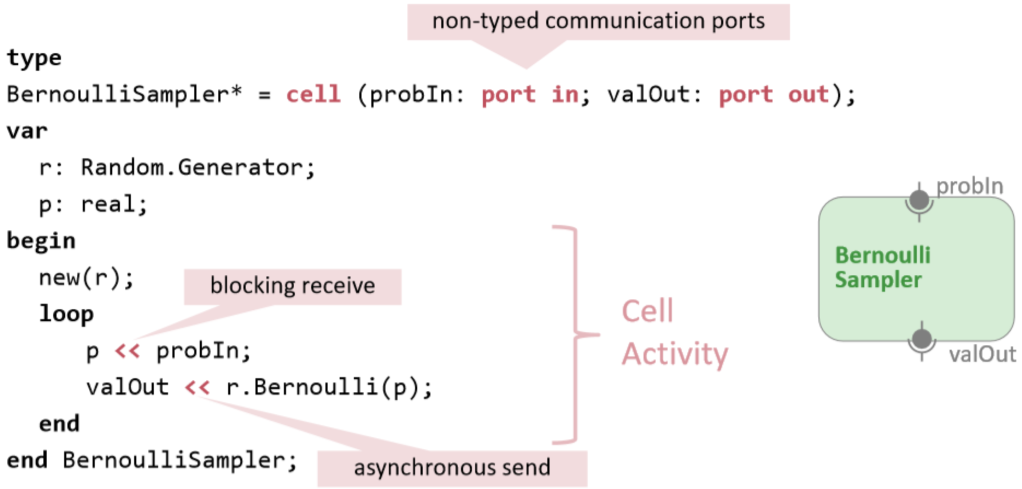
Hardware Software Codesign

- traditional HW/SW codesign used separate toolchains for software (C/C++) and hardware (Verilog) development
- Active Cells provides one toolchain for both worlds
- Active Cells computing model consists of
  - Cells
    - scope and environment for a running isolated process
    - integrated control thread
    - provides communication ports
  - Net
    - Network of communication cells
    - Cells connected via channels (FIFOs)

- A streaming application is described with cells and channels in high-level code (toolchain places network on FPGA)
- Consequences of this approach:
    - no global memory
    - no processor sharing
    - no predefined topology (Network on Chip)
    - no interrupts
    - no OS (works only for streaming application)

Cell

- each cell can be augmented with properties which can influence hardware synthesis (e.g. FPU, data memory, program memory) and software generation (different instruction size)



```
type
BernoulliSampler* = cell (probIn: port in; valOut: port out);
var
   r: Random.Generator;
   p: real;
begin
   new(r);
   loop
      p << probIn;
      valOut << r.Bernoulli(p);
   end
end BernoulliSampler;
```

non-typed communication ports

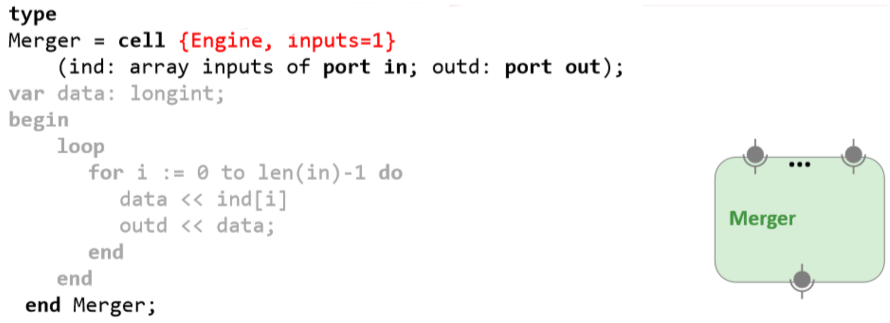
blocking receive

asynchronous send

Cell Activity

Engine

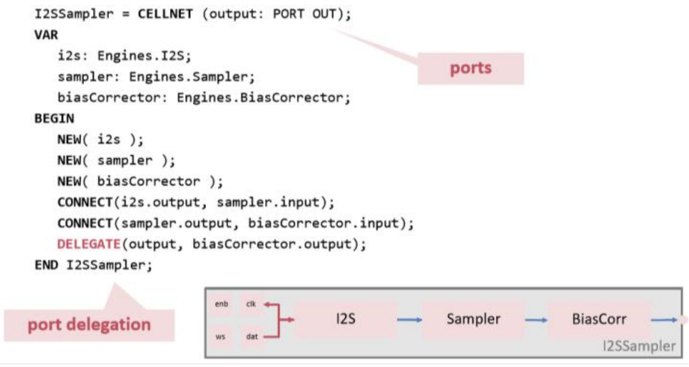
- prebuilt components instantiated as electronic circuits on target device (e.g. written in Verilog)



```
type
Merger = cell {Engine, inputs=1}
    (ind: array inputs of port in; outd: port out);
var data: longint;
begin
    loop
      for i := 0 to len(in)-1 do
        data << ind[i]
        outd << data;
      end
    end
 end Merger;
```

Cellnet

- Two types:
    - non-terminal cellnet: has I/O-ports and contains various cells/engines
    - terminal cellnet: no ports and is unit of deployment (connects non-terminal cellnets and cells/engines)



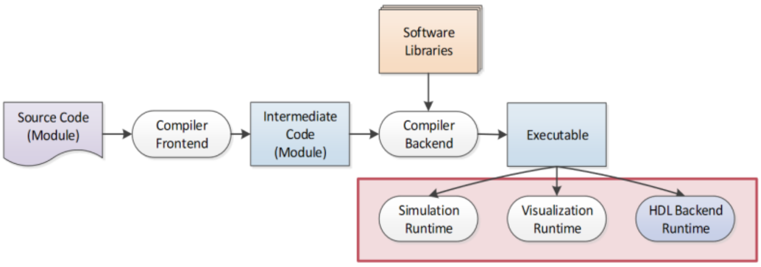
```
I2SSampler = CELLNET (output: PORT OUT);
VAR
   i2s: Engines.I2S;
   sampler: Engines.Sampler;
   biasCorrector: Engines.BiasCorrector;
BEGIN
   NEW( i2s );
   NEW( sampler );
   NEW( biasCorrector );
   CONNECT(i2s.output, sampler.input);
   CONNECT(sampler.output, biasCorrector.input);
   DELEGATE(output, biasCorrector.output);
END I2SSampler;
```

ports

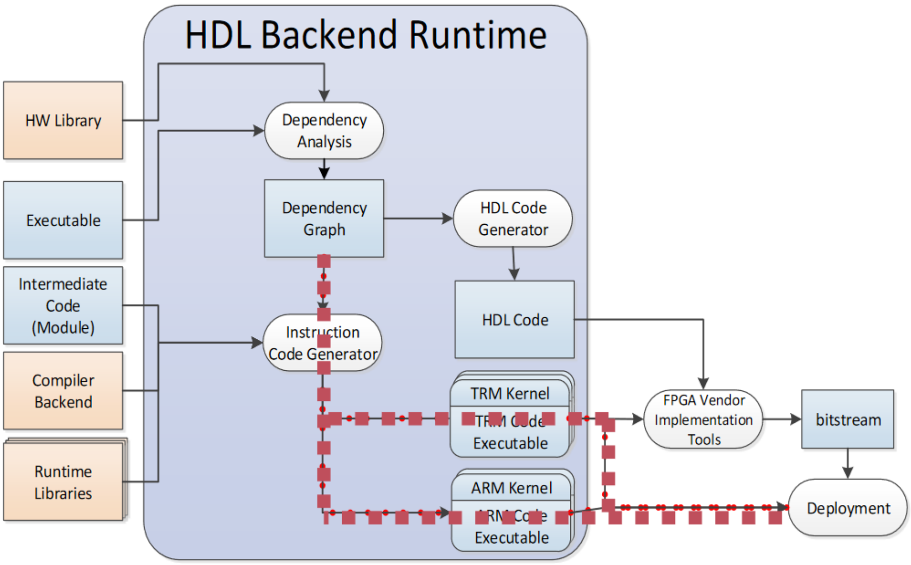
port delegation

Software to Hardware

- In order to map the cellnet to hardware we must compile each part specifically
  - Cell (Progam Logic) gets compiled as normal software to binary code
  - Engine and cellnet needs hardware synthesis
- we can use the following approaches to achieve that:
  - a compiler frontend takes the source code and a backend for each board creates the bit stream
    - simple but many redundancies, not flexible and hard to extend
  - frontend creates code for a cellnet interpreter (backend)
    - interpreter executes cellnet during compilation and finds out which port goes where etc.
    - interpreter uses hardware/component library to create bitstream
    - extensible and no redundancies thanks to libraries but not simple and static configuration limits flexibility
  - frontend creates Oberon executable which runs on different runtimes (best solution)
    - a runtime could be a HDL generator which uses libraries to create bit streams
    - it is extensible, not redundant and simpler
- Compiler frontend creates an executable which can run on different runtimes (e.g. CONNECT can be simulated by a simulation runtime or a Verilog runtime generates a hardware connection)

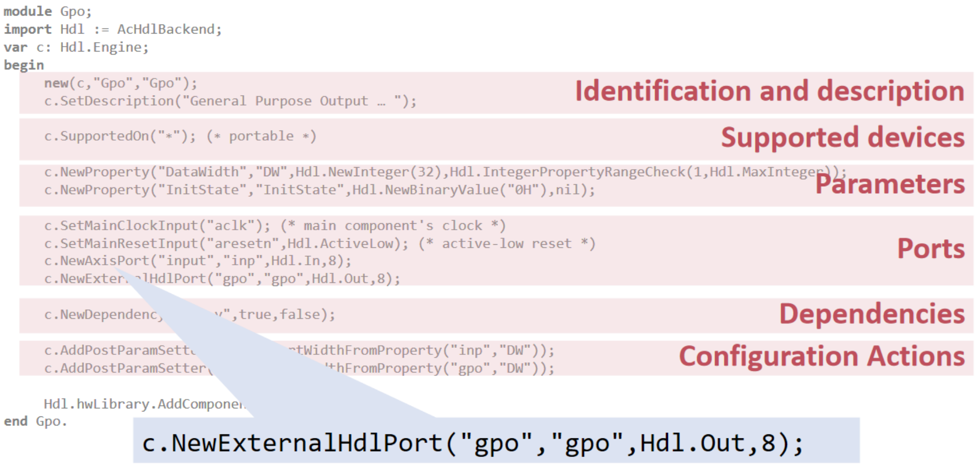


- The HDL backend runtime creates HDL and kernel
  - we need the intermediate code to compile program for our specific architecture
  - bitstream is generated using FPGA vendor tools
  - no re-synthesis if hardware didn't change



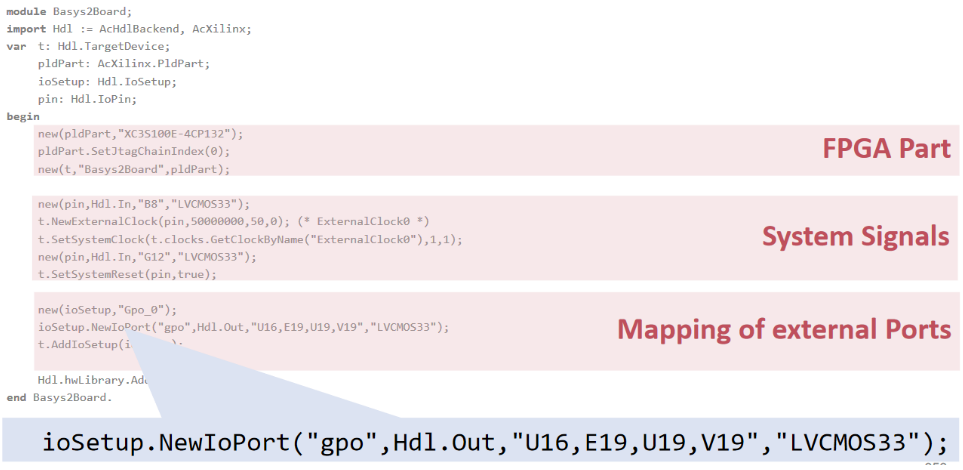
- we can extend our system by defining components and platforms
  - component specifications map Engines to HDL code (Verilog)
  - platform specifications help the build tool to find correct PINs, vendor tools etc.

- In this example we define a Gpo engine and use platform specification to map actual pins



```
module Gpo;
import Hdl := AcHdlBackend;
var c: Hdl.Engine;
begin
    new(c,"Gpo","Gpo");
    c.SetDescription("General Purpose Output … ");

    c.SupportedOn("*"); (* portable *)

    c.NewProperty("DataWidth","DW",Hdl.NewInteger(32),Hdl.IntegerPropertyRangeCheck(1,Hdl.MaxInteger));
    c.NewProperty("InitState","InitState",Hdl.NewBinaryValue("0H"),nil);

    c.SetMainClockInput("aclk"); (* main component's clock *)
    c.SetMainResetInput("aresetn",Hdl.ActiveLow); (* active-low reset *)
    c.NewAxisPort("input","inp",Hdl.In,8);
    c.NewExter  lHdlPort("gpo","gpo",Hdl.Out,8);

    c.NewDependenc          ",true,false);

    c.AddPostParamSett         tWidthFromProperty("inp","DW"));
    c.AddPostParamSetter(          thFromProperty("gpo","DW"));

    Hdl.hwLibrary.AddComponen
end Gpo.
```

```
c.NewExternalHdlPort("gpo","gpo",Hdl.Out,8);
```



```
module Basys2Board;
import Hdl := AcHdlBackend, AcXilinx;
var  t: Hdl.TargetDevice;
     pldPart: AcXilinx.PldPart;
     ioSetup: Hdl.IoSetup;
     pin: Hdl.IoPin;
begin
     new(pldPart,"XC3S100E-4CP132");
     pldPart.SetJtagChainIndex(0);
     new(t,"Basys2Board",pldPart);

     new(pin,Hdl.In,"B8","LVCMOS33");
     t.NewExternalClock(pin,50000000,50,0); (* ExternalClock0 *)
     t.SetSystemClock(t.clocks.GetClockByName("ExternalClock0"),1,1);
     new(pin,Hdl.In,"G12","LVCMOS33");
     t.SetSystemReset(pin,true);

     new(ioSetup,"Gpo_0");
     ioSetup.NewIoPort("gpo",Hdl.Out,"U16,E19,U19,V19","LVCMOS33");
     t.AddIoSetup(i     );

     Hdl.hwLibrary.Ad
end Basys2Board.
```

```
ioSetup.NewIoPort("gpo",Hdl.Out,"U16,E19,U19,V19","LVCMOS33");
```

- ARM AXI4 standard is used to communicate between the different parts on the chip
    - simple handshake: Sender keeps TVALID high (data is ready to send), receiver keeps TREADY high (ready to process), if both signals are high during clock cycle data is considered transferred

Case Studies

- Several case studies showed the following advantages of Active Cells:
    - Configurable interconnect -> Simple Computing, Power Saving
    - Embedding of task engines -> High Performance
    - Hybrid compilation -> Quick Development
    - Backend execution -> Eased Flexibility and Extensibility