Development of an Aos Operating System for the DNARD Network Computer

Diploma Thesis SS 2001
Department of Computer Science
ETH Zürich

Bernhard Egger
August 2001

Diploma Professor: Prof. Dr. Jürg Gutknecht
Supervising Assistants: Pieter Muller and Patrik Reali

Abstract

This report describes the results of the development of an Aos operating system for the DNARD network computer.

Aos is a small and efficient operating system, thus well suited for mobile and wireless devices. The DNARD port can be used as a basis for further Aos ports to devices that use the high-performance Intel StrongARM SA-110 processor (or one of its successors).

The port started with the development of an ARM backend to Paco, the parallel compiler for Aos, including a fully featured ARM assembler.

Once the compiler was able to generate native ARM code, first the Aos kernel with its locking scheme, garbage collector and automatic process synchronization, and finally the whole Aos system including all applications were ported to the DNARD platform.

This report can also be used as a guideline on how a port of Aos to another platform could be approached.

Table of Contents

CHAPTER 1

Introduction

The goal of this diploma thesis was to implement a version of the Aos operating system [Aos01] for the DNARD Network Computer (a.k.a. Shark) [DEC97].

1.1 Task

The Aos operating system consists of several components:

- Active object run-time system
- Active Oberon compiler
- Device driver framework and display, keyboard, mouse and ethernet drivers
- File system framework and several file systems
- TCP/IP and several services
- Window manager and XML browser
- Oberon system and applications

A substantial part of the work was the development of an Active Oberon cross-compiler for the Intel StrongARM processor [Intel], which is an implementation of the ARM processor architecture [ARM], and the development of an Active Oberon run-time system for the DNARD.

The basis for the development was the Intel IA-32 PC-based version (called the “Intel version” in this document) of the Aos operating system being developed at the ETH Zürich Computer Systems Institute [Mull00],[AosOS]. This implementation includes a run-time system for Intel 32-bit multiprocessor machines, and a parallel

compiler for the Active Oberon language producing Intel 32-bit code [Rea00]. These two components had to be adapted for the DNARD environment.

The development of device drivers for the DNARD was based on the SharkOberon system that was developed in an earlier diploma thesis [Sig99].

As the outcome of this project could form the basis of further Aos ports to mobile devices and embedded systems, conceptual clarity and simplicity, reliability and efficiency were all important attributes strived for in the implementation.

1.2 Motivation

The Aos operating system is the latest descendant of the Oberon operating system family [WG92].

Aos is the first Oberon operating system that supports symmetric multiprocessing (SMP). Key features of Aos are:

- a small and efficient kernel
- automatic garbage collection
- support for active objects
- easy inter-process synchronization

Due to its modest requirements, Aos is well suited for mobile and wireless devices with limited resources.

Mobile devices get more important every day. The personal computer is slowly disappearing and will be replaced by many small, mobile and wireless appliances. Therefore it would be appropriate to have an Aos operating system that runs on mobile devices. Many of these devices feature an Intel StrongARM processor [Intel],[ARM] because of its high performance, small size and low power requirements.

The DNARD Network Computer [DEC97] was chosen for this diploma thesis, because it is an ideal platform for the first port of the Aos operating system due to its PC-style architecture.

CHAPTER 2

StrongARM SA-110

This chapter gives a short overview of the processor architecture of the Intel StrongARM SA-110 processor. If you are familiar with ARM architectures, you may want to go directly to chapter 3.

2.1 Architecture

The SA-110 is an implementation of Advanced RISC Machines Ltd. (ARM) Version 4 architecture [Intel],[ARM].

It is a general-purpose, 32-bit microprocessor with a five-stage instruction pipeline, 16Kbyte instruction cache, 16Kbyte write-back data cache, an 8-entry write buffer with 16 bytes per entry, and a memory-management unit (MMU) with two 32-entry fully associative translation lookaside buffers (TLB) in one single chip.

The SA-110 runs at CPU clock speeds from 100 to 233 MHz and has a low power consumption. Thus, the SA-110 (and its successors, the SA-1100 and the SA-1110) are well suited for portable products such as personal digital assistants (PDAs), cell-phones, organizers and many more.

2.1.1 Registers and Processor Modes

The SA-110 has a total of 37 registers.

- 31 general-purpose 32-bit registers. At any one time, only 16 of these registers are visible. The other registers are used to speed up exception processing.

- 6 status registers

The SA-110 supports five types of exception modes, a privileged system mode and an unprivileged user mode. Normally, the SA-110 runs in User or System mode which both have the same register set. If an exception occurs, the respective exception mode is entered and some of the User/System registers are replaced (“banked”) by their equivalent for the exception mode.

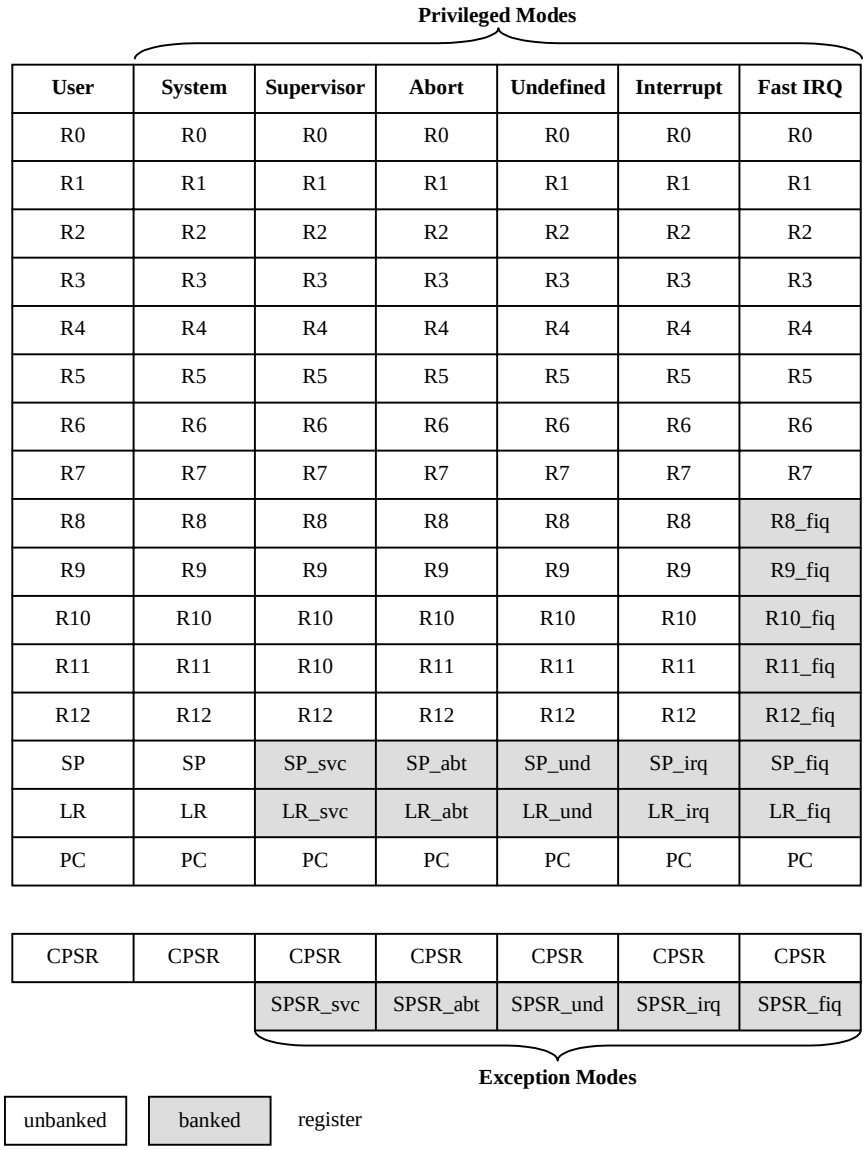

Figure 2-1. Register organization

- R0-R7 are general-purpose *unbanked registers*. This means that each of them refers to the same physical register in all processor modes.
- R8-R14 are *banked registers*. The physical register referred to by each of them depends on the processor mode.

- R13 is normally used as a *stack pointer (SP)*.
- R14 is the *link register (LR)*.
- R15 is the *program counter (PC)*. The PC is always two instructions ahead. This means that if the SA-110 is executing the instruction at address 0000H, the value of the PC is 0008H.Writing a value to the PC causes a branch.
- CPSR is the *current*, SPSR the *saved program status register.* The CPSR is automatically saved in the SPSR when the CPU switches to an exception mode. The PSR contains the *condition code flags*, the *interrupt enable bit* and the *current processor mode*.

2.1.2 Instruction Set

The instruction set can be divided into six broad classes of instructions:

- branch instructions
- data-processing instructions
- status register transfer instructions
- load and store instructions
- coprocessor instructions
- exception-generating instructions

2.1.2.1 Branch instructions

A standard branch instruction with a 24-bit signed offset is provided, allowing forward and backward branches of up to 32MB. Longer branches can be achieved by writing a register value directly into the program counter.

The Branch and Link (BL) option preserves the address of the instruction following the branch in the LR register. This provides a subroutine call which can be returned from by copying the LR into the PC.

2.1.2.2 Data-processing instructions

Data-processing instructions perform calculations on the general-purpose registers. There are three types of data-processing instructions:

- *arithmetic/logic instructions* have a destination register where the result of an arithmetic or logical operation of up to two source operands is written to. Optionally, they can also update the condition code flags based on the result.
- *comparison instructions* always update the condition code flags and do not have a destination register. Otherwise, they work exactly like arithmetic/logic instructions.

- *multiply instructions* multiply two 32-bit register values and store their result either in one 32-bit register or in two separate registers, thus providing a 64-bit result. Both types can optionally perform an accumulate operation.

2.1.2.3 Status register transfer instructions

These instructions transfer the contents of the CPSR or a SPSR to or from a general purpose register. Writing to the CPSR can:

- set the values of the condition codes
- set the value of the interrupt enable bit
- set the processor mode

2.1.2.4 Load and store instructions

Load and store instructions can move a 32-bit word, a 16-bit halfword or an 8-bit byte from or to memory. Load and store instructions have three primary addressing modes. Each uses a *base register* and an *offset* to specify the memory location.

- in *offset addressing*, the memory address is formed by adding/subtracting the offset to or from the base register value
- in *pre-indexed addressing*, the memory address is formed in the same way as for offset addressing, but the calculated memory address is also written back to the base register.
- in *post-indexed addressing*, the memory address is the base register value. As a side-effect, the offset is added to or subtracted from the base register and the calculated value written back to the base register.

Unfortunately, the load and store instructions with immediate offsets are not orthogonal. For word or unsigned byte accesses, a 12-bit unsigned immediate can be added to or subtracted from the base register. For halfword or signed byte accesses, the offset is only 8-bit wide.

Load Multiple (LDM) and Store Multiple (STM) instructions perform a block transfer of any number of the general-purpose registers to or from memory. As in the case of simple load and store, the base register can be incremented or decremented before or after the access.

The Swap (SWP) instruction allows an atomic swap of a register value and a memory location. This instruction can be used to update semaphores.

2.1.2.5 Coprocessor instructions

Coprocessor instructions interact with one of the optional coprocessors. By default, only coprocessor 15 (MMU) is present.

Values can be transferred from a general-purpose register to a coprocessor register and vice-versa.

2.1.2.6 Exception-generating instructions

There are two special instructions that cause a specific exception to occur:

- The software interrupt instruction (SWI) causes a software interrupt
- The software breakpoint instruction (BKPT) causes an abort exception and can be used by a debugger. This instruction is only available on ARM Version 5.

In addition, any encoding that has no meaning as an ARM instruction causes an “undefined instruction” exception.

2.1.3 Exceptions

Exceptions are generated by internal and external sources to cause the processor to handle an event, such as an attempt to access an invalid memory cell. The processor state just before entering the exception handler must be preserved so that it can be restored when the original program is resumed.

When an exception occurs, execution is continued at a fixed memory address corresponding to the type of the exception. On the SA-110 these *exception vectors* are located at 00000000H-0000001CH. The exception vector table contains one instruction for each exception type, which is usually a branch instruction to the exception handler.

According to [ARM96], some ARM implementations allow the exception vector locations to be moved to an alternative address range FFFF0000H-FFFF001C. Unfortunately this is not the case on the SA-110. This imposes some problems for detection of NIL pointer de-referencing.

The current StrongARM processor, the SA-1100, supports high vectors.

2.1.4 Floating-Point support

The SA-110 does not have a floating point unit nor does it support floating-point instructions. If floating-point support is needed, it must be emulated in software.

2.2 A closer look at the instruction set

In this section, some features of the ARM instruction set are inspected in detail.

2.2.1 Conditional codes and conditional execution

The CPU supports *conditional execution* for every instruction. The processor executes the instruction only if the flags in the CPSR satisfy the condition in the opcode.

The condition is always encoded in bits 31:28 of the instruction's opcode:

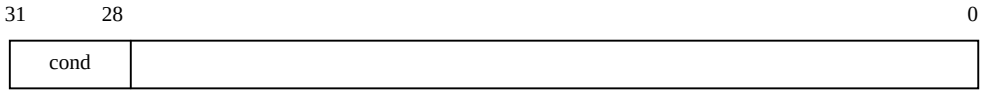

Figure 2-2. Encoding of the condition

One of the sixteen possible combinations in the *cond* field is the *unconditional execution*, i.e., the instruction is executed regardless of the current condition code flags.

The condition code flags in the CPSR are usually modified by

- execution of a comparison instruction.
- execution of a arithmetic, logical or move instruction. Most of these instructions have a flag-setting and a flag-preserving variant.

The condition code flags in the PSR is composed of four bits that usually mean

N	is set to bit 31 of the result of the instruction. If the result is a two's complement signed integer, then N=1 if the result is negative and N=0 if the result is zero or positive
Z	is set to 1 if the result of the instruction is zero, 0 otherwise
C	is set in one of four ways: • for an addition, C=1 if the addition produced a carry • for a subtraction, C=0 if the subtraction produced a borrow (an unsigned underflow) • for instructions that incorporate a shift (excluding addition and subtraction), C is set to the last bit shifted out by the shifter • for any other instruction, C is normally left unchanged
V	is set in one of two ways: • for an addition/subtraction, V=1 if a signed overflow occurred • for any other instruction, V is normally left unchanged

Table 2-1. Condition code flags

2.2.2 Data-processing instructions

Data-processing instructions have one of three forms

- $rD \leftarrow rA \otimes rB$ (*dyadic operators*)
- $rD \leftarrow op(rA)$ (*monadic operators*)
- $rA \otimes rB$ (*comparison operators*)

Data-processing instructions have the following encoding:

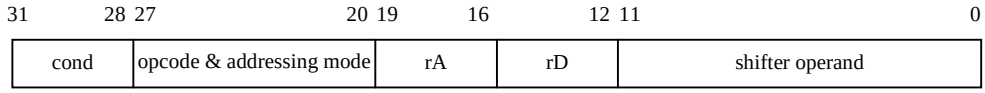

Figure 2-3. Data-processing instruction encoding

rD and rA denote the *destination* and the *source register* respectively. The second source B is encoded in the *shifter operand*, which takes one of three basic forms. The *addressing mode* in combination with some bits of the shifter operand let the CPU determine what form is used. The three basic forms are:

Immediate operand value

The immediate operand value is formed by rotating an 8-bit constant by an even number of bits, encoded in the remaining four bits of the shifter operand.

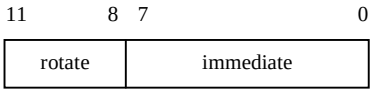

Figure 2-4. Immediate encoding in shifter operand

Therefore, only a few immediate constants can be generated. Some valid constants and their decompositions are

```
FFh       rotate: 0     immediate: FFh
204h      rotate: 1     immediate: 81h
FF0000h   rotate: 8     immediate: FFh
F000000Fh rotate: 14    immediate: FFh
```

(remember that the immediate value is rotated left by 2*rotate!)

Register operand value

A register operand value is simply the value of the register, encoded in the lowest four bits of the shifter operand.

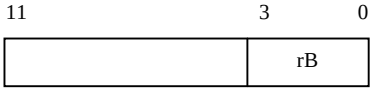

Figure 2-5. Register encoding in shifter operand

Shifted register operand value

A shifted register operand value is the value of the register shifted or rotated by an immediate value or another register value before it is used. The encodings are:

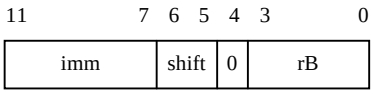

Figure 2-6. Register shifted by immediate encoding

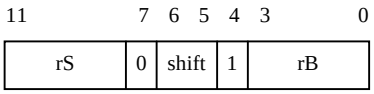

Figure 2-7. Register shifted by register encoding

This variety of encodings allows very efficient and powerful calculations. A good example are array index calculations where we can calculate the address of the element in two instead of three instructions:

```
base    <- base address
offset  <- index shifted by log2(element size)
address <- base + offset
```

becomes

```
base    <- base address
address <- base + index shifted by log2(element size)
```

in assembler, assuming `rIndex` already contains the index and the element is 4 bytes wide

```
LDR  rBase, [some memory location]
ADD  rAddr, rBase, rIndex, LSL #2
```

2.2.3 Load and store instructions

Load and store instructions are divided into two classes

- load and store a 32-bit word or an 8-bit unsigned byte

- load and store a 16-bit halfword (signed/unsigned) or load an 8-bit signed byte. Signed data types are automatically sign extended when loaded into a register, i.e., the most significant bit of the value is propagated left in the register.

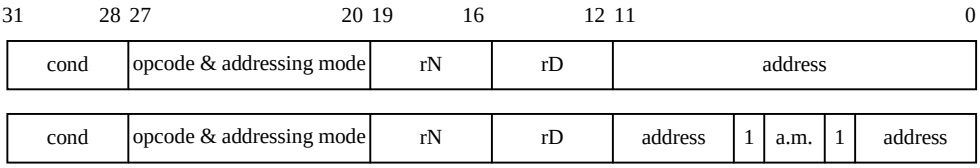

Figure 2-8. Load and store encodings. 32-bit word/8-bit unsigned above, 16-bit half-word /8-bit signed below

In both types, the address of the memory cell is formed from two parts, the *base register* (rN) and an *offset* (address). The base register can be any of the registers (including the PC), whereas the offset takes one of three forms:

Immediate offset

The 12-/8-bit unsigned immediate is placed in the address. A bit in the addressing mode decides whether the offset is added to or subtracted from the base register to form the memory address.

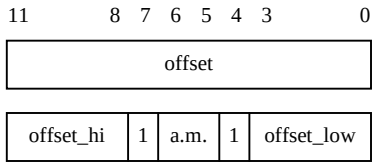

Figure 2-9. Immediate offset encoding. 32-bit word/8-bit unsigned above, 16-bit half-word /8-bit signed below

Register offset

The memory address is formed by adding or subtracting the value of the offset register to or from the base register. The register is placed in bits 3:0 of the address.

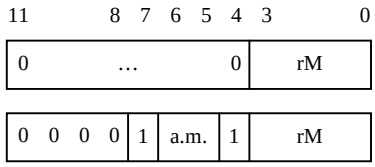

Figure 2-10. Register offset encoding. 32-bit word/8-bit unsigned above, 16-bit half-word /8-bit signed below

Scaled register offset

In this addressing mode, the value of the offset register is shifted or rotated by a 5-bit unsigned immediate and then added to or subtracted from the base register. This addressing mode is not available for halfword or signed byte transfers.

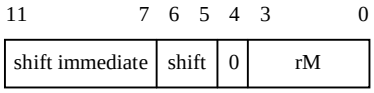

Figure 2-11. Scaled register offset encoding

Again, we are able to save one instruction. In our example of addressing an array element we get:

```
base    <- base address
offset  <- index shifted by log2(size of element)
element <- MEM[base + offset]
```

becomes

```
base    <- base address
element <- MEM[base address + index shifted by log2(elem.size)]
```

in assembler, assuming the index is stored in rIndex and the size of one element equals 4

```
LDR  rBase, [some memory location]
LDR  rElem, [rBase, rIndex, LSL #2]
```

In addition to the three addressing modes, we can define the time order of the memory access and the address calculation (Section 2.1.2.4 on page 6).
This is very useful to implement a *push* instruction which does not exist as such on ARM architectures. Using pre-indexed addressing we get:

```
SP <- SP -4,  MEM[SP] <- register
```

The assembler code for this instruction looks like this

```
STR  reg, [SP, #-4]!    ; first, use SP minus 4 as the memory
                        ; address. Then store the value of
                        ; ‘reg’ to MEM[address] and write the
                        ; memory address back to the SP
```

CHAPTER 3

ARM Backend for Paco

This chapter describes the pluggable ARM backend for Paco [Rea00] in detail.

3.1 Paco

Paco (short for Parallel Compiler) is the latest compiler for Aos. It makes ample use of Aos' features such as active objects and easy process synchronization.

When Paco compiles a source file, the main parser that parses the module body forks off a new parser for each encountered *scope*, i.e. for procedures and records. These parsers have their own local symbol table with symbols only valid within their procedure or record scope respectively. The access to the global symbol table is protected to avoid corruption due to a possible race condition. The main advantage of parallelism in the compiler is that forward references that would usually require a second parser pass can be reduced to a scope-synchronization problem.

The main process waits for each parser to finish and then passes the generated intermediate representation of the source code to the backend. The backend then translates the intermediate representation into executable code. Lastly the object file plug-in serializes the code and the necessary linking information into an object file.

3.1.1 Structure

Paco clearly distinguishes between *front-end* and *back-end*. The front-end reads and parses the source code and transforms it into an *abstract representation* (see "Intermediate Language" on page 15). The back-end takes this abstract representation of

the source code and transforms it into executable machine code. This design decouples the machine-independent parsing process from the machine-dependent code-generation process.

Paco is composed of eight main components:

- *Service Modules*

  These modules contain common functionality such as the StringPool module that contains functions to add and retrieve strings and is used in almost every other module.

- *Symbol Database*

  The symbol database defines symbols, structures and scopes, plus functions to create and manipulate them.

- *Oberon Parser*

  The parser is the main component of the front-end. The parser component performs lexical and semantical analysis, and translates the parsed code into the LIR defined by the abstract representation component (see below).
  Parsing is done concurrently with a scope granularity.

- *Abstract Representation*

  The abstract representation defines an assembly-like machine-independent low-level intermediate code representation (LIR) and functions to manipulate it.

  The abstract machine defined by the LIR has a load-store architecture, an infinite number of registers that are only assigned once and uses compare and branch to change the control flow.

- *Code-Generator Plug-In*

  The code-generator plug-in transforms the LIR into machine code.

- *Assembler Plug-In*

  The assembler plug-in belongs to the front end and is the only machine dependent component in the front end. Whenever the parser detects inlined assembler code it passes the parsing task to the installed assembler plug-in. This allows the use of assembly language for low-level machine-dependent procedures or in speed-critical parts of the module.

- *Symbol File Plug-In*

  The symbol file plug-in serializes the information from the symbol database to permit persistency of symbol information.

- *Object File Plug-In*

  The object file plug-in generates an object file from the compiled code with linking information.

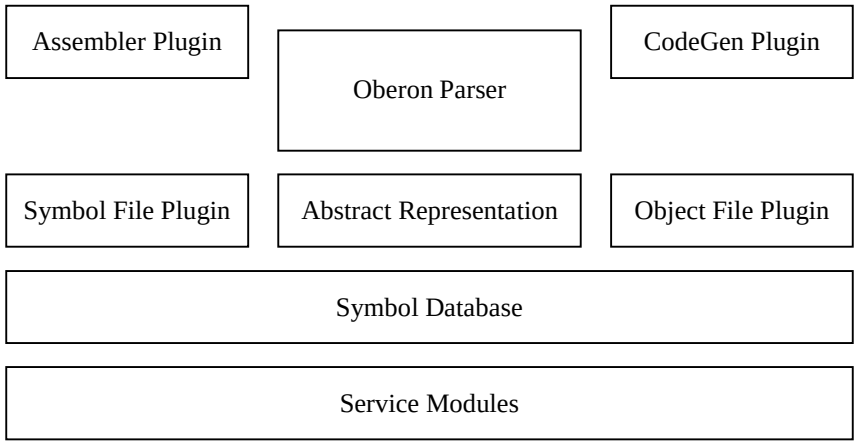

Figure 3-1. Structure of Paco

3.1.2 Intermediate Language

The low-level intermediate language (LIR) is the only interface between the front-end and the back-end. The LIR defines a load-store architecture with infinite registers. There is only one assignment to each register, similar to the static single assignment (SSA) form [Cyt91].

3.1.2.1 Registers

Registers are *typed*; each register takes one of the following types:

- *Integer types*

  Integer types are 8, 16, 32 or 64 bits wide, signed or unsigned. Note that the Oberon language currently defines only one unsigned type, CHAR, which is an 8-bit unsigned integer. All other types are signed.

- *Floating point types*

  Floating point types are either 32 or 64 bits wide and conform to the IEEE standard 754 [IEEE754]. Floating point types are always signed.

3.1.2.2 Access to variables and parameters

The LIR language does not have statements to access variables or parameters. Therefore, all accesses to them are already translated into memory addresses. See “Calling Convention and Procedure Activation Frame” on page 23.

3.1.2.3 Positive and negative aspects of the LIR

The implementation of a backend for LIR is straightforward. The language is already very low-level, so that quite a few LIR instructions can be translated one-to-one into native instructions.

Due to the fact that all accesses to variables/parameters are already translated into memory addresses, the backend must simply generate load and stores from the addresses that are defined in the LIR.

However, this simple interface imposes a few serious disadvantages:

Calling convention

It is not possible to define an architecture-specific calling convention since the access to parameters is already translated into memory addresses. Parameters must be pushed on the stack, and cannot be passed in registers, for example.

Almost-SSA form

Unfortunately, the LIR does not implement the real SSA form. Although every register receives exactly one assignment during its lifetime, the other features of SSA, such as *phi-instructions*, are missing. The optimizations that come almost for free in real SSA intermediate representations (e.g. dead-code elimination, constant folding, etc.) are therefore not possible with LIR.

Architecture dependency

Although many dependencies were removed during this work, the LIR still contains some Intel-specific instructions, such as `in` and `out`, that are not supported by the ARM backend. `In` and `out` are used to access the I/O-ports on Intel processors and are generated for the SYSTEM.PORTINT and SYSTEM.PORTOUT calls. On ARM systems, SYSTEM.GET and SYSTEM.PUT are used instead.

Another architecture dependency is that the first parameter of a procedure is accessed relative to the frame pointer (FP, see below) with offset 8. By chance this works fine on ARM processors, but it might be a problem on other architectures.

3.2 Backend Structure

The pluggable ARM backend to Paco is structured into the following modules

- PCGARM

  This is the main module of the backend. It translates the LIR into native ARM code.

- PCGARMRegisters

  This module implements the register allocator.

- PCGARMCP

  PCGARMCP is the constant and address pool service module

- PCOARM

  PCOARM contains all opcodes and bitmasks of the ARM machine code and provides an interface for each instruction to simplify the code generation. The generated machine code is temporarily stored in this module.

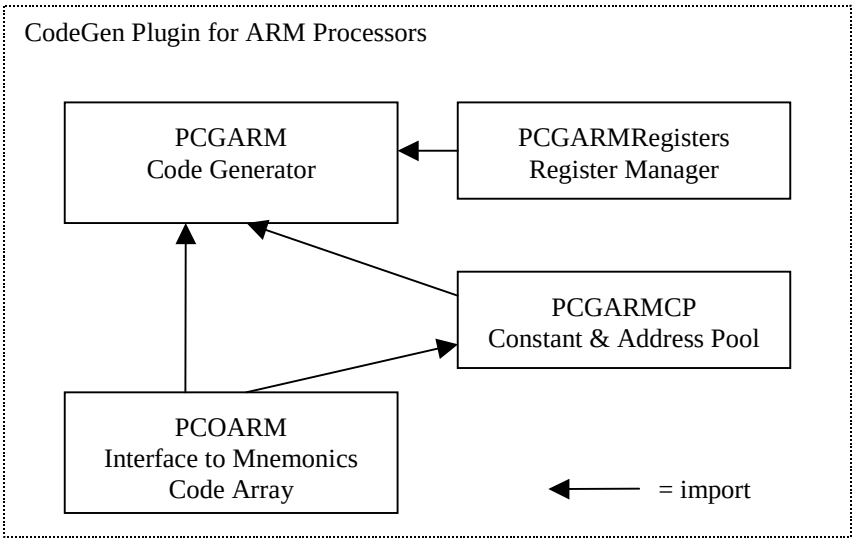

Figure 3-2. ARM Backend Structure

3.2.1 Invoking the ARM Backend

To compile a module for ARM processors, the backend and the object file suffix must be passed to Paco:

```
PC.Compile \dARM \.Oba *
```

The \d<name> installs the <name> backend. The code generator plug-in must be named PCG<name>.Mod and it must provide a procedure

```
PROCEDURE Install*;
```

that will be called by Paco. This installation procedure must register the code-generating procedures for each LIR instruction and also initialize the system call table.

3.3 Code Generation

3.3.1 Time-Flow

After a parser has finished parsing its scope the generated LIR code is passed to the back end's `Optimize` procedure. The purpose of this step is to allow the backend to perform certain optimizations before the actual code generation begins.

Each parser having completed these local optimizations, the (possibly modified) LIR is sequentially passed to the code generator which transforms it into machine code.

When the code generation step has completed, the code is passed to the object file plug-in which then generates the object file.

From a programmer's view this means that the optimizations must be able to run simultaneously, whereas the code generation itself is non-concurrent.

A sample time-flow is illustrated in figure 3-3:

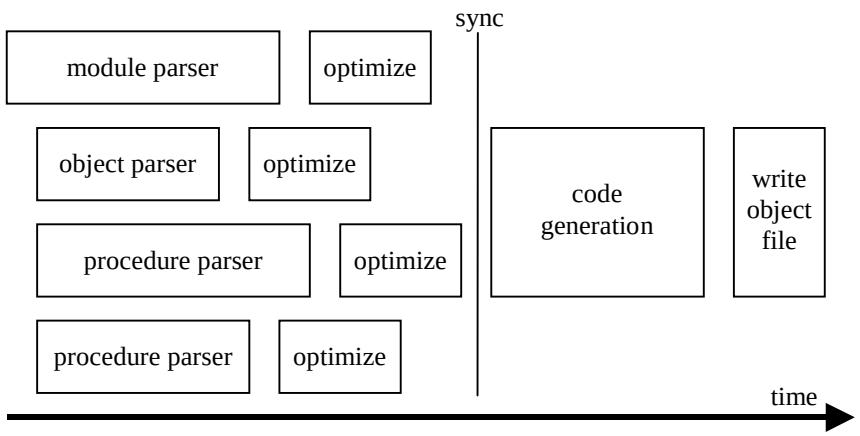

Figure 3-3. Compile time-flow

3.3.2 Optimizing the LIR

The ARM optimizer performs the following steps to make the most of the features of the ARM processor:

3.3.2.1 Inlining of constant values

This optimization step tries to inline immediate values into data-processing instructions.

The second operand of data-processing instructions is the so-called *shifter operand* (“Immediate operand value” on page 9), which can also contain an immediate value (the constraints for possible immediate values are described on page 9).

The general code-pattern looks like this:

```
opA     := register                 opA   := register
opB     := value            =>
res     := op opA, opB              res   := op opA, value
```

Several optimizations improve the number of inlined constant values:

a) if the operator is commutative and the first operand is a constant value and the second operand is not, the operands are swapped.

   Code-pattern:

```
opA     := value
opB     := register          =>   opB   := register
res     := op opA, opB            res   := op opB, value
```

b) if the constant value is negative and the operator is an `add` or a `sub`, the positive constant value is inlined and the operator changed to a `sub` or an `add` respectively.

   Code-pattern:

```
opA     := register                opA   := register
opB     := value (< 0)       =>
res     := add/sub opA, opB        res   := sub/add opA, (-value)
```

c) if the operator is `sub` and the first operand is a constant value, the operator is replaced by rsb (reverse subtract, subtracts the first operand from the second one) and the constant value is inlined.

   Code-pattern:

```
opA     := value
opB     := register          =>   opB   := register
res     := sub opA, opB           res   := rsb opB, value
```

3.3.2.2 Inlining of shifts

A speciality of ARM processors is their ability to perform a data-processing instruction and a shift of the second operand in one instruction. The optimizer recognizes such situations and inlines the shift into the shifter operand. This is specially interesting for array accesses where the address is calculated by adding a base register to an index multiplied by the array element size.

Code-pattern:

```
base    := register                base  := register
idx     := register
size    := value (2^n)       =>
off     := shl idx, size
addr    := add base, off           addr  := add base, idx shl size
```

3.3.2.3 Inlining of address offsets

Similar to the inlining of constant values, offsets relative to a base register can be inlined into load or store instructions. Since the immediate value can be 12-bits wide for loads or stores of words or unsigned bytes, the optimizer also checks for missed inlines in add or sub instructions (8-bit immediate).

Code-pattern (example for load/add):

```
base    := register                 base   := register
opB     := value              =>
addr    := add base, opB
res     := MEM[addr]                res    := MEM[base + value]
```

3.3.2.4 Removal of unnecessary instructions

Inlining constant values obviates a lot of unnecessary instructions as indicated in the examples above.

Removal of these instructions actually happens in parallel to the inlining optimizations. Every instruction that produces a result (such as a load or data-processing instruction) also has a use-count that indicates how many times the result of this instruction will be needed later. Inlining a constant value decreases the use-counter of the instruction that loaded the value. If the counter reaches zero it can be removed.

In the case of unnecessary add or sub instructions (this can happen while inlining address offsets) the situation is a bit more complicated. If the counter of the add or sub instruction reaches zero, the instruction can be removed and the use-counters of the two operands need to be decremented. These updates propagate until no instructions can be removed anymore.

3.3.2.5 Example

To illustrate the optimizations of the LIR, let us look at an example:

```
PROCEDURE Example;
VAR a,b,c: LONGINT; arr: ARRAY 10 OF LONGINT;
BEGIN
  a := 7;
  b := 20H - a;
  c := a DIV b;
  arr[c] := b
END Example;
```

The LIR generated by Paco is shown on the left side of figure 3-4. On the right side is the LIR after it has been optimized by the backend:

LIR generated by Paco

```
PROCEDURE Example;
VAR a,b,c: LONGINT;
    arr: ARRAY 10 OF LONGINT;
BEGIN
 0    enter
a := 7;
 1    label  102
 2 1  loadc  SD2, 7
 3 1  loadc  D3, -4
 4 1  add    D4, FP, D3
 5    store  0[D4], SD2
b := 20H – a;
 6    label  111
 7 1  loadc  SD7, 32
 8 1  load   SD8, -4[FP]
 9 1  sub    SD9, SD7, SD8
10 1  loadc  D10, -8
11 1  add    D11, FP, D10
12    store  0[D11], SD9
c := a DIV b;
13    label  126
14 1  load   SD14, -4[FP]
15 1  load   SD15, -8[FP]
16 1  div    SD16, SD14, SD15
17 1  loadc  D17, -12
18 1  add    D18, FP, D17
19    store  0[D18], SD16
arr[c] := b
20    label  141
21 1  loadc  D21, -52
22 1  add    D22, FP, D21
23 2  load   SD23, -12[FP]
24 1  loadc  SD24, 10
25    cmp    SD23, SD24
26    tae    7
27 1  loadc  B27, 2
28 1  ash    SD28, SD23, B27
29 1  add    D29, D22, SD28
30 1  load   SD30, -8[FP]
31    store  0[D29], SD30
END Example;
32    label  154
33    exit
```

LIR after optimization

```
PROCEDURE Example
VAR a,b,c: LONGINT;
    arr: ARRAY 10 OF LONGINT;
BEGIN
 0    enter
a := 7;
 1    label  102
 2 1  loadc  SD2, 7
~~ 3 0  loadc  D3, 4~~
~~ 4 0  sub    D4, FP, 4~~
 5    store  -4[FP], SD2
b := 20H – a;
 6    label  111
~~ 7 0  loadc  SD7, 32~~
 8 1  load   SD8, -4[FP]
 9 1  rsb    SD9, SD8, 32
~~10 0  loadc  D10, 8~~
~~11 0  sub    D11, FP, 8~~
12    store  -8[FP], SD9
c := a DIV b;
13    label  126
14 1  load   SD14, -4[FP]
15 1  load   SD15, -8[FP]
16 1  div    SD16, SD14, SD15
~~17 0  loadc  D17, 12~~
~~18 0  sub    D18, FP, 12~~
19    store  -12[FP], SD16
arr[c] := b
20    label  141
~~21 0  loadc  D21, 52~~
22 1  sub    D22, FP, 52
23 2  load   SD23, -12[FP]
~~24 0  loadc  SD24, 10~~
25    cmp    SD23, 10
26    tae    7
~~27 0  loadc  B27, 2~~
~~28 0  ash    SD28, -1, SD23, 2~~
29 1  add    D29, D22, SD23, 2
30 1  load   SD30, -8[FP]
31    store  0[D29], SD30
END Example;
32    label  154
33    exit
```

Figure 3-4. Optimization of LIR

Not counting the labels which produce not code, the original LIR contains 29 instructions. The optimizations removed 11 instructions, so that the LIR after the optimization only contains 18 instructions!

3.3.3 Code Generation

For each LIR instruction that is not suppressed (i.e., the use-counter is greater than zero), the appropriate handler in the code-generator plug-in is called. The translation into a machine instructions is normally straightforward.

3.4 Implementation Details

Some interesting implementation details are described in this section.

3.4.1 Register Allocation

The register usage is shown in Table 3-1:

Register	Usage
`R0`	general purpose register, return value
`R1`	general purpose register, 2nd half of large return value
`R2-R11`	general purpose register
`R12`	Frame Pointer (FP)
`R13`	Stack Pointer (SP)
`R14`	Link Register (LR)
`R15`	Program Counter (PC)

Table 3-1. Register Usage

At the entry to a procedure all registers are free. Results that are less than or equal to four bytes are returned in R0. Results that need 8 bytes are returned in R1:R0 (the high four bytes are returned in R1).

The caller is responsible for saving any registers that are in use when calling a procedure. This is a better solution than to have the callee save the whole register set, because the caller knows exactly what register values will be needed after the call and can explicitly save only those. The callee, on the other hand, does not know which register values must be preserved and would have to save all registers.

Registers are managed by the register manager (cf. Table 3-1). In normal operation, the register manager keeps track of which registers are used and which registers are free.
Registers are allocated by the code generator with a *use-count*. After each use, the code generator decreases the use-count the register, but the register will only become free if its use-count reaches zero.

The ARM backend supports a much more elaborate register allocation scheme which is described in “Code Optimization” on page 29.

3.4.2 Calling Convention and Procedure Activation Frame

To understand the structure of the *procedure activation frame* (PAF), let us take a look at what happens exactly when a procedure is called:

1. the *caller* pushes the parameters from left to right.
2. the *caller* pushes the static link (SL) if the callee is a local procedure or the pointer to the object (SELF) if the callee is a method of an object (but not both).
3. the *caller* branches to the prologue of the *callee*.
4. the *callee* saves the frame pointer (FP) and the link register (LR) on the stack
5. the *callee* allocates space for local variables on the stack and fills it with zero.
6. the *callee* makes a copy of each open array passed as a value parameter.
7. the *callee* runs.
8. the *callee* places any result in the result register.
9. the *callee* restores the FP, LR, and SP, thereby removing the parameters from the stack
10. the *callee* branches to the LR.

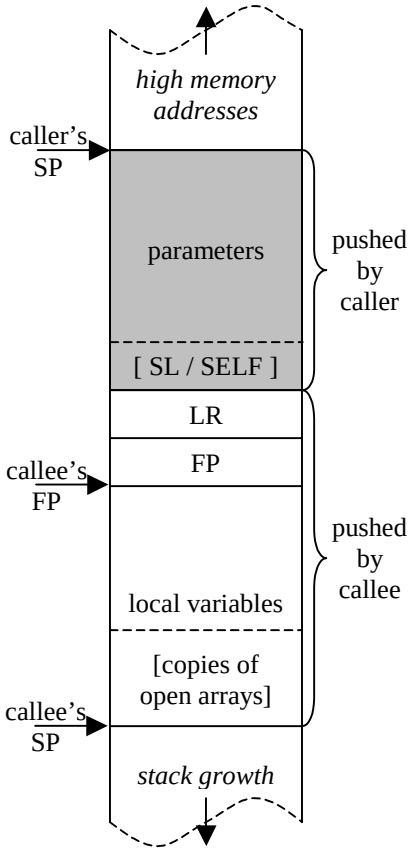

Figure 3-5. Procedure Activation Frame

The following conclusions can be drawn:

- *Parameters* are accessed with a *positive offset relative to the frame pointer.* The offset of the first parameter depends on the type of the procedure. For procedures that are neither local nor a method of an object, the first parameter is located at `MEM[FP+8]`. Local procedures and object methods have the first parameter at `MEM[FP+12]`.
- *Local variables* are accessed with a *negative offset relative to the frame pointer.* The variables are allocated in order of declaration, so if the first declared variable is a `LONGINT`, it is located at `MEM[FP-4]`.
- `MEM[FP+0]` is the caller's frame pointer, `MEM[FP+4]` the return address.

What sounds quite complicated is really easy to implement. Paco's Oberon parser takes care of all the details. The only thing the backend must take care of is that it pushes 8 bytes on the stack in the prologue of a procedure and sets the frame pointer below these 8 bytes to ensure the assumptions made in the intermediate representation are correct.

3.4.2.1 Await Calling Convention

An await call is generated for the `AWAIT` statement. An `AWAIT(cond)` statement blocks the process until the condition `cond` becomes true. The active objects runtime system ensures that the condition is re-evaluated whenever the value of the condition could have changed.

The await call uses a special calling convention: it has only one parameter, the static link (which is the frame pointer of the enclosing procedure). In the prologue of an await call the frame pointer must be set to the static link which enables the AWAIT procedure to access the variables of the enclosing procedure, even if it does not run in the context of the procedure that called AWAIT.

For further details please refer to [Mull00],[Rea00].

3.4.3 Constant and Address Pool

The ability to load constants into a register, as described in "Immediate operand value" on page 9, is limited on ARM processors. Only constants that fit into the shifter operands rotate-immediate scheme can be loaded in one instruction.
An arbitrary 32-bit constant can be loaded into a register with the following instructions (we load 12345678h as an example):

```
MOV     R0,#78H              ; load lowest byte
ADD     R0,R0,#56H,LSL #8    ; add 2nd byte shifted left 8 times
ADD     R0,R0,#34H,LSL #16   ; add 3rd byte shifted left 16 times
ADD     R0,R0,#12H,LSL #24   ; add last byte shifted left 24 times
```

Example 3-1. Loading an arbitrary 32-bit constant directly

Four instructions to load one constant is definitely too much. The common solution on ARM processors is to embed constants into the code and use PC-relative addressing to load them into a register.
The code pattern is:

```
LDR   R0, [PC, offset]      ; loads the word MEM[PC+offset] into R0
...
...
DCD   12345678H
```

Example 3-2. Loading an arbitrary constant using PC-relative addressing

Two important points must be noted about this solution:

1. The control flow must not reach the constant, otherwise the processor would try to execute it as an instruction.
2. When calculating the offset, we must keep in mind that the PC is already two instructions ahead. The formula to calculate the correct offset is thus

   offset := address of constant - (address of instruction + 8)

   and the offset must lie in the range

   -1000H < offset < 1000H

The same situation arises for addresses. Addresses are used to access variables that are globally visible within a module and also for imported variables. These addresses are not known at compile-time, but will need to be fixed by the loader when the module is loaded into memory.

The constant and address pool service module provides an simple interface to the code generator, so that the code generator can focus on what it should do - generate code, of course.

```
TYPE
  ConstantPool = OBJECT
    PROCEDURE & Init;
    PROCEDURE AddConstant (pc, c: LONGINT): LONGINT;
    PROCEDURE AddAddress (pc: LONGINT; adr: PCM.Attribute): LONGINT;
    PROCEDURE Flush (pc: LONGINT);
  END;
```

Figure 3-6. Interface of the constant and address pool

The constant and address pool manages a list of all constants or addresses that the code generator has used in the code. Occasionally, the code generator calls the pool's `Flush` method to give it the possibility to embed the constants and addresses into the code.
These calls always happen at specific locations in the code that cannot be reached by the program counter - after an unconditional branch, a software interrupt or at the end of a procedure. The pool will then embed the accumulated constants and addresses into the code and fix up all references to them.

The pool remembers constants and addresses that are already embedded into the code. If the code generator requests the use of a constant or address that is still within the offset limits, the constant is reused and does not need to be put into the code for a second time. This helps to keep the code size small and reduces the number of fixups needed when the module is loaded into memory, thus decreasing the linking time.

These situations are illustrated in Figure 3-7: Three constants are loaded, once C1 (2) which is already imbedded in the code and twice C2 (1, 3) which is new. Loading C2 therefore adds a use to the constant C2 in the constant and address pool (4), while loading C1 is directly encoded with a PC-relative load (5). When the code generator calls the `Flush` method (6), the constant C2 is embedded into the code (7) and the offsets of the two load instructions are fixed (8).

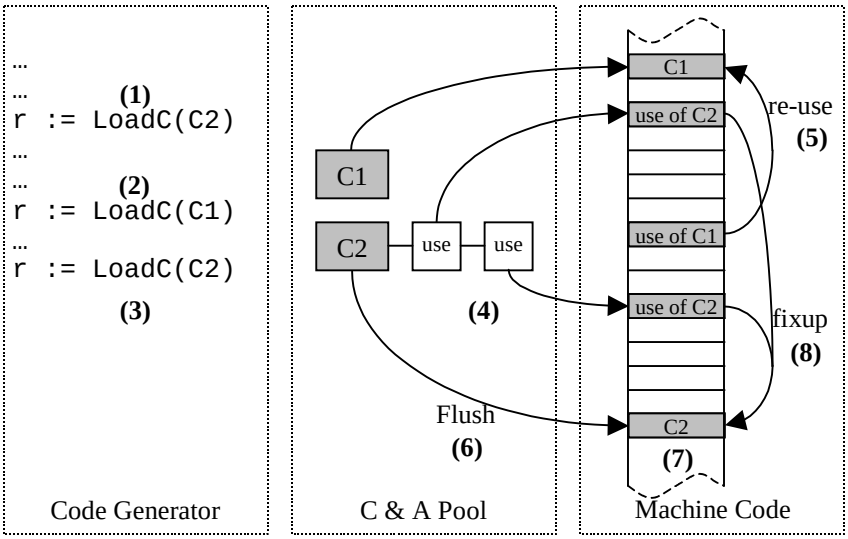

Figure 3-7. Interaction between the code generator, the constant and address pool and the machine code array

In very rare situations it can happen that the control flow of a module is not interrupted for a very long time. In these cases, the constant and address pool might not be able to flush its data before the offsets get too big.
To prevent this, the pool installs a callback handler in the PCOARM module which fills the generated opcodes into the code array. The pool will then set up a watchdog somewhat before the last possible position where it needs to put the constants or addresses into the code.

If the generated code module reaches this watchdog the upcall is invoked. The pool will then embed the constant and addresses into the code, preceded by an unconditional branch immediately behind the data section.
The code generator has the possibility to ‘lock’ or ‘unlock’ this feature in some situations where a certain code sequence may not be intercepted by data blocks (e.g. during an epilogue of a procedure).

The following figure illustrates such a situation:

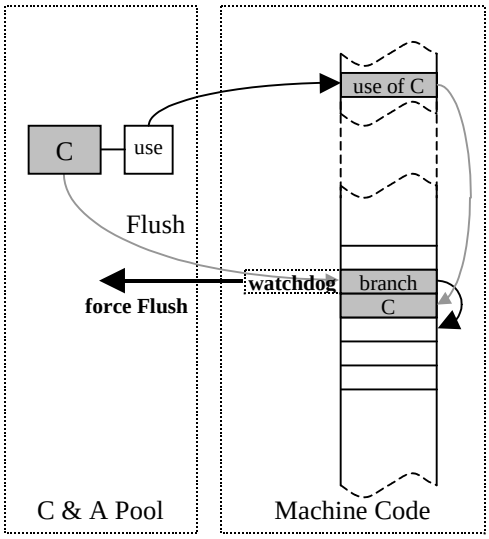

Figure 3-8. Forced flush due to activation of the watchdog

3.4.4 Calls

3.4.4.1 Local calls

Local calls - calls to procedures within the same module - are encoded using the ARM's BL instruction, which automatically saves the return address to the link register.

The maximum branch offset (forward or backward) is approximately 32MB. Since the maximum code size of one module is 256KB (see "Extended Object File Format" on page 30), all local calls can be encoded with the BL instruction.

Local calls can always be resolved at compile-time. If the address of the callee is already known, the backend can encode the branch directly. If the address is not yet known because it is a forward call, a branch with offset 0 is placed in the code and the location added to the callee's fix list. As soon as the backend is invoked to encode the prologue of the callee, it fixes unresolved references.

3.4.4.2 External calls

External calls are calls to procedures of imported modules. They always have to be fixed by the loader, because at compile time we cannot know the address the imported module will be loaded at.

External calls possibly exceed the 32MB limit of the BL instruction, therefore a different code pattern must be used. The ARM backend encodes external calls with two instructions and one data word:

```
ADD     LR, PC, #4[1]       ; let LR point to the return address
LDR     PC, [PC, #-4]       ; load the address into the PC (branch)
DCD     address             ; this address needs to be fixed
...                         ; return here
```

3.4.4.3 System calls

System calls are special calls to certain low-level system procedures. From the back-end's point of view, a system call is just a call to an arbitrary address. Therefore the ARM backend uses exactly the same code pattern to implement system calls as for external calls (see 3.4.4.2).

3.4.5 Divide/Modulo Support

ARM processors lack both the divide and the modulo operation, so it must be implemented in software. Unfortunately, ARM Ltd. does not provide a software div/mod solution.

Aan extensive search on the web lead to a really fast and simple div/mod algorithm [Wil91] with an average execution time of 58.25 cycles in the main loop. Because the definition of div/mod in Oberon is more mathematical than it is in C, some 'glue' is needed to obtain correct results with negative denominators.

Compared to Intel Pentium processors which need 43 cycles to perform a signed division, the ARM software solution is not far from the Pentium.

The ARM backend implements div/mod operations as a system call.

Operation	System call	Result returned in
`a DIV b`	`FastDivMod.DivMod`	`R0`
`a MOD b`	`FastDivMod.DivMod`	`R1`

Table 3-2.

3.4.6 Floating Point Support

Floating point support is provided by a software floating point library by John R. Hauser [Hau98]. The library supports both 32-bit single precision and 64-bit double precision according to the IEEE standard 754-1985 [IEEE754].

The library was originally written in C and has been ported to ARM assembler in an earlier diploma thesis [Sig99].

As in the case of div/mod ("Divide/Modulo Support" on page 28), all operations that involve floating point values are implemented as system calls.

1. Note again that the PC is always two instructions ahead!

3.4.7 Code Optimization

Normally, Oberon compilers do not keep values in registers. Each time a variable is accessed, it is loaded from memory into a register, and each time a variable is assigned a value, the value is written back to memory.

To make the most of the SA-110's registers, the code generator together with the register manager can keep track of values that are currently held in registers.

Every time the code generator needs to load a constant value or a memory location into a register, it generates a *content object* that describes what the register will contain and passes this content object to the register manager. The register manager checks if any of the registers that are free already contains the requested value. If this is so, the register containing the desired content is allocated and the code generator will not generate code to load the constant value or memory location into the register.

Registers are allocated with a *least recently used (LRU) strategy* to make the reuse of registers more efficient.

It is very important that the code generator informs the register manager of *every change* in the register content. Otherwise, the register manager could wrongly return a register, claiming that it already contains the desired content. This would lead to incorrect machine code.

Table 3-3 shows how the register contents are affected by certain actions:

Action	**Register Content**
load	content is replaced
store	content is added to register and removed from all other registers
call	content of free registers is destroyed
branch target	content of all registers is destroyed

Table 3-3.

Currently, the register manager supports the following content types

- constant values
- memory addresses that have FP as the base register

  This content type affects all parameters and local variables of a procedure.

Absolute memory locations are not supported, because it could affect memory mapped I/O.

This simple optimization reduces the average code size by 5% when compiling the whole Aos system. It is even more effective on modules with lots of mathematical calculations.

To illustrate the effects of the optimization, let us look at an example:

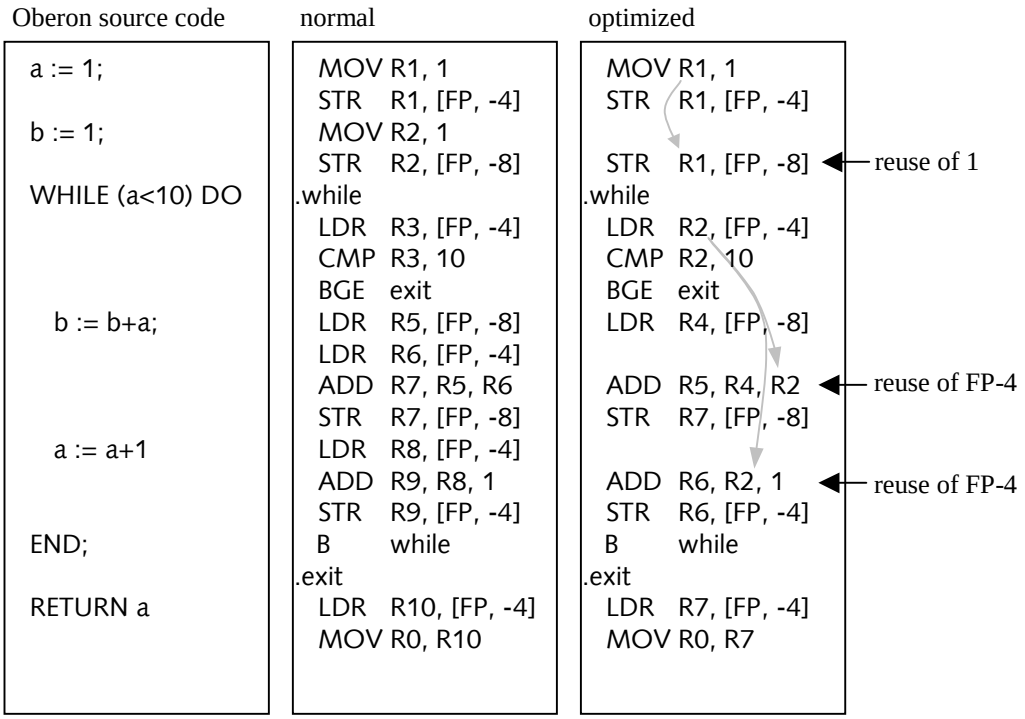

Figure 3-9. Code Optimization Example

The number of instructions has been reduced by three, two of them inside the WHILE-loop. The code inside the loop has not only been reduced by almost 20%, but it will also execute at least 20% faster! (The effective performance gain will be even bigger because the omitted instructions are `load` instructions that would cause a pipeline-stall in the SA-110).

The trap handling mechanism of Oberon is not compromised because `store` instructions are never omitted, thus the value on the stack always contains the true value of the variable.

The optimizations can be turned on with the \o switch:

```
PC.Compile \dARM \o *
```

3.4.8 Extended Object File Format

The original object file format limits the size of the code to 64KB. Modules compiled for ARM processors generally have a bigger code size than their Intel versions because:

a) ARM processors are RISC architectures that sometimes need more than one instruction to accomplish the same task as one instruction on a CISC architecture such as Intel's.

b) an ARM instruction is always four bytes long whereas Intel instructions can be as small as one byte.

Modules that were already close to the 64KB limit when compiled for Intel Aos did not compile any more for ARM Aos. Therefore an extended object file format had to be developed.

Every location in the code that needs to be fixed by the loader is stored in the object file as an unsigned 16-bit integer. Since all offsets in an ARM object file are multiples of four, we simply shift every offset right by two. This simple idea extends the maximum code size to 256KB, which is certainly enough.

The object file format itself does not change at all. It is the task of the module loader to correctly interpret the shifted offsets.

CHAPTER 4

ARM Assembler

Do we need an assembler at all? We can do almost everything in Oberon, but to gain access to the full possibilities of the processor, it is essential to have the possibility to write assembler code. Examples of modules that could not be programmed without assembler support are usually device drivers and very low-level modules that handle special operating system-specific tasks, such as the memory management or the interrupt handling.

The assembler developed during this work is a table driven, concurrent assembler that supports every instruction the SA-110 can execute.

4.1 Structure

The assembler plug-in for Paco consists of one single module. Within this module, the following objects are defined:

- *Assembler*

  This is the main assembler object. Several instances can run concurrently. It parses the assembly language scope and generates the machine code plus a list of fixups that will need to be handled by the ARM code generator.

- *Scanner*

  The scanner object is responsible for scanning the assembly language scope. A specialized scanner is needed, because the ARM assembly language contains some syntax elements that cannot be scanned by Paco's default scanner.

- *SymbolTable*

  Since the extended assembly language allows definition of labels and register synonyms, the assembler needs some sort of a symbol table. This symbol table contains functions to add and retrieve labels and definitions.

- *Mnemonics*

  This singleton object contains the list of mnemonics and directives as well as functions to add and retrieve mnemonics.

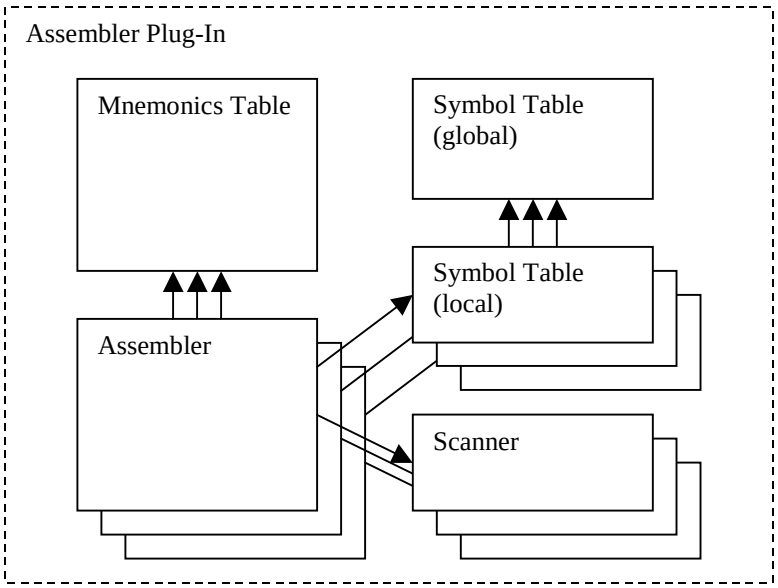

Figure 4-1. Assembler Plug-In Structure

4.2 Mnemonics

The assembler supports the whole assembly language as described in [ARM96]. A list of the mnemonics can be found in appendix B.

Unfortunately, parsing the instructions is not trivial because the assembly language was not carefully designed:

```
LDR         R0, [FP, #-8]
```

is a simple load instruction that loads the word at `FP-8` into register zero, and

```
LDRSB       R0, [FP, #-8]
```

is a load of a signed byte at `FP-8` into register zero.

Now we want to execute those instructions conditionally if the carry flag is set. We get

```
LDRCS       R0, [FP, #-8]
LDRCSSB     R0, [FP, #-8]
```

The condition code mnemonic was actually placed in between the LDRSB mnemonic! To make the situation worse, the LDR instruction has a completely different opcode compared to that of the LDRSB instruction. The opcode is not known until the end of the mnemonic has been parsed, so the assembler needs to know if the first part of the mnemonic was a LDR or a STR. For a table-driven assembler this is unfortunate, because it needs to preserve some kind of state information between procedure calls.

4.3 Labels and Directives

The assembler supports labels and the three directives:

```
DCB, DCW, DCD
```

All of these directives are used to put raw data into the code. All three support comma-separated numbers. DCB also accepts 0X-terminated strings and characters:

```
DCB   "this is a string"
DCB   'a', 'b', 'c', 0FFX
```

DCD also accepts global variable names to allow access to those

```
DCD   globalVar
```

More details are described in the following section.

Labels can be used to simplify branches; they can be used as a branch target after B[L] and must be defined somewhere in the same scope. Here's an example:

```
B       label
...
...
.label
```

4.4 Extended Functionality

To simplify writing assembly procedures, extra functionality has been added to the assembler.

4.4.1 DEFINE directive

It is always tedious to remember what register a value is stored in, and quite often an error comes from a wrong assumption. To simplify this task to the programmer, the `DEFINE` directive can be used:

```
DEFINE buf = R0 : DEFINE len = R1 : DEFINE value = R2
```

These aliases can now be used just like registers

```
SUB   len, len, #1
STR   value, [buf]
```

4.4.2 Accessing local variables and parameters

Local variable and parameter identifiers can be used as addresses in load and store instructions to spare the programmer from calculating the correct offset relative to the frame pointer. This also has the advantage that the procedure activation frame could be changed without invalidating assembly procedures that contain ‘hard-coded’ offsets.

Here’s an example:

```
PROCEDURE Copy(VAR buffer: ARRAY OF CHAR; length: LONGINT;...
VAR a: LONGINT;
CODE
      LDR   R0, buffer
      LDR   R1, length
      STR   R1, a
...
```

The generated code looks like this:

```
LDR   R0, [FP, #CH]
LDR   R1, [FP, #8H]
STR   R1, [FP, #-4H]
...
```

4.4.3 Accessing global variables

Global variables can only be accessed via their address. Since the address is not known at compile-time, the loader will need to patch the address when the module is loaded into memory.

The address of a global variable can be embedded into the code like this:

```
DCD   globalVar
```

To load that address into a register use:

```
LDR   R0, globalVar
```

The assembler calculates the correct offset to generate a PC-relative load:

```
LDR   R0, [PC, offset]
```

It's the programmer's responsibility to ensure that the control flow cannot reach embedded data.

4.4.4 ADR directive - accessing embedded data blocks

The DCx directives allow embedding arbitrary data into the code (see "Labels and Directives" on page 35).

To get the address of such data blocks, a new directive has been introduced: ADR, which takes a register and a label as arguments. The address of the data block will then be loaded into the specified register:

```
ADR   R0, msg
...

.msg  DCB   "Example"
```

The assembler translates this directive into a PC-relative load, as in the case of global variables (see previous section):

```
LDR   R0, [PC, offset]
...

.msg DCB    "Example"
```

4.4.5 Calls

Procedures within the same module can be called using the BL instruction:

```
BL    PrintString
```

The Oberon calling conventions must be met, i.e., parameters are pushed from left to right.

It is also possible to bypass Oberon's calling convention and pass the parameters in registers by defining procedures 'without' parameters. The prologue code of a procedure does not destroy the register values, so the caller can just put the parameter into the registers.

Procedures of imported modules cannot be called directly. To call procedures of imported modules, the address of the procedure can be assigned to a procedure variable and then the procedure can be called by moving the value of the variable into the PC:

```
MODULE Example;

IMPORT AosOut;

VAR aosoutLn: PROCEDURE;

PROCEDURE ExternalCall*;
CODE
  LDR   R0, aosoutLn      ; load address of 'aosoutLn' into R0
  ADD   LR, PC, #4        ; set up link register to point after the DCD data
  LDR   PC, [R0]          ; load value of 'aosoutLn' into PC -> branch
  DCD   aosoutLn
END ExternalCall;

BEGIN
  aosoutLn := AosOut.Ln
END Example.
```

4.4.6 Procedures without a Procedure Activation Frame

Normally, the compiler automatically generates a procedure activation frame (PAF) for every procedure, also for assembler procedures.

By using the {NOPAF} directive, the PAF can be omitted

```
PROCEDURE DisableInterrupts;
CODE {NOPAF}
...
```

If the PAF is omitted, it is the programmer's responsibility to:

- save and restore the caller's FP and the link return register, if necessary
- initialize the local variables with zero and
- remove the parameters from the stack

Accesses to local variables and parameters are still possible; the assembler addresses both relative to the stack pointer.

4.4.6.1 Inlined assembly procedures

Preceding the identifier of an inline assembly procedure with a "-" instructs the compiler to inline the assembly code, i.e., each time the inlined procedure is "called", the compiler does not generate a call to that procedure but instead inlines its code.

Inlined procedures automatically have no procedure activation frame.

CHAPTER 5

DNARD Port

This chapter describes the port of Intel Aos to ARM Aos. This port is indended to be used as a reference for ports to mobile devices with StrongARM processors. Quite often, not the simplest, but the most portable design was chosen to minimize the efforts needed to port this implementation to a Compaq iPAQ handheld, for example.

5.1 DNARD Network Computer Overview

The DIGITAL Network Appliance Reference Design (DNARD) network computer was developed in 1997 by DEC as a reference design for network computers [DEC97].

Table 5-1 lists the main components of the DNARD network computer:

Processor	SA-110, 233MHz
Memory	32MB SDRAM, 66MHz memory bus
Firmware	OpenFirmware (IEEE Standard 1275) from FirmWorks
Network	Crystal Semiconductor CS8900 10BASE-T ethernet controller
Video	IGS CyberPro 2010 graphics adaptor (2MB memory)
Input	PS-2 keyboard and mouse ports

Table 5-1. DNARD Technical Specification Summary

The DNARD has a PC-style architecture. Connected to an ISA bus are several PC-compatible peripherals such as the keyboard and mouse controllers.

The DNARD firmware has many useful features such as discovering the devices that are attached to the bus, allocating and deallocating memory or printing out a disassembly of an arbitrary memory block.

The DNARD boots by loading a boot image from a specified server with the TFTP protocol [Sol92] into memory and then starts executing the boot image. It also offers other possibilities to load a boot image, but these are not relevant for this port. For more information about the DNARD network computer refer to [DEC97].

5.2 Booting ARM Aos

The ARM Aos kernel is located at the virtual address 100000H (1MB) in order to remain as compatible as possible to Aos on Intel architectures.

Unfortunately the boot image is loaded at the virtual memory address F0000000H by the firmware, and starts executing there. Theoretically, it would be possible to configure the load address to another value, but this does not work in practice since OpenFirmware's boot loader itself crashes if the load address is different from the address mentioned above.

For this reason, the boot process got somewhat tricky. The boot image actually consists of two separate boot images, the first one (the relocator, see below) merely copies the second boot image to the virtual address 1MB and continues execution there, while the second boot image is the real ARM Aos boot image.

5.2.1 The Relocator

The relocator's job is to create an environment similar to those on mobile devices, i.e., without the luxury of OpenFirmware, to load the Aos boot image to the virtual address 100000H, and to pass control to this image.

To accomplish this task, the relocator disables the memory management unit (MMU), sets up the new page tables and turns the MMU back on. One problem when disabling the MMU is that virtual addresses are suddenly not valid any more, therefore the code that disables the MMU *must* be located at identical virtual and physical addresses.

At this point, another problem of the DNARD comes into play: the physical memory does not start at the address 00000000H and is not contiguous. Figure 5-1 shows the physical memory locations in the DNARD:

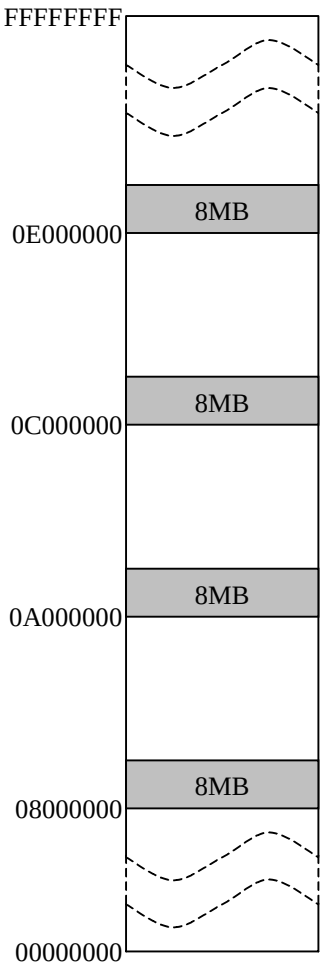

Figure 5-1. Physical memory layout on the DNARD

The attentive reader will notice that the DNARD does not have any memory at the physical address 00100000H.

Therefore, the relocator has to performs the following steps:

1. copy itself to an address that has identical physical and virtual addresses and jump into the copied code. This memory is mapped by OpenFirmware.
2. copy the second boot image to the physical start of the memory.
3. switch the MMU off.
4. set up the page tables. The second boot image will be mapped in at the virtual address 00100000H.
5. set up the stacks for all processor modes.

6. set up the interrupt vector table.
7. copy some system information that was gathered while the OpenFirmware was still running into the second boot image.
8. turn the MMU back on.
9. pass the control to the Aos boot image.

This rather complicated sequence is illustrated in the following figure:

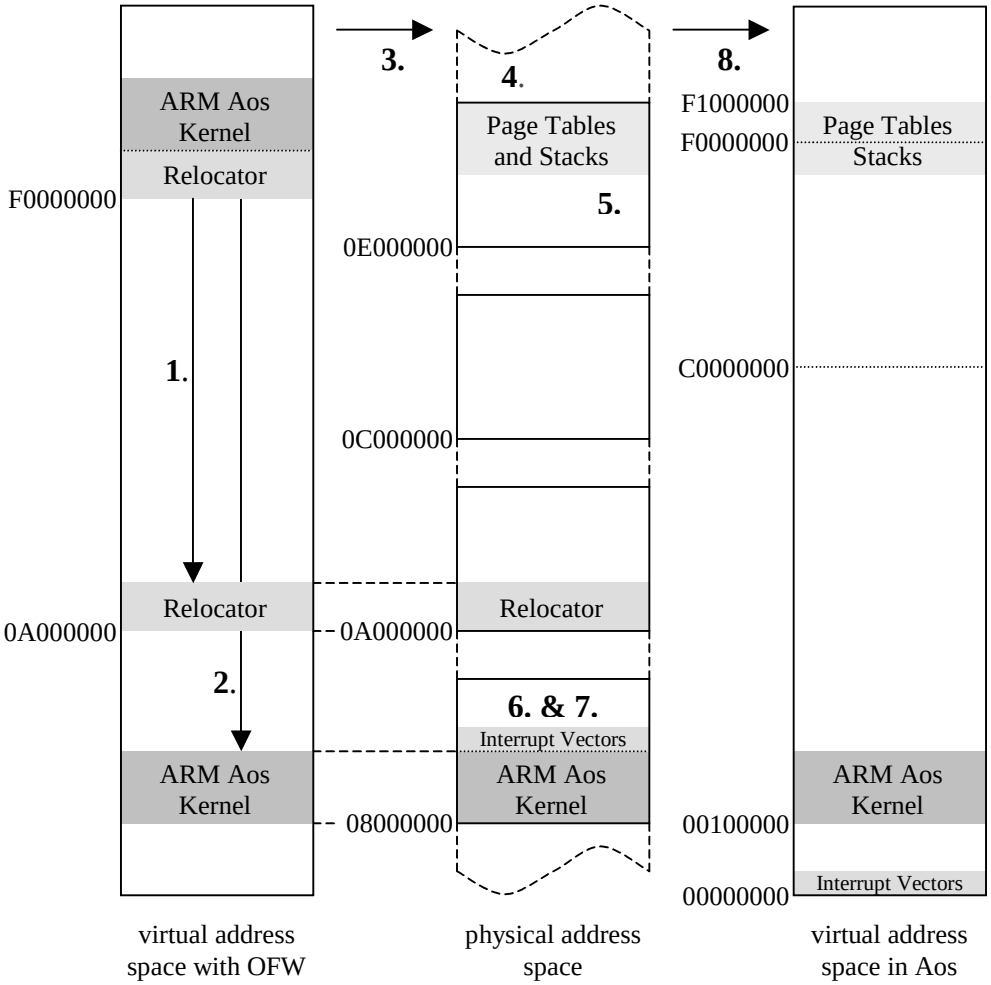

Figure 5-2. Booting ARM Aos

5.3 ARM Aos Kernel

The ultimate ambition during the port of the kernel was *total compatibility* to minimize the changes to higher level modules. This implies that the interfaces remain the same wherever possible.

To achieve this goal, the module structure of the ARM Aos kernel is exactly the same as that on Intel Aos, except for two additional modules to provide the missing arithmetic operations.

The ARM Aos kernel consists of the following modules:

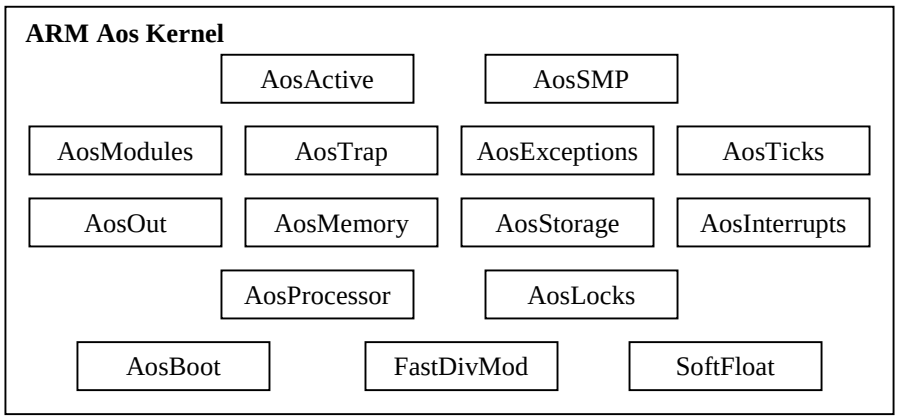

Figure 5-3. ARM Aos Kernel Structure

- *AosBoot* provides bootstrapping and other low-level functions such as enabling or disabling interrupts.
- *FastDivMod* and *SoftFloat* provide arithmetic operations not supported by the processor (see “Divide/Modulo Support” on page 28, and “Floating Point Support” on page 28).
- *AosProcessor* defines atomic operations.
- *AosLocks* provides fine-grained locks to protect kernel data structures.
- *AosOut* provides low-level tracing support to the kernel log and the serial port.
- *AosMemory* and *AosStorage* are responsible for the memory management. AosMemory provides low-level memory management support, whereas AosStorage contains the high-level functions such as the garbage collector or functions to allocate memory blocks.
- *AosInterrupts* contains interrupt controller functions and low-level glue code to convert interrupts into standard Oberon procedure calls.
- *AosModules* controls the modules that are loaded into memory and the type system.
- *AosExceptions* and *AosTrap* provide the Oberon trap handling support.
- *AosTicks* provides a periodic timer.
- *AosActive* and *AosSMP* provide the run-time support for scheduling, automatic condition-evaluation and active objects.

  AosSMP could actually be left out, but is implemented to maintain a maximum interface compatibility between Intel and ARM Aos.

ARM Aos normally runs in *System or IRQ mode* (see “Registers and Processor Modes,” on page 3). The advantages of running in a privileged processor mode are that the kernel can enable or disable interrupts and access the MMU coprocessor very quickly.

The only, but serious disadvantage of this implementation is that the memory area at 00000000H cannot be protected from read accesses because that’s where the SA-110 expects the interrupt vectors to be located at. This means that read accesses from dereferenced NIL pointers cannot be detected.

Modern ARM processors, such as the SA-1100, support interrupt vectors at FFFF0000H.

5.3.1 Memory Management

The SA-110 has a built-in memory management unit (MMU) that supports virtual-to-physical address mapping through one first-level and many second-level page tables. If the MMU is enabled, every memory address is treated as a virtual memory address and the virtual-to-physical address translation logic is invoked to translate the address into a physical memory address.

The SA-110 supports memory pages of 1MB, 64KB, 4KB and 1KB. 1MB pages are called *sections* and are registered in the first-level page table. The smaller pages are called *large*, *small* and *tiny* pages, respectively. All of these page types are registered in second-level page tables.

On systems with limited memory resources it is important not to waste too much memory for page tables. If the whole 4GB memory space would be split into small pages (as it is done on Intel Aos), 4MB of RAM would be needed for the second level page tables (the size of each second-level page table is 1KB).

ARM Aos maps the heap area with sections, therefore no second level page tables are needed for the whole heap address space. This implementation decision saves 3MB of RAM since we only need 1MB for second-level page tables.

5.3.1.1 Virtual Memory Layout

The virtual memory layout is shown in figure 5-4. The interrupt vector table must be located at 00000000H, because the SA-110 does not support the alternate location at FFFF0000H.

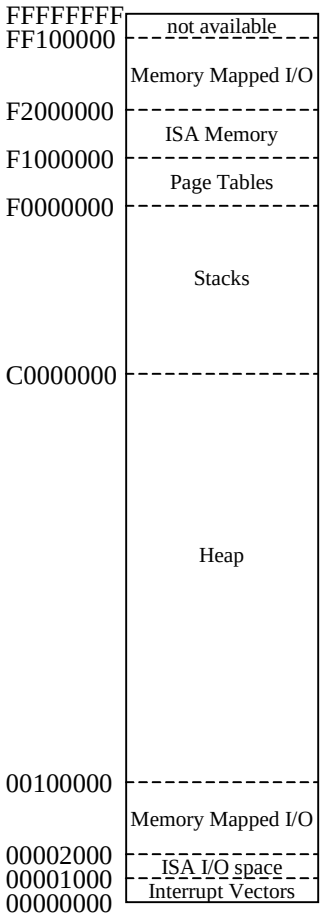

Figure 5-4. Virtual memory layout on ARM Aos (not to scale)

This layout is almost identical to that on Intel Aos. This has the advantage that the garbage collector does not need to be changed.

5.3.1.2 Heap Organization

The heap starts at the virtual address 00100000H and ends at BFFFFFFFH, although - unless the computer has lots of memory - the actually mapped area will be much smaller.

Modules, objects and dynamically allocated types are located on the heap. Every block that is allocated on the heap has a clearly defined structure that allows the garbage collector to remove memory blocks that are not accessible by any module or object any more.

5.3.1.3 Stack Organization

The stack area starts at C0000000H and ends at EFFFFFFFH. Since all processes share the same address space, the stack of each process cannot be located at the same virtual address. For this reason, the stack area is split into smaller blocks of 128KB. Each block is a - initially unallocated - stack that can be assigned to a process when it is created.

Of course, not the whole 128KB will be physically allocated, but a process may not have more than 128KB of stack even if there would still be free memory.
The initially allocated physical memory is 4KB.

By dividing the stack area into stacks of 128KB each we get a total of 6144 stacks. One stack area is reserved for the kernel and the remaining 6143 stacks can be assigned to processes. It is therefore not possible to have more than 6143 active processes at one time.

For systems that need more processes, the maximum stack size can be reduced.

5.3.1.4 Low-level memory management

The AosMemory module is responsible for the low-level memory management such as expanding the heap area, creating, deleting and expanding stacks or mapping physical memory ranges to a virtual address.

Heap management

The heap can only be expanded. If the AosStorage module does not have enough memory to fulfill a NEW() request, it calls AosMemory.SetHeapEndAdr to get more physical heap memory.

As mentioned before, the heap is allocated in sections (1MB blocks). Sections need to be aligned on a 1MB boundary, small pages and sections can not be mixed in the physical memory.

AosMemory allocates heap blocks at the lowest possible physical address.

Stack management

Each time a new process is created is gets assigned to a new stack. AosMemory maintains a bitmap of the stacks area to quickly find an empty stack.

Initially, only 4KB of physical memory are allocated on the new stack. If the process tries to access an invalid address in its stack area, a page-fault exception is raised by the MMU and AosMemory maps more physical memory to expand the stack.
When a process terminates, the stack becomes free again and the physical memory pages are unmapped.

The stack pages are located at the top of the physical memory. AosMemory maintains a linked list of free memory pages that can be allocated for stacks. The list is stored in the page itself and does not need any memory at all except the pointer to the first memory page.

This allocation strategy is illustrated in figure 5-5:

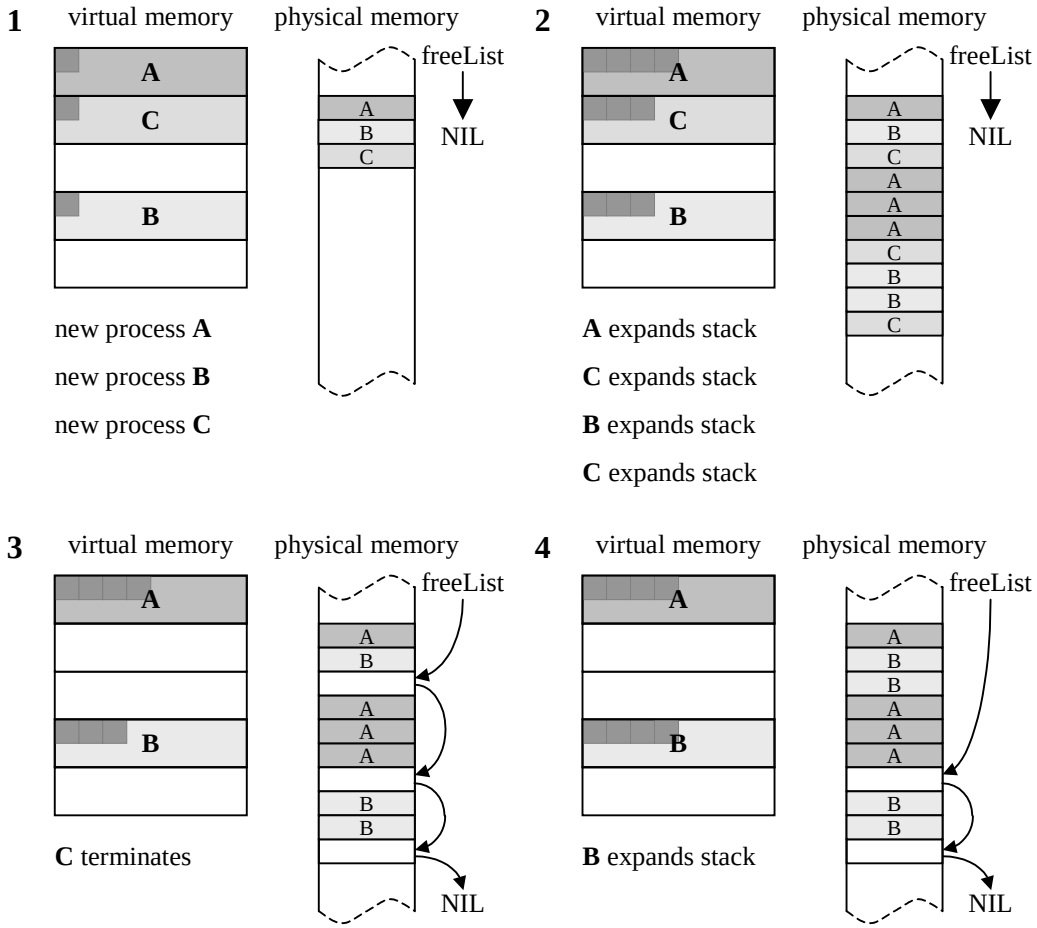

Figure 5-5. Virtual and physical stack allocation strategy

1. three new processes A, B and C are created. Each process is assigned to a stack area and initially one page is physically available.
2. the stacks of the processes grow and AosMemory maps more pages (in chronological order): A grows the stack by 3 pages, C by one page, B by two pages and again process C by one page.
3. process C terminates. AosMemory links the free physical pages.
4. the stack of process B grows again. Now AosMemory assigns first pages that are linked in the freeList.

Physical memory-mapping

The areas 00002000H-000FFFFFH and F1000000H-FF100000H are available for mappings to specific physical addresses. Mapping physical address ranges into the virtual address space is often needed by device drivers to perform memory mapped I/O.

Physical address ranges can be mapped into the virtual address space, but once they are mapped, it is not possible to unmap them again. However, this is not a problem, since mapping the same physical address twice yields the same virtual address. In fact, even overlapping physical regions are mapped to overlapping virtual regions to avoid doubly mapped memory ranges. Doubly mapped memory ranges could lead to apparently inconsistent memory due to write-caching.

Figure 5-6 shows such a situation where two independent physical address mappings are mapped to an overlapping region:

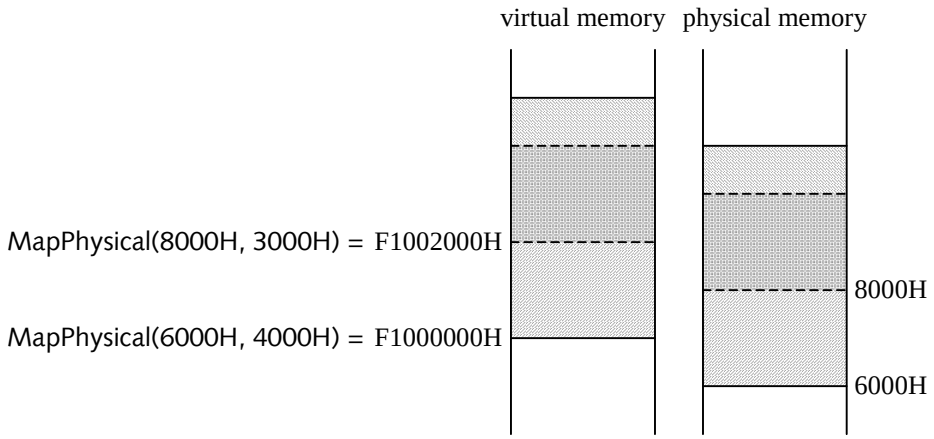

Figure 5-6. Mapping of physical addresses

5.3.2 Interrupt Handling

The DNARD network computer has an Intel 8259 programmable interrupt controller (PIC) that handles hardware interrupts and signals the CPU if any interrupts are pending. If interrupts are enabled in the CPSR[1], the CPU checks for interrupts at instruction boundaries.

If an interrupt is pending, the CPU switches to IRQ mode, saves the CPSR into the SPSR, reads the address of the interrupt handler from the interrupt handler table and branches to the interrupt handler routine.

1. CPSR = current program status register, see “Registers and Processor Modes” on page 3

5.3.2.1 Aos Interrupt Handlers

A low-level interrupt handler in Aos must have the following signature

```
Handler* = PROCEDURE(VAR state: State);
```

and the type “State” is defined as follows

```
TYPE
   (** processor state *)
   State* = RECORD
     R*: ARRAY 12 OF LONGINT;(** registers 0-11 *)
     FP*, SP*, LR*, PC*: LONGINT;(** special registers *)
     PSR*: LONGINT;(** processor state register *)
     INT*: LONGINT(** IRQ number *)
   END;
```

The glue code in AosInterrupts will create this processor state record - which reflects the processor state immediately before the handler was called - and call the appropriate low-level interrupt handler.

The low-level handler can not only inspect the values of the record, it can also *change* them. The glue code will restore the processor state according to the values in the ‘state’ parameter.

This provides a very easy and convenient way to program a preemptive task scheduler, for example. Just set the LR to where you want to return to and the CPU will continue to execute there after the interrupt handler has terminated.

However, this design has a small limitation: if more than one interrupt handler is installed for the same hardware interrupt, they are called sequentially. It is theoretically possible that an interrupt handler will overwrite the changes made by a handler that has been called just before.

The same problem arises if more than one hardware interrupt is pending and the glue code handles all pending interrupts. To avoid this problem, the low-level interrupt handler just handles the IRQ with the highest priority and then returns.

5.3.3 Task Management

The task management is implemented in AosActive, which is probably the most complicated module to understand.

AosActive has a *ready-queue* that contains all processes that are ready to run. Processes that are waiting for a condition in a specific object are linked in the object itself.

There are five situations that lead to a task switch

- the process must wait for an object lock or awaits a condition

- the process is preempted by an interrupt
- the process is preempted by a task switch interrupt
- the process relinquishes control of the processor voluntarily
- the process terminates

The last two situations are easy to understand and implement.

5.3.3.1 Waiting for an object lock or awaiting a condition

In Active Oberon the methods of an object can be marked exclusive, i.e., at any time there is at most one process running inside an exclusive method of an object. This invariant is enforced and controlled by the Aos run-time system.

If a process wants to enter a method marked exclusive and there is already another process running inside an exclusive method of the same object instance, the first process has to wait. As soon as the process that is running inside the exclusive method exits, the waiting process becomes ready to run.

AosActive implements this mechanism with *object locks*. Every instance of an object can be locked exactly once. Each time a process enters an exclusive method of an object, AosActive implicitly locks that object. If the object is already locked, the process is inserted into the object's waiting list. At the exit of an exclusive method, AosActive checks if there are any processes waiting for the lock and if so puts them into the ready-to-run list.

A similar situation arises if a process calls the AWAIT() statement. AWAIT awaits a condition to become valid and can only be called inside exclusive methods. If the condition is not true, AosActive releases the object lock and puts the AWAITing process into the object's waiting-for-condition list and schedules another process. Each time a process exits an exclusive method, the conditions of the waiting processes are automatically re-evaluated and the process put into the ready list if the condition is satisfied.

The object locking and condition handling mechanism is portable and did not need to be changed in ARM Aos.

5.3.3.2 Time slicing

In ARM Aos the real-time clock (RTC) is used to generate a periodic interrupt with a frequency of 1024 Hertz. This interrupt invokes the task scheduler which will schedule another process to run after the interrupt handler has terminated.
The task scheduler is implemented in AosActive.Timeslice.

5.3.3.3 Hardware interrupts

Interrupt handlers for hardware interrupts are wrapped into a *interrupt process* (high-level interrupt handler). The interrupt handler does not handle the interrupt but inserts its interrupt process into the ready-queue. The next time the scheduler is invoked, it will select pending interrupt processes to be run next.

5.3.3.4 Implementation details

Each task switch is protected by a fine-grained lock, the *AosLock.AosActive* lock. The lock must be taken before any of the data-structures in AosActive are modified and must be released immediately after the CPU has switched to the new process.

The following invariants must hold to ensure correct task switching:

- Any process that relinquishes control of the processor because it has to wait for an object lock or a condition to become true must hold the AosActive lock.
- Any process that gets scheduled because it got an object lock or runs because the condition it was waiting for is true must immediately release the AosActive lock.
- Processes that are preempted by an interrupt don’t hold the AosActive lock. Therefore, AosActive.Timeslice must take this lock, since it may schedule a process that will release the lock because it assumes that the lock is taken. On the other hand, if the scheduler schedules a preempted process, it must release the AosActive lock because the preempted process won’t release it.

Below the procedure AosActive.Timeslice is shown. The first action is to get the AosActive lock, which must have been free, because interrupts are off while a lock is active. If there is another process ready to run, the current process is preempted and put into the ready-queue.

Now the new process is scheduled. Note that the AosActive lock is not released if the process has not been preempted; the Timeslice procedure exits at (1).
If the process has been preempted, the state of the preempted process is copied to the ‘state’ record (see “Interrupt Handling” on page 48) and immediately before exiting the AosActive lock is released.

The interrupt handler will restore the CPU state according to the ‘state’ record, which now contains the state of the new, preempted process. As soon as the interrupt handler terminates, the preempted process continues to run.

```
(** Preempt the current process. *)

PROCEDURE Timeslice*(VAR state: AosInterrupts.State);
VAR id: LONGINT; new: Process;
BEGIN (* {interrupts OFF, but were on when the timeslice interrupt happened} *)
    (* handle a timer tick *)
  AosLocks.Acquire(AosLocks.AosActive); (* interrupts were on *)
  id := AosProcessor.ID();
    (* pre-empt the current process *)
  IF AosLocks.PreemptCount(id) = 1 THEN
    IF state.PSR MOD 20H = AosProcessor.System THEN(* not kernel mode *)
      IF running[id] # idle[id] THEN
        Select(new);
        IF new # NIL THEN
          INCL(running[id].flags, Preempted);
          CopyState(state, running[id].state);
          Enter(running[id]);
          running[id] := new;
          new.mode := Running; new.procID := id;

          IF Preempted IN new.flags THEN
            EXCL(new.flags, Preempted);
            CopyState(new.state, state)
          ELSE
            SwitchToState(new, state);
            AosLocks.SetReleaseInterruptState(TRUE);
            (* do not release AosActive lock - will be done by SwitchTo caller *)
(1)         RETURN
          END
        END
      END
    END
  END;
  AosLocks.Release(AosLocks.AosActive)
END Timeslice;
```

Figure 5-7. AosActive.Timeslice

5.3.4 Division, Modulo and Floating Point Support

Support for DIV/MOD and floating point operations is provided by *system-calls* (see “System calls” on page 28). The module loader fixes the addresses of the system calls according to the fix-list in the object file.

A list of the system calls in ARM Aos can be found in appendix C, on page 75.

5.3.5 Boot Parameters

On Intel systems, the configuration of Aos is stored at the start of the Aos partition on the hard drive.

Of course, this approach does not work on the DNARD network computer. However, the OpenFirmware BIOS offers the possibility to store user-defined environment variables in a non-volatile RAM chip.

These environment variables are queried *before* the relocator disables the OpenFirmware and are passed to AosBoot, which offers the same interface to the higher level modules to query those settings.

5.3.6 Compatibility

The goal of total compatibility of the interfaces has been achieved. During the port of the higher level modules there was never a module that could not be compiled because of a missing procedure in the kernel.

However, a few incompatibilities remain due to differences between RISC and CISC architectures. They are described in “Incompatibilities to Intel Aos” on page 56.

5.4 Drivers

5.4.1 Keyboard and Mouse

Thanks to the PC-style architecture of the DNARD network computer, the device drivers for keyboard and mouse were easily ported. In AosKeyboard.Mod, the key-mapping table which is defined in assembly language and the base I/O port had to be adapted. In AosMousePS2.Mod only the base I/O port had to be changed.

5.4.2 Crystal 10Mbps Ethernet

The Crystal Semiconductor CS8900 10BASE-T ethernet controller driver is a new implementation. The initialization code for the controller was taken from [Sig99].

Unfortunately, under heavy load, the controller suddenly stops sending or receiving packets for a few seconds before returning to normal operation. The problem gets worse if the controller is initialized to run in full-duplex mode, thus it currently runs in half-duplex mode. It is not yet clear if this is a hardware- or software-related problem.

5.4.3 IGS CyberPro 2010 Graphics Driver

The IGS CyberPro 2010 graphics driver was implemented from scratch, because the driver in [Sig99] only supported a resolution of 1024x768 pixels with 256 colors at 70Hz.

Aos treats all color values as true-color (32-bit) values, and the conversion to 256 colors is a very time consuming operation (find the best matching value in a table with 256 colors) that has to be done for each pixel. Therefore a better driver was developed.

The basis for the current driver was a sample driver from IGS written in C. Although this sample driver initializes many registers step-by-step, the driver didn't work at first. It was only after some "black-magic" initialization code was copied from the existing driver that a picture was finally displayed.

The current driver supports up to 1024x768 pixels with 65536 colors at 76Hz. There is still some computation needed to convert true-color values to 16-bit colors, but it can be done much faster than the table lookup mentioned above.

Unlike display drivers on Intel Aos, this driver supports dynamic changes of the resolution.

5.5 The Bimbo File System

Since the DNARD network computer does not have a local disk, some kind of remote storage system had to be implemented.

A simple, but effective remote file-system called BimboFS was developed. During the initialization of the kernel, a RAM disk is created with an integrated TFTP client [Sol92]. If the system tries to open a non-existing file on the RAM disk, the TFTP client is invoked to get the file from a remote TFTP server. If the file exists, it is downloaded and stored locally on the RAM disk.

The necessary parameters to mount a Bimbo file system are as follows:

```
<number of blocks> <ip-address of TFTP server>.
```

The parameter <number of blocks> indicates how many 4KB blocks should be allocated for the RAM disk and <ip-address of TFTP server> is an IP address in ASCII format, e.g. 129.132.134.53 which will be used by the TFTP client as the server's address.

5.6 Porting the whole Aos system

Once the kernel and the device drivers have been ported, only a few incompatible modules had to be adapted to ARM Aos.

Refer to appendix D for a complete list of ported modules and the necessary changes.

The two main reasons for incompatibility are

- inlined assembly code and
- unaligned memory accesses.

5.6.1 Inlined assembly code

A few high-level modules contain inlined assembly code for speed critical parts of a module. As there is no way to make the Intel assembly code compatible, the only solutions are

a) translate the Intel assembly to Oberon or

b) translate the Intel assembly to ARM assembly code

In principle solution a) is to be preferred since it will avoid the same problem when porting to another platform. However, if the speed is really critical, it will be necessary to translate the code to ARM assembly code.

TODO

5.6.2 Unaligned memory accesses

Most of the incompatible modules belong to this category. The module compiles and can be run, but it traps as soon as it tries to make an unaligned memory access.

The built-in MMU cannot access the memory with unaligned addresses, i.e., if a word (4 bytes) needs to be read from or written to memory, the memory address must be aligned on a 4-byte boundary (address MOD 4 = 0). The same applies for halfword accesses which must be 2-byte aligned.

On Intel architectures unaligned accesses to the memory are possible, however they suffer from a performance penalty.

Because all variables and parameters are aligned by the compiler, unaligned memory accesses can only occur if:

- a module uses SYSTEM.PUT, SYSTEM.GET to read data types that are bigger than one byte from an ARRAY OF CHAR.
- SYSTEM.VAL is used to convert a data type into another type.

In such situations the following workaround can be used:

Replace all

```
SYSTEM.GET(adr, long) and
SYSTEM.PUT(adr, long)
```

calls with

```
GET(adr, long) and
PUT(adr, long)
```

and define

```
PROCEDURE GET(adr: LONGINT; VAR long: LONGINT);
BEGIN (* little-endian memory system *)
  long := SYSTEM.GET8(adr) +
          SYSTEM.GET8(adr+1)*100H +
          SYSTEM.GET8(adr+2)*10000H +
          SYSTEM.GET8(adr+3)*1000000H
END GET;

PROCEDURE PUT(adr, long: LONGINT);
BEGIN (* little-endian memory system *)
  SYSTEM.PUT8(adr, CHR(long MOD 100H));
  SYSTEM.PUT8(adr+1, CHR(long DIV 100H MOD 100H));
  SYSTEM.PUT8(adr+2, CHR(long DIV 10000H MOD 100H));
  SYSTEM.PUT8(adr+3, CHR(long DIV 1000000H MOD 100H))
END PUT;
```

This translates word accesses into 4 byte accesses which cannot be misaligned.

The same workaround can be used for halfword accesses.

If SYSTEM.VAL is used, the programmer must ensure that the variable is expanded to the correct size prior to the conversion:

```
VAR b: BYTE; d: LONGINT;
BEGIN
  d := SYSTEM.VAL(LONGINT, LONG(LONG(b)))
END
```

5.6.3 Incompatibilities to Intel Aos

The SYSTEM module defines a few procedures that cannot be ported to ARM Aos:

- SYSTEM.CLI and SYSTEM.STI

  These two procedures disable and enable hardware interrupts. Paco translates these calls into Intel inline assembly code, therefore the backend cannot translate such calls to ARM machine code.
  This is actually a small design flaw in Paco which should be removed.

  Workaround: on ARM Aos use the two procedures

  ```
  AosBoot.DisableInterrupts() and
  AosBoot.EnableInterrupts()
  ```

  to disable or enable hardware interrupts.

- SYSTEM.GETREG

  This procedure stores the value of a hardware register into a variable. Again, this is translated into inline assembly code and cannot be translated.

  To achieve the same functionality, we actually need inline assembly:

```
PROCEDURE -GetRx;
CODE
  MOV   R0, Rx
END GetRx;
```

Note: Rx is just a placeholder for one of the real hardware registers (R0 - R15).

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

The task of this diploma thesis was the development of an Aos Operating System for the DNARD network computer.

This goal has been accomplished. In four months, an ARM backend to Paco including a fully developed assembler plug-in was developed and the whole existing Intel Aos system was ported to ARM Aos.

There are only a few Intel Aos applications that do not run on ARM Aos. This is not due to incompatibilities or limitations of the ARM Aos operating system, but mostly due to carelessly programmed modules that make unaligned memory accesses.

As a side effect, a TFTP server and a TFTP client have been developed.

Writing the backend to Paco (chapter 3) took about six weeks. Of course there were some bugs that had to be fixed during the remaining ten weeks. The development the ARM Aos kernel (see “ARM Aos Kernel” on page 42) also needed about six weeks. The most difficult parts were the memory management (“Memory Manage-

ment” on page 44) and the active object runtime support (“Task Management” on page 49).

6.2 Future Work

Both the compiler and the Aos operating system are evolving very rapidly. Therefore, it is very important to keep both the backend for the ARM processor and the ARM Aos operating system synchronized with the latest developments on the Intel platform.

Once the gap between the Intel and the ARM system becomes too large, it will be necessary to invest a lot of time to restore today’s compatibility.

This work forms an excellent base for future ports to other systems. Each day, more mobile and wireless devices such as handhelds, PDAs and smart phones from Compaq, Psion, Palm, Samsung, and other manufacturers that use ARM processors enter the market. Since ARM Aos has been designed to be as independent as possible from the very beginning of the boot process, porting ARM Aos, for example, to the iPAQ from Compaq should be quite easy:
Only the initial boot loader (see “The Relocator” on page 40), the device drivers and the hardware-dependent parts of the runtime system need to be changed. The memory management, the locking scheme and the active object run-time support are completely hardware independent.

Another possible port would be that to PowerPC processors. The knowledge gathered in this work can be used to implement a PowerPC Aos in considerably less time than four months, because PowerPCs mostly feature an OpenFirmware BIOS and the instruction sets are not very different.

Acknowledgements

Many people have contributed to the success of this project. In particular, I thank the following for their assistance.

First of all Pieter Muller and Patrik Reali, for helping with the implementation of the ARM backend and the porting of the Aos system to ARM. Both of them always tried to fulfill my wishes concerning the re-design of, for example, the intermediate language.

Prof. Jürg Gutknecht for having accepted to be my supervising professor.

My friend, Thomas Frey, for his fair critique and having read this report.

My girlfriend Miriam Kägi, who didn’t see a lot of me during these last four months.

My uncle, Norman Azadian, who donated the better part of a weekend to proofread this report.

Finally, I thank my parents for having supported me during the whole period of study at ETH.

References

[Aos01] Institute for Computer Systems, ETH Zurich - Active Object System, http://www.oberon.ethz.ch/active/

[AosOS] Aos Operating System, http://www.oberon.ethz.ch/native/aos/

[ARM] Advanced RISC Machines Ltd., ARM processor architecture, http://www.arm.com/

[ARM96] ARM Architecture Reference Manual, second edition, Addison-Wesley, 2000

[Cyt91] Ron Cytron et al., An efficient method for computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems, 13(4):451-490, 1991

[DEC97] Digital Research, DIGITAL Network Appliance Reference Design, http://www.research.compaq.com/SRC/iag/

[Hau98] John R.Hauser, SoftFloat, http://www.cs.berkeley.edu/~jhauser/arithmetic/softfloat.html

[IEEE754] IEEE 754: Standard for Binary Floating-Point Arithmetic, http://grouper.ieee.org/groups/754/

[Intel] Intel StrongARM processors, http://developer.intel.com/design/strong/

[Mull00] Pieter Muller, A Multiprocessor Kernel for Active Object-Base Systems, in Lecture Notes in Computer Science 1897, Springer, 2000

[Rea00] Patrik Reali, Structuring a Compiler with Active Objects,
in Lecture Notes in Computer Science 1897, Springer, 2000

[Sig99] Andreas Signer, SharkOberon - Entw
icklung eines Oberon System 3 für den DNARD Network Computer,
http://www.cs.inf.ethz.ch/group/gutknecht/stud_work/1998WS_asigner/

[Sol92] K. Sollins, RFC 1350: The TFTP protocol (revision 2),
http://www.ietf.org/rfc/rfc1350.txt

[WG89] N.Wirth and J.Gutknecht, The Oberon System,
Software - Practice and Experience, 19(9):857-893, Sept. ‘89

[WG92] N.Wirth and J.Gutknecht, Project Oberon - The Design of an Operating System and Compiler, ACM Press, New York, 1992

[Wil91] Graeme Williams, Fast Integer Division,
http://www.poppyfields.net/acorn/tech/division.shtml

APPENDIX A

Paco Intermediate Language

Opcode	Format	Source			Val	Adr	Description
		1	2	3			
load	1M	x			x		Rpc := MEM[R1+val]
		A			x	x	Rpc := MEM[Adr+val]
loadC	1C				x		Rpc := val
					x	x	Rpc := Adr+val
store	M1	x	x		x		MEM[R1+val] := R2
		A	x		x	x	MEM[Adr+val] := R2
in	11	x					Rpc := HwPort(R1) (not supported on ARM)
out	02	x	x				HwPort(R1) := R2 (not supported on ARM)
nop	00						no operation
saveregs	00						save registers currently in use
loadregs	00						restore registers most recently saved with ‘saveregs’
label	0C				x		val = position in source code (positive)
					0		anchor for jumps, contains links list

Opcode	Format	Source			Val	Adr	Description
		1	2	3			
jCC	02C	x	x		x		if CC(R1, R2) -> PC := val CC = {e, ne, lt, le, gt, ge, ae, f, nf}
jmp	0C						PC := val
call	0C					x	subroutine call to Adr
callreg	01	x					subroutine call to R1
syscall	0C				x		subroutine call to SYSTEM_CALL[val]
enter	xx				x	x	enter subroutine with CALLCONV[val] additional info in Adr
exit	xx	x			x		exit subroutine with CALLCONV[val] Src1 := size of parameters, in bytes
trap	0C				x		HALT(val)
tCC	02C	x	x		x		if CC(R1, R2) -> HALT(val)
setCC	12	x	x				if CC(R1, R2) -> Rpc := 1 else Rpc := 0
result	10						Rpc := 1st result of previous call (must follow ‘call’)
result2	10						Rpc := 2nd result of previous call (must follow ‘result’)
ret	01	x					1st result of current call := R1
ret2	01	x					2nd result of current call := R1 (must follow ‘ret’)
pop	10						Rpc := MEM[CP], SP := SP + 4
push	01	x					SP := SP - 4; MEM[SP] := R1
loadsp	01	x					SP := R1
convs	11	x					Rpc := SIGNED_CONVERSION(R1)
convu	11	x					Rpc := UNSIGNED_CONVERSION(R1)
copy	11	x					Rpc := R1 (no conversion)
not	11	x					Rpc := BINARY_NEGATION(R1)
neg	11	x					Rpc := ARITH_NEGATION(R1)
abs	11	x					Rpc := ABSOLUTE(R1)
add	12	x	x				Rpc := R1 + R2
sub	12	x	x				Rpc := R1 - R2
mul	12	x	x				Rpc := R1 * R2
div	12	x	x				Rpc := R1 / R2
mod	12	x	x				Rpc := R1 % R2
and	12	x	x				Rpc := AND(R1, R2)
or	12	x	x				Rpc := OR(R1, R2)
xor	12	x	x				Rpc := XOR(R1, R2)

Opcode	Format	Source			Val	Adr	Description
		1	2	3			
bts	12	x	x				Rpc := OR(R1, 1 << R2)
btc	12	x	x				Rpc := AND(R1, NOT(1 << R2))
bsh	12	x	x				Rpc := BINARY_SHIFT(R1, R2)
ash	12	x	x				Rpc := ARITH_SHIFT(R1, R2)
rot	12	x	x				Rpc := ROTATE(R1, R2)
phi	12	x	x				Rpc := PHI(R1, R2)
kill	01	x					kill R1 (used in conjunction with 'phi')
move	03	x	x	x			MEM[R1..R1+R3] := MEM[R2..R2+R3]
inline	XX						inline assembler code
case	XX	x	x	x			CASE on R1; R2 = min; R3 = max
casel	XX	x			x		caseline: R1 = case; val = case-offset
casee	XX	x					caseelse: R1 = case

Remarks:

- an *instruction* is encoded in a record defined in PCLIR.Mod:

```
TYPE
  InstructionAttribute* = POINTER TO RECORD   END;
  Instruction* = RECORD
    op*: Opcode;
    src1*, src2*, src3*: Register;
    val*: LONGINT;        (*src for constant values, used by form*C instr. *)
    adr-: PCM.Attribute;  (* reference to absolute addresses *)
    barrier-: LONGINT;    (* pos of previous barrier in code *)

    suppress*: BOOLEAN;(* suppress this instruction*)

    dst*: RECORD(* form1X only *)
        count*: SHORTINT;   (* counts use of this register *)
        size-: Size;        (* size of this register *)
        signed-: BOOLEAN;   (* TRUE if the value in dst is signed *)
    END;
    info*: InstructionAttribute;(* link to user-defined information *)
  END;
```

- the *type* of the result is determined by Instruction.dst.size and Instruction.dst.signed. The type of the source operands can be determined by finding the instruction that defines a source operand and looking at the dst.size and dst.signed fields there.

APPENDIX B

Aos ARM Assembly Language

This appendix lists all supported mnemonics and directives, including their syntax and semantics.

B.1 Mnemonics

Elements in curly brackets {} are optional. * indicates privileged instructions.

Mnemonic	Syntax	Semantics
Branch instructions		
B	B *label*	branch to *label*
BL	BL *label*	branch to *label* and set LR
Data-processing instructions		
ADC	ADC rD, rA, *shifter_operand*[a]	rD := rA + *shifter_operand* + carry
ADD	ADD rD, rA, *shifter_operand*	rD := rA + *shifter_operand*
AND	AND rD, rA, *shifter_operand*	rD := rA AND *shifter_operand*
BIC	BIC rD, rA, *shifter_operand*	rD := rA AND NOT(*shifter_operand*)
EOR	EOR rD, rA, *shifter_operand*	rD := rA XOR *shifter_operand*

Table B-1. Aos ARM Assembler mnemonics

Mnemonic	Syntax	Semantics
MOV	MOV rD, *shifter_operand*	rD := *shifter_operand*
MVN	MVN rD, *shifter_operand*	rD := NOT(*shifter_operand*)
ORR	ORR rD, rA, *shifter_operand*	rD := rA OR *shifter_operand*
RSB	RSB rD, rA, *shifter_operand*	rD := *shifter_operand* - rA
RSC	RSC rD, rA, *shifter_operand*	rD := *shifter_operand* - rA - NOT(carry)
SBC	SBC rD, rA, *shifter_operand*	rD := rA - *shifter_operand* - NOT(carry)
SUB	SUB rD, rA, *shifter_operand*	rD := rA - *shifter_operand*
CMN	CMN rA, *shifter_operand*	update flags after rA + *shifter_operand*
CMP	CMP rA, *shifter_operand*	update flags after rA - *shifter_operand*
TEQ	TEQ rA, *shifter_operand*	update flags after rA XOR *shifter_operand*
TST	TST rA, *shifter_operand*	update flags after rA AND *shifter_operand*
Multiply instructions		
MLA	MLA rD, rA, rB, rC	rD := rA * rB + rC
MUL	MUL rD, rA, rB	rD := rA * rB
SMLAL	SMLAL rDlow, rDhigh, rA, rB	rDhigh:rDlow := rA * rB + rDhigh:rDlow
SMULL	SMULL rDlow, rDhigh, rA, rB	rDhigh:rDlow := rA * rB
UMLAL	UMLAL rDlow, rDhigh, rA, rB	rDhigh:rDlow := rA * rB
UMULL	UMULL rDlow, rDhigh, rA, rB	rDhigh:rDlow := rA * rB + rDhigh:rDlow
Status register access instructions		
MRS	MRS rD, CPSR	move CPSR to rD
	MRS rD, SPSR	move SPSR to rD
MSR(*)	MSR CPSR_*fields*[b], *#imm*	move *imm* to certain *fields* of the CPSR
	MSR CPSR_*fields*, rS	move rS to certain *fields* of the CPSR
	MSR SPSR_*fields*, *#imm*	move *imm* to certain *fields* of the SPSR
	MSR SPSR_*fields*, rS	move rS to certain *fields* of the SPSR
		Note: accessing the SPSR while in User/System mode is unpredictable.
Load and store instructions		
LDR	LDR rD, *address*[c]	rD := MEM[*address*..*address*+3]
LDRB	LDRB rD, *address*	rD := zero_ext(MEM[*address*])
LDRH	LDRH rD, *address_h*[d]	rD := zero_ext(MEM[*address_h*..*address_h*+1])
LDRSB	LDRSB rD, *address_h*	rD := sign_ext(MEM[*address_h*])
LDRSH	LDRSH rD, *address_h*	rD := sign_ext(MEM[*address_h*..*address_h*+1])

Table B-1. Aos ARM Assembler mnemonics

Mnemonic	Syntax	Semantics
STR	STR rS, *address*	MEM[*address*..*address*+3] := rS
STRB	STRB rS, *address*	MEM[*address*] := rS[7:0]
STRH	STRH rS, *address_h*	MEM[*address_h*..*address_h*+1] := rS[15:0]
Load and store multiple instructions		
LDM	LDM*addr_mode*[e] rN{!}, *registers*[f]{^}	load a subset, or possibly all, of the general purpose registers from memory.
STM	STM*addr_mode* rN{!}, *registers*{^}	store a subset, or possibly all, of the general purpose registers to memory
Semaphore instructions		
SWP	SWP rD, rM, [rN]	rD := MEM[rN..rN+3]; MEM[rN..rN+3] := rM
SWPB	SWPB rD, rM, [rN]	rD := zero_ext(MEM[rN]); MEM[rN] := rM[7:0]
Exception-generating instructions		
SWI	SWI #*immed_24*	raise software interrupt *immed_24*
Coprocessor instructions		
CDP*	CDP coproc, opcode1, crD, crN, crM, opcode2	instructs coprocessor ‘coproc’ to perform an coprocessor-specific operation
LDC*	LDC coproc, crD,*address_cp*[g]	loads memory data from MEM[*address_cp*] to the coprocessor ‘coproc’
MCR*	MCR coproc, opcode1, rD, crN, crM{, opcode2}	pass the value of register rD to coprocessor ‘coproc’
MRC*	MRC coproc, opcode1, rD, crN, crM{, opcode2}	causes the coprocessor ‘coproc’ to transfer a value to register rD
STC*	STC coproc, crD, *address_cp*	stores data from the coprocessor ‘coproc’ to MEM[*address_cp*]
Additional instructions		
ADR	ADR rD, *label*	rD := ADR(*label*)

Table B-1. Aos ARM Assembler mnemonics

a. The *shifter_operand* has one of the following formats:
   1. #<immediate>
      the value of the shifter operand is the immediate value
   2. rB
      the value of the shifter operand is the value of register rB
   3. rB, *shift shifter*
      shift = (LSL, LSR, ASR, ROR) and *shifter* = (#*imm*, rS)
      the value of the shifter operand is the value of register rB shifted by the value of *shifter.*
   4. rB, RRX
      the value of the shifter operand is the value of register rB rotated right by one. The carry flag is used as the 33rd bit

b. *fields* is a sequence or more of "c", "x", "s", "f". These flags affect which bits in the PSR will be replaced:
   c: bits [7:0]
   x: bits [15:8]
   s: bits [23:16]
   f: bits [31:24]

c. *address* has one of the following formats:
   1. [rN, #+/-*offset_12*]{!}
      the memory address is formed by adding/subtracting *offset_12* to/from rN. If "!" is specified, the calculated value is written back to rN.
   2. [rN, +/-rM]{!}
      the memory address is formed by adding/subtracting the value of rM to/from rN. If "!" is specified, the calculated value is written back to rN.
   3. [rN, +/-rM, *shift*, #*imm*]{!}
      the memory address is formed by adding/subtracting the value of rM shifted by *imm* to/from rN. If "!" is specified, the calculated value is written back to rN.
   4. [rN], #+/-*offset_12*
      the memory address is the value of rN. After accessing the memory, *offset_12* is added to or subtracted from rN and the calculated value written back to rN.
   5. [rN], +/-rM
      the memory address is the value of rN. After accessing the memory, the value of rM is added to or subtracted from rN and the calculated value written back to rN.
   6. [rN], +/-rM, *shift*, #*imm*
      the memory address is the value of rN. After accessing the memory, the value of rM shifted by *imm* is added to or subtracted from rN and the calculated value written back to rN.

   with *shift* = (LSL, LSR, ASR, ROR, RRX)

d. *address_h* has one of the following formats:
   1. [rN, #+/-*offset_8*]{!}
      the memory address is formed by adding/subtracting *offset_8* to/from rN. If "!" is specified, the calculated value is written back to rN.
   2. [rN, +/-rM]{!}
      the memory address is formed by adding/subtracting the value of rM to/from rN. If "!" is specified, the calculated value is written back to rN.
   3. [rN], #+/-*offset_8*
      the memory address is the value of rN. After accessing the memory, *offset_8* is added to or subtracted from rN and the calculated value written back to rN.
   4. [rN], +/-rM
      the memory address is the value of rN. After accessing the memory, the value of rM is added to or subtracted from rN and the calculated value written back to rN.

e. *addr_mode* is one of IA, IB, DA, DB and determines the way the address is formed:
   IA - increment after: the registers values are written to MEM[rN..rN+4*nregs-4]
   IB - increment before: the registers values are written to MEM[rN+4..rN+4*nregs]
   DA - decrement after: the register values are written to MEM[rN-4*nregs+4..rN]
   DB - decrement before: the register values are written to MEM[rN-4*nregs..rN-4]
   with *nregs* = number of registers in *registers* (see below)

f. *registers* is a list of registers enclosed in curly brackets, separated by commas.
   Ranges of registers are also supported, e.g. {R1-R10, FP, SP}

g. *address_cp* has one of the following forms:
   1. [rN, #+/-*offset_8*]{!}
      produces a sequence of consecutive memory addresses, the first address is calculated by adding or subtracting four times *offset_8* to or from the value of rN. If "!" is specified, the first address is written back to rN.
   2. [rN], #+/-*offset_8*
      produces a sequence of consecutive memory addresses, the first address is the value of rN. The base register rN is updated with four times the value of *offset_8* added to or subtracted from rN.
   3. [rN], *option*
      produces a sequence of consecutive memory addresses, the first address is the value of rN. The *option* field, an 8-bit immediate value, can be used to pass additional information to the coprocessor.

B.2 Directives

Curly brackets {} indicate optional repetition.

Directives	Syntax	Description
DCD	DCD imm_32{, imm_32}	Embeds a (list of) 32-bit immediate value(s) into the code.
	DCD *global_variable*	Embeds the address of a global variable into the code. The address will be automatically fixed by the loader. Use LDR rD, *global_variable* to load the address of the variable into a register.
DCW	DCW imm_16{, imm_16}	Embeds a (list of) 16-bit immediate value(s) into the code. **Important**: it's the programmer's responsibility to ensure that the following code is word-aligned!
DCB	DCB imm_8{, imm_8}	Embeds a (list of) 8-bit immediate value(s) into the code. imm_8 can have one of the following formats: 1. decimal value, e.g. 37 2. hexadecimal value, e.g. 0FFH 3. character, either quoted ('A') or direct (41X)
	DCB quoted_string	Embeds a 0X-terminated string into the code. Example: DCB "Hello World !" **Important**: it's the programmer's responsibility to ensure that the following code is word-aligned!
DEFINE	DEFINE *ident* = rX	Defines an alias *ident* for register rX.

Table B-2. ARM Aos Assembler directives

APPENDIX C

ARM Aos System Calls

The following system calls are available on ARM Aos:

System call	Procedure signature
200	SoftFloat.RealAdd
201	SoftFloat.RealSub
202	SoftFloat.RealMul
203	SoftFloat.RealDiv
204	SoftFloat.RealNeg
205	SoftFloat.RealAbs
206	SoftFloat.RealEntier
207	SoftFloat.LIntToReal
208	SoftFloat.RealToLInt
209	SoftFloat.RealToLReal
210	SoftFloat.RealEq
211	SoftFloat.RealLe
212	SoftFloat.RealLt
213	SoftFloat.RealNe
214	SoftFloat.RealGe
215	SoftFloat.RealGt
220	SoftFloat.LRealAdd
221	SoftFloat.LRealSub

Table C-1. ARM Aos system calls

System call	Procedure signature
222	SoftFloat.LRealMul
223	SoftFloat.LRealDiv
224	SoftFloat.LRealNeg
225	SoftFloat.LRealAbs
226	SoftFloat.LRealEntier
227	SoftFloat.LIntToLReal
228	SoftFloat.LRealToLInt
229	SoftFloat.LRealToReal
230	SoftFloat.LRealEq
231	SoftFloat.LRealLe
232	SoftFloat.LRealLt
233	SoftFloat.LRealNe
234	SoftFloat.LRealGe
235	SoftFloat.LRealGt
241	FastDivMod.DivMod
242	FastDivMod.DivMod
246	AosActive.Unlock
247	AosActive.Lock
249	AosActive.Await
250	AosActive.CreateProcess
251	AosStorage.NewArr
252	AosStorage.NewSys
253	AosStorage.NewRec

Table C-1. ARM Aos system calls

Notes:

- the system calls 246-253 are identical to those on Intel Aos
- the system calls 241 & 242 both refer to the same procedure.
  System call 241 is generated for DIV operators and the result is returned in R0.
  System call 242 is generated for MOD operators; here the result is returned in R1.
- the “system calls” 254 and 255 are used by the compiler and module loader to fix up local procedure calls (254) and case tables (255)

APPENDIX D

List of Changes

This appendix lists all modules that needed to be adapted to ARM Aos and those that are not available on ARM Aos. The ARM Aos release contains over 500 modules, 15 of which are completely new and 23 are ported. The modules that are not available on ARM Aos are mostly device drivers for devices that are not present in the DNARD network computer.

Modules in topological order	Status	Remarks
ARM.Relocator.Mod	new	Bootstrapping
ARM.AosBoot.Mod	new	
ARM.FastDivMod.Mod	new	DIV/MOD support
ARM.SoftFloat.Mod	new	Floating-point support
ARM.Processor.Mod	new	
ARM.AosLocks.Mod	new	
ARM.AosOut.Mod	new	
ARM.AosMemory.Mod	new	
ARM.AosStorage.Mod	ported	Garbage collector is portable
ARM.AosInterrupts.Mod	new	
ARM.AosModules.Mod	ported	System call table needs to be adapted
ARM.AosTrap.Mod	ported	Register names need to be adapted
ARM.AosTicks.Mod	new	
ARM.AosActive.Mod	ported	Task-switching is not protable
ARM.AosExceptions.Mod	new	
ARM.AosSMP.Mod	new	

Modules in topological order	Status	Remarks
ARM.AosFido.Mod	ported	Register names need to be adapted
ARM.AosClock.Mod	new	
ARM.AosKeyboard.Mod	ported	Contains assembly code
ARM.AosMousePS2.Mod	ported	Memory-mapped I/O not portable
ARM.Aos.V24.Mod	ported	Memory-mapped I/O not portable
ARM.AosDisplayLinear.Mod	new	Display driver
ARM.Aos.Math.Mod	ported	Intel-versions contain assembly code, ARM versions are portable
ARM.Aos.MathL.Mod	ported	
ARM.AosBeep.Mod	ported	Memory-mapped I/O not portable
ARM.AosCS8900.Mod	new	Ethernet adaptor driver
ARM.AosIP.Mod	ported	Contains assembly code
ARM.AosLoader.Mod	ported	System-calls need to be adapted
ARM.AosConsole.Mod	ported	Adapted boot-process
ARM.WMObjectTracker.Mod	ported	Register names need to be adapted
ARM.WM.Input.Mod	ported	TimeUnit contant adapted
ARM.Fonts.Mod	ported	Intel-version contains assembly code
ARM.Aos.System.Mod	ported	Register names need to be adapted
ARM.Aos.Centronics.Mod	ported	Memory-mapped I/O not portable
ARM.AosMemInfo.Mod	ported	Architecture-dependant
ARM.AosProf.Mod	ported	Register names need to be adapted
ARM.AosTestLocks.Mod	ported	Interrupt enabling/disabling has changed
ARM.AosProfiler.Mod	ported	Register names need to be adapted
Not available on ARM Aos		
Aos.PCI.Mod, PCITools.Mod		PCI bus not present
AosATADisks.Mod, AosSCSI.Mod, Adaptec7Script.Mod, AosAdaptec7.Mod, NCRScript.Mod, AosSymbio-sASPI.Mod, AosSymbiosDisks.Mod		Disk not present
AosUsb.Mod, AosUsbUhci.Mod, AosUsbStorage.Mod, AosUsbMouse.Mod		USB not present
AosDisplayPermedia2.Mod, AosS3Aurora64.Mod		Graphics adaptors not present
Aos3Com509.Mod, Aos3Com90x.Mod, AosRTL8139.Mod		Ethernet adaptors not present
MathH.Mod		Contains assembly code

Modules in topological order	Status	Remarks
Aos.Diskette.Mod, Diskette.Mod, Backup.Mod, Partitions.Mod, FATFiles.Mod, DOS.Mod		Diskette not present
Snapshot.Mod		Contains assembly / memory-mapped I/O
Asteroids.Mod, ComplexMath.Mod, TestComplex.Mod, MatrixMath.Mod, TestMatrix.Mod		Do not compile with Paco (also on Intel)