# Code generation for the attribute evaluator of the protocol engine grammar processor unit

Research Report
ISSN 0167-9708
Coden: TEUEDE

Eindhoven
University of Technology
Netherlands

Faculty of Electrical Engineering

# Code Generation for the Attribute Evaluator of the Protocol Engine Grammar Processor Unit

by
R.H.J. Bloks

EUT Report 93-E-271
ISBN 90-6144-271-0
March 1993

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
Eindhoven, The Netherlands

ISSN 0167-9708 Coden: TEUEDE

# Code generation for the attribute evaluator of the protocol engine grammar processor unit

by

R.H.J. Bloks

EUT Report 93-E-271
ISBN 90-6144-271-0

Eindhoven
March 1993

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Bloks, R.H.J.

Code generation for the attribute evaluator of the
protocol engine grammar processor unit / by R.H.J. Bloks.
- Eindhoven : Eindhoven University of Technology, Faculty
of Electrical Engineering. - Fig., tab. - (EUT report,
ISSN 0167-9708 ; 93-E-271)
Met lit. opg.
ISBN 90-6144-271-0
NUGI 832
Trefw.: compilers ; codegeneratie / systeemontwikkeling ;
telecommunicatie.

# Code generation for the attribute evaluator of the protocol engine grammar processor unit

## Abstract

report is one of the results of the protocol engine research project, which is targeted towards the automatic generation of hardware implementations for modern complex data communication protocols. The word 'automatic' implies the use of computers to accomplish the task which in turn requires a formal language that can be used to describe these protocols. Such a language and a compiler for it have now been created. The language is based on an extension of context-free grammars and is called ProGrIL (Protocol Grammar Interface Language).

In this document, the process of code generation for the attribute expression evaluation will be explained. As described in the Ph.D. thesis (see [Bloks93b]), a protocol grammar consists of a set of attributes or variables, a set of symbols, a set of production rules and attribute relations between the attributes of symbols appearing in the rules. Whenever a certain symbol is parsed, the expressions for its attributes must be evaluated. These protocol grammars can be implemented on protocol pushdown automata, which are extensions of the standard pushdown automaton. One of the extensions to this basic automaton is the attribute evaluator, which is in fact a custom designed processor capable of performing all operations required for the manipulation of attributes of the protocol grammar.

All expressions given in the grammar are assignment expressions, using constants, declared attributes, and system variables or functions. These expressions must be translated directly into machine language code for the attribute evaluator by the ProGrIL compiler. This code will be referred to as explicit code. In contrast to the explicit code, there is also implicit code, which is necessary for correct operation of the system, but is not directly 'programmed' by the writer of the grammar. Instead, this code is implicitly defined by and extracted from the grammar. It is used to do attribute memory management, passing attribute values, and other system functions.

*Keywords*: compilers, telecommunication systems, code generation

- Bloks, R.H.J.
  Code generation for the attribute evaluator of the protocol engine grammar processor unit.
  Eindhoven: Faculty of Electrical Engineering, Eindhoven University of Technology, 1993.
  EUT Report 93-E-271.

- Address of the author:
  R.H.J. Bloks
  van Norenburchstraat 2
  5622 KP Eindhoven, The Netherlands

# Table of Contents

# Introduction

The exchange of data between computers is governed by common rules dictating the format and meaning of message units (packets) used for the communication. Two computers can only exchange data if they use the same set of rules, which is usually referred to as a communication protocol. Modern protocols that are used on large computer networks are flexible and very complex systems, which is why they are usually implemented in software running on the communicating computers themselves. A big disadvantage of software implementations is that they are inherently slower than special hardware solutions. With increasing demands for high speed communication and extremely high speed network technology (using fiber optics) software can no longer utilize available bandwidth and hardware solutions must be found. Because of their complexity this is not a trivial matter and it would be very desirable to have a system that can construct an implementation automatically.

The protocol engine Ph.D. research project is directed towards the automatic creation of a hardware implementation for any protocol specified by a formal description. The basic idea is that once a protocol has been conceived, it is written down or converted into a formal implementation description and then automatically processed, resulting in an output suitable for a low level silicon compiler.

This means that a formal language is required in which protocols can be described. Furthermore, it must be possible to translate any valid description in this language into hardware, which implies that a target architecture and a mapping model from language to architecture (language semantics) must be defined. Obviously, the target architecture and description language are closely related.

The language that was created is based on standard context-free grammars (ref. [Aho72], [Aho86],[Denning78], [Kain72], [Knuth68] and[Lewis81] for the theory of languages, automata and computation). Grammars are mathematical models for the specification of formal languages and can be used to describe the communication language of a protocol. The advantage is that the implementation architecture and semantic model are known (pushdown automaton) and mathematically provable.

In order to allow an easy and practical description of modern protocols, some extensions must be made to the standard context-free grammars (see [Bloks93b], [Haas85] and [Anderson85a] for the reasons and some possible ways of doing this). The resulting protocol grammar has to be formally defined and a language definition given. Then the implementation architecture (the pushdown automaton, see [Denning78] and [Lewis81]) and the mapping model must also be extended and formally defined. Finally, the correctness of the mapping must be shown. This is all done in [Bloks93b], but not found in any work done by others in this field.

When a communication protocol is defined in the form of a (set of interconnected) protocol grammar(s), it can be translated directly into a (set of) extended pushdown automata interconnected as indicated by the grammars to obtain a system whose external behaviour is precisely as specified by the grammars. Such a system therefore implements the communication protocol. Some specialized protocol functions may be implemented in dedicated hardware units outside the pushdown automata for reasons of efficiency. The abstract version of the extended pushdown automaton is called a protocol pushdown automaton, and its physical implementation is the grammar processor. A protocol engine consists of a network of interconnected grammar processors and some dedicated hardware.

Each grammar processor is 'programmed' for a certain protocol by storing the grammar parse tables and microcode for context changes/updates in its memory. Context changes occur when the protocol uses variables to store information about the past and uses them to guide further actions. The concept of variables is incorporated in the grammar by the extension with attributes. Every symbol can have zero or more associated attributes whose values are sent and received along with the symbol and store context information that is important for the processing of that symbol. When a symbol is processed, the values of attributes can change. In the grammar this is indicated by assignment expressions to the attributes.

To store and update the context of all symbols, the grammar processor contains a microprogrammable processing unit, called the attribute evaluator. Every symbol of every rule of the grammar has a small program stored in the attribute evaluator, which does all the necessary context updating when executed. The parse tables and microprograms are all created automatically by a compiler that derives this information from the protocol grammars. This compiler is part of a protocol engine design system that is currently under development.

This document describes the architecture and global operation of the attribute evaluator and how the compiler generates the microcode for the expressions specified in the grammar description.

# Code generation for the attribute evaluator of the protocol engine grammar processor unit

## 1 The grammar processor

The grammar processor is a design for a physically implementable version of the rather abstract protocol pushdown automaton described in the Ph.D. thesis (see chapters 4 and 5 of [Bloks93b]). Its operation is also explained there in some detail. The diagram of the top level design is reprinted in figure 1.

### 1.1 Top level overview

The *pushdown controller* implements the actual parsing mechanism. It contains the parse tables generated by the compiler, by means of which it can decide what action to perform based on the symbol on top of the parse stack, the symbols available at the inputs, its internal status and a boolean rule enable flag computed by the attribute evaluator. The pushdown controller controls all other parts of the grammar processor.

The *parse stack* stores the symbols and their action procedure numbers. The symbol at the top is used by the pushdown controller, and the action procedure number refers to a small program in the attribute evaluator, which contains the microcoded version of the expressions to be evaluated for the associated symbol on the stack, as specified in the ProGrIL description.

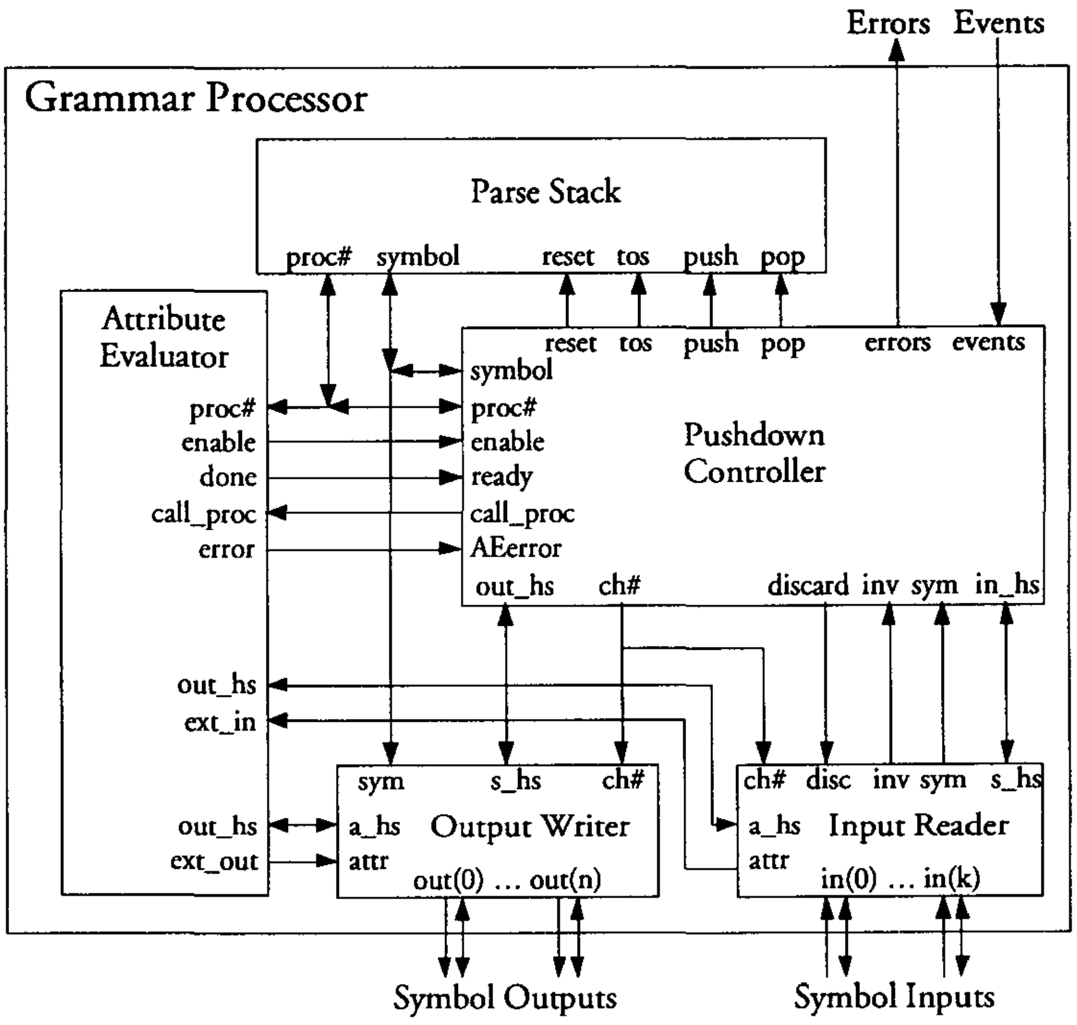

**Figure 1.** *Top level design of the grammar processor.*

The ***attribute evaluator*** contains a RAM (Random Access Memory) to store the attribute values, and a custom microprocessor to perform the necessary operations on these attributes by executing small programs which are compiled by the ProGrIL compiler and stored in a program ROM (Read Only Memory) where they can be referred to by a unique number, the so called action procedure number.

Values of attributes can be sent to the output writer or received from the input reader when required. This I/O is also microprogrammed in the procedures and is necessary to implement channel communications of symbols with attributes.

[Bloks93b]The *output writer* interfaces a number of output channels to the rest of the grammar processor. When a message has to be sent on an output channel, the pushdown controller specifies the channel number, the parse stack delivers the symbol and the attribute evaluator generates the attribute values. The symbol and values are combined and queued for output at the specified channel by the output writer.

The *input reader* interfaces a number of input channels to the rest of the grammar processor. When a message is received on an input channel, it is queued for processing. When the pushdown controller specifies that a message from a channel can be processed, it outputs the channel number. The attribute evaluator is started by invoking the action procedure for the input symbol, and will receive and process the attribute values from the corresponding queue.

## 1.2 The attribute evaluator

Since the goal of this document is to describe the mechanism used to generate the microcode programs for the attribute evaluator, a more detailed description of this processor shall now be given. Its top level design is shown in figure 2.

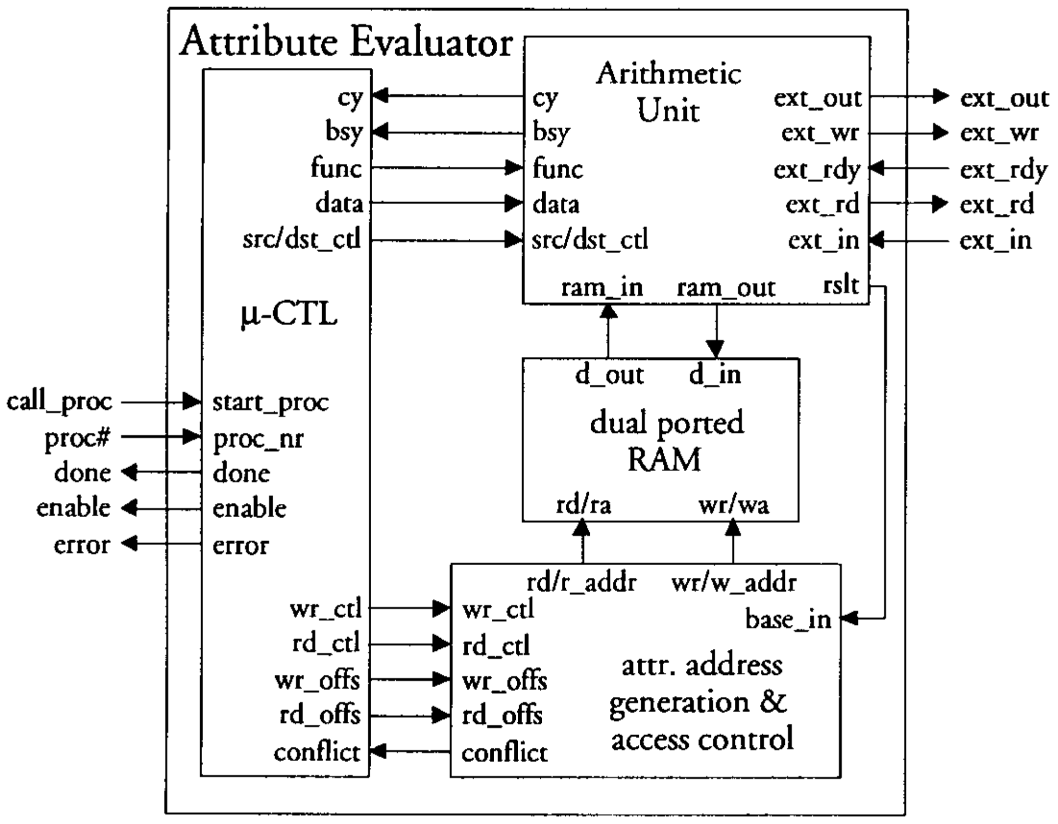

**Figure 2.** *The architecture of the attribute evaluator.*

The 4 main parts are the attribute storage, an access address generation unit, a processing unit and a control unit containing the action procedure microcode. The Attribute Evaluator has been designed as a multistage pipelined processor unit to increase performance. A high performance of this unit is of the utmost importance for obtaining very high throughput protocol engines. For this reason, it was decided that 3-operand instructions were to be used (2 sources and 1 destination).

The attribute storage is implemented as a dual ported RAM with a single read port and a single write port which are mutually independent. Because of the pipelining, this means that one operand can be read and the result of a previous operation can be written simultaneously.

The micro controller (μ-CTL) contains a ROM in which all action procedures are stored in the form of microcode. A lookup table translates a procedure identification number into a start address in this ROM. The controller starts executing instructions from that address, while generating control and data vectors for the other parts of the Attribute Evaluator.

### 1.2.1 Instruction execution and pipelining

Instruction execution takes 5 (pipeline) cycles:

| | |
|---|---|
| cycle 1: | μ-CTL generates control vectors and computes next instruction address. |
| cycle 2: | Address generators compute addresses for RAM access and update base address registers (if necessary). |
| cycle 3: | RAM and/or external input are accessed for source operand (if necessary). |
| cycle 4: | ALU processes data and computes a result (fast operations). |
| cycle 5: | Result is written to destination. |

As long as no conflicts occur in the pipeline, it remains full. This means that one instruction is completed every cycle. Conflicts may hold up or clear (part of) the pipeline structure, thereby decreasing performance, but all conflicts are automatically resolved entirely in hardware (i.e. the pipelining effect is transparent to software running on the processor).

All ALU operations may take one or more cycles. For example an ADD will probably take one cycle, but a multiply or bit shift over multiple bits will take longer unless the ALU contains special hardware for these operations. The arithmetic unit can inform the μ-CTL that the current operation will not complete at the next clock by means of the *bsy* lines. The pipeline will then be put on hold until the ALU has finished its operation.

### 1.2.2 Address generators

Addresses of attributes in the storage are generated by two independent address generators, one for the read access and one for the write access. This must be done on a register direct with displacement base (in local environments) or by absolute addressing (in the global environment). This is explained in more detail in chapter 5 of [Bloks93b]. Global attributes can be accessed using a fixed (absolute) address in memory, and local attributes must be accessed by an offset to a base register, because local environments are dynamically allocated and the positions of local attributes are therefore not known in advance. The **ind** function can be implemented by setting the base register to its argument and then using offset 0. An array element must be accessed by computing its address, storing it in the base register, and then using offset 0. This is called indirect addressing. To allow indirect and direct addressing simultaneously (for 2-source operations) it is necessary to have 2 base registers: one always contains the start of the current local environment (*standard base register*) and the other can be used for indirect addressing (*alternate base register*). This has resulted in the schematic diagram shown in figure 3.

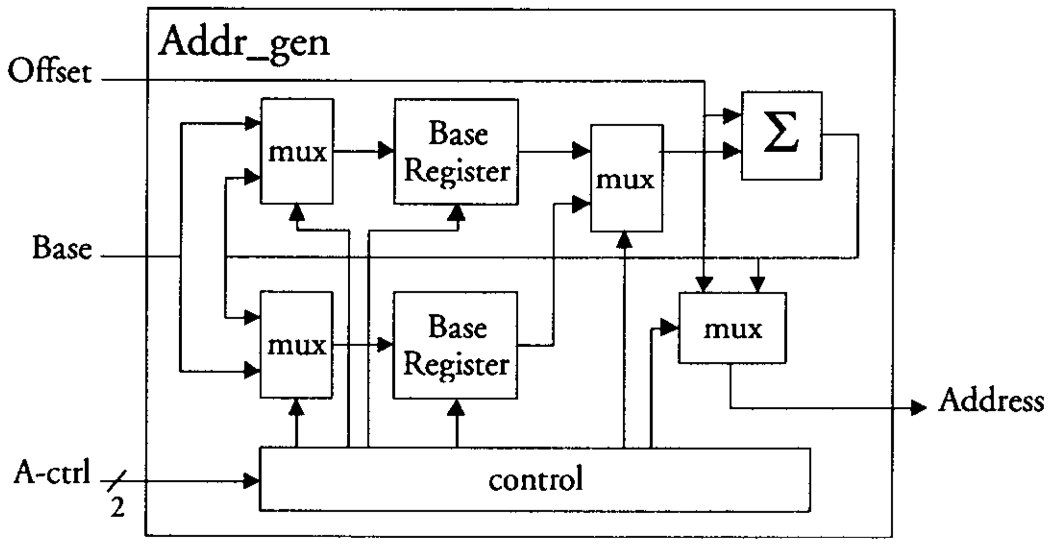

**Figure 3.** *A single address generator.*

The *address generators* will compute access addresses for attributes under control of the μ-CTL. Since destination addresses are not needed until the fifth cycle, they will be delayed before they are sent to the RAM. After an address is used, the base register can be loaded with the generated address. This represents a pre-add adjust mode. Since the offset can be positive or negative, it is a generalization of both pre-decrement and pre-increment modes. This mode is very useful for copying blocks of memory, which is required for some operations on structures and arrays, as well as data I/O on the external input/output of the attribute evaluator. Summarizing, the address generators offer the following functionality ('basereg' is either standard or alternate base register):

**Output value (generated address):**

- Register based indirect addressing with displacement:
  (address_out = basereg + offset_in)
- Absolute addressing:
  (address_out = offset_in)

**Base register contents:**

- hold basereg at end of cycle:
  (basereg := basereg)
- adding offset to basereg at end of cycle:
  (basereg := basereg + offset_in)
- loading basereg with absolute value at end of cycle:
  (basereg := base_in)

### 1.2.3 The arithmetic unit

The arithmetic unit contains a processing unit (ALU), a timer (TIME), two source selectors for the operands and several registers. The value for each operand can come from any of 7 sources, and is first stored in a pipeline register ('opr# reg') before it is used by the ALU.

The 'alu_reg' always contains the value computed by the ALU during the previous instruction and it used to resolve many source/destination pipeline conflicts. It is considered a pipeline register, since it also holds the destination operand value when such a value has to be written to RAM.

The temporary register ('temp_reg') makes efficient implementation of more complex expression possible because it can be used to hold intermediate values during expression evaluation. This register can be used freely (it is only written when explicitly specified as the destination operand of an instruction).

The carry output ('cy') plays a special role, because its value is sent to the pushdown controller as the 'enable' value at the end of a rule enable condition procedure. The ALU can execute many different operations. Some are very general and can be found in most general purpose processors. Others are specialized and not present in any other processor. For a list of all high level (ProGrIL) operators and functions, see [Bloks93a]. A list of ALU functions is provided later in this document.

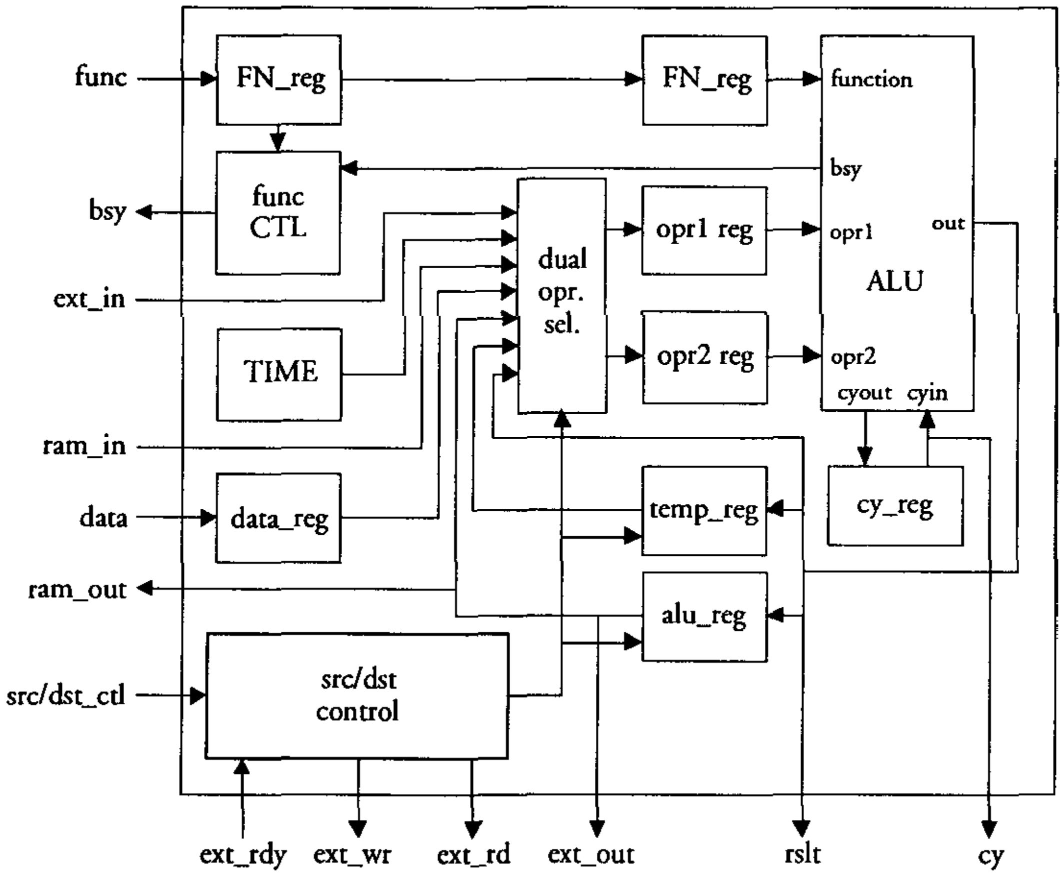

**Figure 4.** *Internal architecture of the arithmetic unit.*

Because the ALU function code and constant operands (coded in instruction) are generated in cycle 1 while the ALU operates in cycle 3, an additional pipeline register is required for both values ('FN_reg' and 'data_reg').

## 2 Attribute evaluator control functions

Expression operands can be divided into 5 categories:

| | |
|---|---|
| category 1: | Constants |
| category 2: | Internal registers |
| category 3: | Global attributes, excluding array elements |
| category 4: | Local attributes of the current environment, excluding array elements |
| category 5: | Local attributes in other environments and array elements |

Each operand category requires another method of accessing:

| | |
|---|---|
| category 1: | immediate value, coded in ROM |
| category 2: | implied addressing |
| category 3: | absolute addressing, coded in ROM |
| category 4: | register direct with displacement, using base register and coded offset in ROM |
| category 5: | register with displacement indirect, using base register and ROM coded offset to find value for alternate base register. Then access target using offset 0 to alternate base (double category 4 access). |

The structure of the processor allows a maximum of two operands for every function performed by the ALU. These operands can each be taken from several sources, with the following restrictions:

- Only one of the operands may be of category 1
- Only one of the operands may be from category set {3, 4, 5}

These restrictions are the result of hardware limitations. There is only one data field in the control ROM that can contain data for the ALU, and only one read access can be made to the attribute RAM in each cycle. In fact, these restrictions do not hold if both arguments are identical, but that situation would be rare, and will therefore not be considered a special case. The first restriction is no problem, since any operation on two constants can be precomputed at compile time, and thus eliminated. The second restriction is a problem, and requires preloading of one operand into either the temporary register or the ALU bus register. It always requires at least one additional cycle.

General MNEMONIC instruction format:

```
<alu function>
          [ <access mode><source1>
                    [ , <access mode><source2> ]
          ]
          [ , <access mode><dest> ]
| <flow control code>  <flow control data>
```

Since there is at most one destination, and attribute RAM has a separate and fully independent write port, there are no destination access restrictions.

The access mode indicates the category of the operand. The following mnemonic symbols are defined for these categories:

| SYMBOL | ACCESS MODE | OPERATOR CATEGORY |
|---|---|---|
| # | immediate | (category 1) |
| % | implied | (category 2) |
| <none> | absolute | (category 3) |
| ^ | reg. direct + displ. | (category 4) |
| ^^ | alt. reg. direct + displ. | (converted category 5) |

**Table 1.** *Access Mode Symbols.*

For the register based access modes (4 and 5) it is possible to load the base register after an access with the address generated during that access. Actually, the offset used during the last access is added to the value in the base register and stored back in it. This can be considered as a pre-increment mode with a variable increment value. In the remainder, this mode will be referred to as 'pre-add' mode, and it will be denoted by a '+' sign preceding the access mode symbol.

Numbers can be presented in decimal or hexadecimal format. In the mnemonic output listings, all numbers will be in the hex format. To emphasize this, they will be preceded by a '$' sign, which is commonly used for this mode. For better legibility, numbers in the range 0..9 are not preceded by a '$' since these are the same in both radix systems.

Notations:

| | |
|---|---|
| • s, s1, s2 | = source operands |
| • d | = destination |
| • w | = width of system word and memory |
| • a | = address offset (jumps) |
| • p | = procedure number |
| • v | = constant value |
| • a[b] | = bit b of value of a (0 = LSB) |

## 2.1 Source operands

| BITS | OPERAND | MNEMONIC NAME |
|---|---|---|
| 000 | ALU bus | %ALU |
| 001 | Temp. register | %TEMPREG |
| 010 | Attribute RAM | <num> OR ^<num> OR ^^<num> |
| 011 | ROM | #<num> |
| 100 | External input | %EXTIN |
| 101 | System timer | %SYSTIME |

**Table 2.** *Bit coding and mnemonic names for source operands.*

## 2.2 Destination operands

| BITS | OPERAND | MNEMONIC NAME |
|---|---|---|
| 0000 | ALU bus (<none>) | %ALU |
| 0001 | Temp. register | %TEMPREG |
| 0010 | Attribute RAM | <num> OR ^<num> OR ^^<num> |
| 0011 | External output | %EXTOUT |
| 0111 | ALU bus, hold CY | %ALU_HCY |
| 1010 | Base0 register WR | %WBASE0 |
| 1011 | Base1 register WR | %WBASE1 |
| 1100 | Base0 register RD | %RBASE0 |
| 1101 | Base1 register RD | %RBASE1 |
| 1110 | Base0 register RW | %RWBASE0 |
| 1111 | Base1 register RW | %RWBASE1 |
| 0100 | <undefined> | |
| 0101 | <undefined> | |
| 0110 | <undefined> | |
| 1000 | <undefined> | |
| 1001 | <undefined> | |

**Table 3.** *Bit coding and mnemonic names for destination operands.*

## 2.3 Address generator control

Note that in pre-add access mode, the base register adjust takes place immediately after the RAM access. Since the new base value equals the access address, this load value has already been computed which means that the load can take place at the clock edge between two accesses (i.e. it does not take any additional time).

| BITS | MODE | BASEREG | BASE NR. |
|---|---|---|---|
| 000 | ind | hold | 0 |
| 001 | ind | hold | 1 |
| 010 | ind | pre-add | 0 |
| 011 | ind | pre-add | 1 |
| 10x | abs | hold | - |
| 110 | abs | pre-add | 0 |
| 111 | abs | pre-add | 1 |

**Table 4.** *Bit coding for address generator control.*

## 2.4 Flow control

| BITS | MNEMONICS | | FUNCTION |
|---|---|---|---|
| 0000 | CONT | | pc := pc + 1 |
| 0001 | IDLE | | <reset registers, remain idle> |
| 0010 | JUMP | a | pc := pc + 1 + a |
| 0011 | JUMPC | a | **if** cy = 1 → pc := pc + 1 + a; ▯ cy = 0 → pc := pc + 1; **fi** |
| 0100 | JUMPNC | a | **if** cy = 1 → pc := pc + 1; ▯ cy = 0 → pc := pc + 1 + a; **fi** |
| 0101 | JUMPP | p | pc := PROC(p) |
| 0110 | RET | | pop(pc) |
| 0111 | RETC | | **if** cy = 1 → pop(pc); ▯ cy = 0 → pc := pc + 1; **fi** |
| 1000 | INITLP | v | pc := pc + 1; push(pc), ldcnt(v) |
| 1001 | TELP | | decrcnt; **if** cntzero → pop(), pc := pc + 1; ▯ ¬cntzero → pc := tos(); **fi** |
| 1010 | EXITLP | a | pop(), pc := pc + 1 + a |
| 1011 | EXITLPC | a | **if** cy = 1 → pop(), pc := pc + 1 + a; ▯ cy = 0 → pc := pc + 1; **fi** |
| 1100 | CALL | p | push(pc + 1), pc := PROC(p) |
| 1101 | CALLC | p | **if** cy = 1 → push(pc + 1), pc := PROC(p) ▯ cy = 0 → pc := pc + 1; **fi** |
| 1100 .. 1111 | | | <undefined> |

**Table 5.** *Bit coding, mnemonics and semantics for flow control functions.*

## 2.5 ALU functions and operands

| BITS | MNEMONICS | | OPERATION |
|---|---|---|---|
| 00000 | ADD | s1, s2, d | cy, d := s1 + s2 |
| 00001 | UNPK | s1, s2, d | d := **unpk**(s1, s2[w-1], s2[0..w/2 - 1], s2[w/2..w-2]) |
| 00010 | SUB | s1, s2, d | cy, d := s1 - s2 |
| 00011 | CPLC | | cy := $\neg$cy |
| 00100 | GT | s1, s2 | cy := s1 > s2 |
| 00101 | GTE | s1, s2 | cy := s1 $\geq$ s2 |
| 00110 | LT | s1, s2 | cy := s1 < s2 |
| 00111 | LTE | s1, s2 | cy := s1 $\leq$ s2 |
| 01000 | EQ | s1, s2 | cy := s1 = s2 |
| 01001 | NEQ | s1, s2 | cy := s1 $\neq$ s2 |
| 01010 | AFTER | s1, s2 | cy := s1 **after** s2 { == (s2 – s1)[w-1] } |
| 01011 | GETC | s1, s2 | cy := s2[s1] |
| 01100 | ANDC | s1, s2 | cy := cy $\wedge$ s2[s1] |
| 01101 | ORC | s1, s2 | cy := cy $\vee$ s2[s1] |
| 01110 | PUTNC | s, d | d := $\neg$cy << s |
| 01111 | PUTC | s, d | d := cy << s |
| 10000 | SETB | s1, s2, d | d := s2 **bitor** (1 << s1) |
| 10001 | CLRB | s1, s2, d | d := s2 **bitand** (**bitnot** (1 << s1)) |
| 10010 | MOVE | s, d | d := s |
| 10011 | MUL | s1, s2, d | d := s1 * s2 |
| 10100 | SHL | s1, s2, d | cy, d := s1 << s2 (undefined for s2 > w) |
| 10101 | SHR | s1, s2, d | cy, d := s1 >> s2 (undefined for s2 > w) |
| 10110 | SHLC | s, d | d := (s << 1) + cy, cy := s[w-1] |
| 10111 | SHRC | s, d | d := (s >> 1) + (cy << (w - 1)), cy := s[0] |
| 11000 | AND | s1, s2, d | d := s1 **bitand** s2 |
| 11001 | OR | s1, s2, d | d := s1 **bitor** s2 |
| 11010 | XOR | s1, s2, d | d := s1 **bitxor** s2 |
| 11011 | CHK | s1, s2 | **if** (s1<0) $\vee$ (s1 $\geq$ s2) $\rightarrow$ **error** ;<br>[] 0 $\leq$ s1 < s2 $\rightarrow$ **skip** ;<br>**fi** |
| 11100 | DIV | s1, s2, d | d := s1 **div** s2 |
| 11101 | MOD | s1, s2, d | d := s1 **mod** s2 |
| 11110 | XSGN | s1, s2, d | d[0..s2] := s1[0..s2]; d[s2+1], .. , d[w-1] := s1[s2] |
| 11111 | WRAP | s1, s2, d | **if** s1 $\geq$ s2 $\rightarrow$ d := s1 **mod** s2;<br>[] s1 < 0 $\rightarrow$ d := s2 - 1 - (**bitnot**(s1) **mod** s2);<br>[] 0 $\leq$ s1 < s2 $\rightarrow$ d := s1;<br>**fi** |

**Table 6.** *Bit coding, mnemonics and semantics for ALU operations.*

# 3 Microcode generation

Microcode generation for the pascal type expressions of the protocol grammar is quite difficult, because there are so many different types of attributes. Each of the 5 categories mentioned above requires another way to access it, but for attribute references (category 3, 4 and 5), the position and size of the attributes also has to be taken into account. An attribute can occupy a whole memory word (simple types), a partial memory word (structure fields) starting at any bit offset, or more than a whole word (structures and arrays). All of these cases require different strategies for accessing such a variable. In the following sections a method for handling these cases will be presented.

In all cases, the generation of code to evaluate a subexpression can be omitted if the subexpression is a constant or a simple reference to an attribute, which is not an array element. Instead, the subexpression can be compiled directly into a single operand of the parent expression. This is not a necessary step, but it will generate much better code (more optimized). In the sequel, this type of optimization will not be mentioned explicitly to keep the descriptions as simple as possible. However, in practice optimization steps are performed whenever appropriate.

## 3.1 Long attributes

These are variables that occupy more than one whole word in memory. This can only be the case for structures and arrays. There are only 4 types of expressions that can return or operate on a whole structure:

```
simple assignment: a := b
bit test function:  a := tstb(x, b)
bit set function:   a := setb(x, b)
bit clear function: a := clrb(x, b)
```

The latter two can be rewritten as a simple assignment (a := b), followed by evaluating the expression for x, computing the word address and bit number of bit x in a, and finally performing the requested operation. The bit test function can be evaluated similarly by evaluating b into a temporary variable space t (t := b) and performing the test operation on t (b can be a complex expression). Therefore, the only expression type of concern here is the simple assignment. Since structures always start at bit position 0 of a memory word, all words except the last can simply be copied from source to destination. Only the last word of the destination should be masked if some of its bits belong to other variables, which can be the case if and only if the destination is itself a field of a structure.

## 3.2 Short and packed attributes

Short attributes are attributes whose actual size (number of bits) is less than the number of bits in a memory word. This is the case for booleans, octets, ranges, etc. Since attributes always start at a word boundary in memory, short attributes will still occupy a whole word. Only fields of structures are tightly packed to fit as many fields together in each memory word as possible, with the exception that structures and all array elements always start at a word boundary, even if they are fields of a structure. A packed variable is a variable that actually occupies less than a whole word in memory, i.e. at least one of the bits of its memory location belongs to another variable.

A short attribute is stored in a location where some of the bits are undefined. This is inconvenient for accessing these attributes, because the unwanted bits have to be masked, before the value can be used for any operation (always word based). The only good solution is to specify that the unused bits in short but unpacked attributes always have a value, such that accessing the attribute as a word will also return the proper value. This can be guaranteed by a proper assignment coding algorithm, which simply sets unused bits to zero (for unsigned variables) or sign extends the store value to full word length before storing it (for signed variables).

A packed attribute must first be unpacked before it can be used. Unpacking an attribute consists of reading the value from the location where it is stored, then shifting this value right so that the attribute appears at bit position zero, and finally either setting all bits that do not belong to the attribute to zero (if it is unsigned) or sign extending the variable to the full word length (for signed variables).

Assigning a value to a packed attribute is even more difficult. The destination location must first be read, and all bits that do belong to the target attribute must be set to zero. Then the value to be stored must be shifted left to the corresponding bit position and masked so that it can not overwrite any bits not belonging to the target. Then the destination and new value are merged by ORing and finally the result is stored at the destination location.

Furthermore, a wrapping mechanism will make sure that every value assigned to an attribute will be within the value range specified for that attribute.

## 3.3 Word sized attributes

Word sized attributes are easy to access. They occupy exactly one whole memory location, which can be accessed either by absolute addressing or by register direct with displacement. No packing or unpacking is required. The internal registers and constants (category 1 and 2) operands do not require any additional processing since they are also one word wide and never packed.

## 3.4 Microcode generation for attribute expressions

To describe the mechanism used for generation of microcode, a few abstract data types and definitions have to be introduced:

### 3.4.1 Access mode of an operand

An *AccessMode* is a set type representing the method to be used for accessing an operand. It has the following elements:

AccessMode = { noAccess, immediate, internal, absolute, stdBase, altBase, deref }

which mean:

| | |
|---|---|
| noAccess: | inaccessible in attribute RAM in current situation |
| immediate: | operand is constant value, coded in instruction |
| internal: | operand is internal register, implied coding in instruction |
| absolute: | operand is attribute, use fixed address (coded in instruction) |
| stdBase: | operand is attribute, use standard base register plus offset |
| altBase: | operand is attribute, use alternate base register plus offset |
| deref: | operand is attribute, address can be found at location accessible from standard base plus offset. |

For accessing variables, it is necessary to have information about their position (offset in segment) and their size (number of bits) as well. The following functions return respectively the accessmode, the offset the needed bit size and the actual bit size of any attribute represented by a type VarID:

| | | |
|---|---|---|
| **AM**: | VarID → AccessMode | { method for access } |
| **VO**: | VarID → Integer | { whole var. offset in workspace in words } |
| **AO**: | VarID → Integer | { field offset from start of variable in bits } |
| **VNB**: | VarID → Integer | { needed no. of bits } |
| **TNB**: | VarID → Integer | { allocated no. of bits } |

### 3.4.2 Operand references

An *operand_ref* is an abstract data type representing any allowed operand in any microcode instruction. It has one of the following forms:

| | |
|---|---|
| $()_{nop}$ | representing the absence of an operand (empty) |
| $(v)_{con}$ | with $v \in$ Integer<br>representing a constant with value 'v' |
| $(nr)_{tv}$ | with $nr \in$ Integer<br>representing the temporary variable stored at offset 'nr' in the space available for storage of temporary variables. |
| $(rn)_{int}$ | with $rn \in$ Name<br>representing the internal register, whose name is 'rn' |
| $(vn, vo, s, v, t, am, vl)_{ext}$ | with: $vn \in$ VarID<br>$vo, s, v, t \in$ Integer<br>$am \in$ AccessMode<br>$vl \in$ Boolean<br>representing the (field of the) attribute whose name is 'vn', beginning at bit position 's' from the start of the whole attribute (which starts at word offset 'vo' in the workspace), actually needing 'v' bits, but occupying 't' bits in memory, accessible using mode 'am'. If 'vl' is true, then the operand_ref refers to the value of the variable, otherwise it refers to its location (for unpacking). |

The above functions and data types are related by the conversion function:

**varopr**: VarID $\rightarrow$ operand_ref

defined by:

$$\text{varopr}(vn) = (vn, \mathbf{VO}(vn), \mathbf{AO}(vn), \mathbf{VNB}(vn), \mathbf{TNB}(vn), \mathbf{AM}(vn), \text{true})_{ext}$$

### 3.4.3 Detection of packed operands

Next, a function named *packed* is introduced:

**packed**: operand_ref $\times$ accesstype $\rightarrow$ boolean

where: accesstype $\in$ { readAccess, writeAccess }

The function packed returns a boolean result indicating whether or not the variable it received as an argument is considered packed for the given type of access, i.e:

```
packed(op, at) =
        if op :: (. , . , s, v, t, . , vl)$_{ext}$→
          [    if at = readAccess →
                  vl $\wedge$ ((s mod syswidth $\neq$ 0) $\vee$ (t $\neq$ syswidth))
               [] at = writeAccess →
                  vl $\wedge$ ((s mod syswidth $\neq$ 0) $\vee$ (v $\neq$ syswidth))
               fi
          ]
        [] otherwise → false;
        fi
```

**Note:** The system constant *syswidth* is defined to be equal to the width of the system data path (memory, alu, buses, etc.).

To refer to the *value* of an unpacked word sized operand, a function named *value* is defined that will return this value.

**value**: operand_ref → Integer

Some of the functions to be defined next will generate output in the form of microcode. This output can be either text or binary code. In either case, it will be referred to as microcode, and a procedure *execute* is introduced that will accept any microcode and execute it.

microcode = 'a sequence of ascii characters' or
'a sequence of binary codes'
representing an executable program for the PPDA.

**execute**: microcode → { }

The unpack procedure is defined as follows: given a source operand_ref (src) and a destination operand_ref (dst), it generates microcode to unpack 'src' into 'dst'.

**unpack**: operand_ref × operand_ref → microcode

unpack(src, dst):

{ src :: $(vn, vo, s, v, t, am, true)_{ext}$ }
**execute**(**unpack**(src, dst))
{ ¬**packed**(dst) ∧ **value**(dst) = **value**(src) }

The implementation of unpack will be described later.

### 3.4.4 Low level instruction generator

Some other data types are defined for the description of the lowest level instruction generation process. The first data type is called *InstrOpr* and it has one of the following formats:

| | |
|---|---|
| $(am, offs)_{ram}$: | am ∈ { stdb, altb, stdbpincr, altbpincr, abs0, abs1, abs0pincr, abs1pincr }, is the access mode to generate. |
| | offs ∈ Integer, is the fixed address or offset for the access. |
| $(nr, offs)_{tmpv}$: | nr ∈ Integer, is the offset for the temporary variable. |
| | offs ∈ Integer, is the offset for access to the temp. var. |
| $(rn)_{intl}$: | rn ∈ Name, is the internal register whose name is rn. |
| $(d)_{imme}$: | d ∈ Integer, is a constant with value d. |
| $()_{noam}$: | an empty (missing) operand |

The second new data type is called *FlowCtl* and has only one format:

| | |
|---|---|
| $(fc, fl)_{flw}$: | fc ∈ { cont, idle, jump, jumpc, jumpnc, jumpp, ret, retc, initlp, telp, exitlp, exitlpc, call, callc }, is the flow control function. |
| | fl ∈ Name, is a label representing flow data (only required for some flow control functions). |

The third new data type is simply the collection of all possible alu functions as defined in table 6. It is called *ALUFunc*.

Now, a function is defined that will return (generate) a complete instruction (mnemonic and/or binary code), when given an alu function, source and destination operands and flow control information as arguments.

**GenInstr**: ALUFunc ×
InstrOpr × InstrOpr × InstrOpr ×
FlowCtl
→ microcode

### 3.4.5 Operand class

A very important piece of information for any operand_ref is its access mode. Since the access mode of an operand is referred to very often, a function is defined that will return the access mode of any operand. It will be called *class.*

**class**: operand_ref $\rightarrow$ AccessMode

It is defined as follows:

| | | |
|---|---|---|
| $\text{class}(()_{nop})$ | = | noAccess |
| $\text{class}((v)_{con})$ | = | immediate |
| $\text{class}((nr)_{tv})$ | = | stdBase |
| $\text{class}((rn)_{int})$ | = | internal |
| $\text{class}((.\,,\,.\,,\,.\,,\,.\,,\,.\,,\,am,\,.)_{ext})$ | = | am |

### 3.4.6 Operand locations and read/write accessing

Next, a few more code generation aid functions will be introduced. The function *OO* will return the location of a variable either as an offset (in the current workspace) or as an absolute address (in the global segment).

**OO**: operand_ref$\rightarrow$Integer

It is defined as follows:

```
OO(()_nop)      = 0 ;
OO((v)_con)     = 0 ;
OO((nr)_tv)     = nr + TempSpaceOffset ;
OO((rn)_int)    = 0 ;
OO((. , vo, s, . , . , am, .)_ext) =
        if am = absolute   →   vo + (s div syswidth) ;
        [] am = stdBase    →   vo + (s div syswidth) ;
        [] am = altBase    →   s div syswidth ;
        [] am = deref      →   vo ;
        fi
```

where TempSpaceOffset is the offset for the first location in the current workspace that is available for storage of temporary variables.

The function *GV* will convert an operand_ref for a word sized (unpacked) operand to an *InstrOpr* data type for use by the *GenInstr* function.

**GV**: operand_ref$\rightarrow$InstrOpr

*GV* does not accept packed attributes or attributes that occupy more than one word in memory, because it simply calculates the address or offset of the memory word where the first bit of the operand is stored.

This is how the mapping function GV is defined:

```
$GV(()_{nop})$        $= ()_{noam}$ ;
$GV((v)_{con})$       $= (v)_{imme}$ ;
$GV((nr)_{tv})$       $= (nr, \mathbf{OO}((nr)_{tv}))_{tmpv}$ ;
$GV((rn)_{int})$      $= (rn)_{intl}$ ;
$GV((. , vo, s, . , . , am, .)_{ext}) =$
        if $am = absolute \rightarrow (abs0, \mathbf{OO}((. , vo, s, . , . , am, .)_{ext}))_{ram}$ ;
        $\square$ $am \in \{ stdBase, deref \} \rightarrow (stdb, \mathbf{OO}((. , vo, s, . , . , am, .)_{ext}))_{ram}$ ;
        $\square$ $am = altBase \rightarrow (altb, \mathbf{OO}((. , . , s, . , . , am, .)_{ext}))_{ram}$ ;
        fi
```

To access the destination operand a similar function is used, called PV. The function PV can only access unpacked word sized destination operands. It is identical to GV except for constants, which can never be the destination of an operation.

**PV**: operand_ref$\rightarrow$ InstrOpr

This is how the mapping function PV is defined:

```
PV(op )=      if op :: $(.)_{con}$   $\rightarrow$      $()_{noam}$ ;
              $\square$ otherwise     $\rightarrow$      GV(op) ;
              fi
```

Next, the third code generation function is defined. It is called *ShallowCompile* and functions as an interface between higher level compiler functions and the low level *GenInstr* function. *ShallowCompile* converts an ALU function with operands and flow control into microcode.

**ShallowCompile**: ALUFunc × operand_ref × operand_ref × operand_ref × FlowCtl $\rightarrow$ microcode

ShallowCompile ( opr, s1, s2, d, fflow) =
**GenInstr**(opr, **GV**(s1), **GV**(s2), **PV**(d), fflow );

### 3.4.7 Setting up base registers for the address generators

The alternate read and write base registers are used for every read or write access to passed attributes and array elements, for multi word attribute copying (structures) and for input and output data transfers to and from the external data path connected to the Attribute Evaluator. Therefore they are used quite often and generating code for setting those registers up for an access has been implemented in a specialized function called *SetupAltBase*. To define it, yet another data type is introduced:

| | | |
|---|---|---|
| basetype | = | { rdb, wrb, rwb } is a set of values representing alternate read, alternate write or both base registers. |

And the function definition is:

**SetupAltBase**: operand_ref × operand_ref × basetype $\rightarrow$ operand_ref × microcode

*SetupAltBase* generates code to compute the target address (where the first operand can be found in attribute RAM), to add that address to the second operand and to store the result in the requested alternate base register(s). In most cases, this will take just one microcode instruction, but in some cases it takes two. Furthermore, it will return an operand_ref which is identical to the first argument but with an access mode that has been changed to reflect the change in the alternate base register (i.e. the operand is now accessible using an offset to the alternate base). The function result of SetupAltBase is a pair whose second element, denoted by $(.\,,\,.)[\![2]\!]$ is microcode, and whose first element $(.\,,\,.)[\![1]\!]$ is an operand_ref.

**execute**(**SetupAltBase**(x, d, bt)$[\![2]\!]$) has the following effect:

**if** x :: $(.)_{con}$ ∨ x :: $(.)_{int}$ $\rightarrow$ base1 := base0 + **value**(x) + **value**(d);
▯ x :: $(n)_{tv}$ $\rightarrow$ base1 := base0 + **OO**(x) + **value**(d);

```
▯ x :: (. , . , . , . , . , . , .)_ext →
        if class(x) = absolute → base1 := OO(x) + value(d);
        ▯ class(x) = stdBase → base1 := base0 + OO(x) + value(d);
        ▯ class(x) = deref → base1 := MEM(OO(x) + base0) + value(d);
        ▯ otherwise → skip
        fi
fi
```

where MEM(a) returns the value stored at RAM location 'a'.

The first function result element, the new operand_ref is defined as follows:
SetupAltBase(x, . , .)⟦1⟧ =

```
        if x :: (.)_con ∨ x :: (.)_int ∨ x :: (.)_tv → x ;
        ▯ x :: (vn, vo, s, v, t, am, vl)_ext →
                if am = absolute  →  (vn, vo, 0, v, t, altBase, vl)_ext ;
                ▯ am = stdBase    →  (vn, vo, 0, v, t, altBase, vl)_ext ;
                ▯ am = deref      →  (vn, vo, s, v, t, altBase, vl)_ext ;
                ▯ otherwise       →  skip
                fi
        fi
```

Here, base1 is the value in the alternate read, write or both register(s), depending on the basetype argument. Similarly, base0 represents the current value of the standard base registers (these are always equal to each other).

Actually, the value of the base registers can not be read back from them. Therefore, the value for base0 must be stored in a so called shadow location in RAM. This is done in the system segment, which must be accessed using absolute addressing mode.

### 3.4.8 Intermediate level instruction generation

Now the microcode compilation process can be defined in terms of the functions and procedures described above. The following list shows all possible combinations of source operand types, and the resulting microcode. Note that this list is for both operations requiring zero, one or two operands (if either source or destination operand is not required, the corresponding *InstrOpr* argument will be $()_{nop}$ and the access class value will be 0.

Consider a function *Compile* that will map an ALU function with at most 2 source operands and 1 unpacked word sized destination operand to microcode. The function *Compile* can be defined recursively. Operations with packed attribute operands (classes absolute, stdBase, altBase and deref) can be compiled by performing some preprocessing, and then calling *Compile* again with simpler arguments. A final argument to this function gives the flow control function that should be generated at the very last instruction generated by this call to *Compile*.

```
    Compile: ALU function ×
             operand_ref × operand_ref × operand_ref ×
             FlowCtl
             →    microcode
```

Compile (opr, s1, s2, d, ff) will now be defined. The output varies according to the operand access modes, but from a higher level the following mode sets must be distinguished (and require different preprocessing):

|  |  |
|---|---|
| set 1: | { noAccess, immediate, internal } |
| set 2: | { absolute, stdBase, altBase } |
| set 3: | { deref } |

There are 9 set combinations, and for 7 of those a different strategy for code output is necessary.

**A)** **class**(s1) $\in$ set1 $\wedge$ **class**(s2) $\in$ set1

***Note***: the subcase class(s1) = immediate $\wedge$ class(s2) = immediate should (and will) not occur, since:

a) hardware does not support it and
b) the high level expression compiler will precompute constant expressions.

**compile** (opr, s1, s2, d, ff) = **shallowcompile**(opr, s1, s2, d, ff);

**B)** **class**(s1) $\in$ set2 $\wedge$ **class**(s2) $\in$ set1

```
compile (opr, s1, s2, d, ff) =
    if packed(s1, readAccess) →
            if (opr = MOVE) $\wedge$ (class(d) $\notin$ { deref, altBase}) → unpack(s1, d);
            [] (opr $\neq$ MOVE) $\vee$ (class(d) $\in$ { deref, altBase}) →
```

```
                ⟦ unpack(s1, $(ALU)_{int}$);
                  compile(opr, $(ALU)_{int}$, s2, d, ff);
                ⟧
          fi
    ▯ ¬packed(s1, readAccess) → shallowcompile(opr, s1, s2, d, ff);
    fi
```

C) **class**(s1) ∈ set3 ∧ **class**(s2) ∈ set1 ∪ set2

```
compile (opr, s1, s2, d, ff) =
    ⟦ s1 := setupaltbase(s1, $()_{nop}$, rdb)⟦1⟧ ;
      compile(opr, s1, s2, d, ff);
    ⟧
```

D) **class**(s1) ∈ set1 ∧ **class**(s2) ∈ set2

This case is identical to **B** when s1 is swapped with s2.

```
compile (opr, s1, s2, d, ff) =
    if packed(s2, readAccess) →
          if (opr = MOVE) ∧ (class(d) ∉ { deref, altBase} )→ unpack(s2, d);
          ▯ (opr ≠ MOVE) ∨ (class(d) ∈ {deref, altBase} )→
                ⟦ unpack(s2, $(ALU)_{int}$);
                  compile(opr, s1, $(ALU)_{int}$, d, ff);
                ⟧
          fi
    ▯ ¬packed(s2, readAccess) → shallowcompile(opr, s1, s2, d, ff);
    fi
```

E) **class**(s1) ∈ set2 ∧ **class**(s2) ∈ set2

```
compile (opr, s1, s2, d, ff) =
    if ¬packed(s1, readAccess) ∧ ¬packed(s2, readAccess)→
          ⟦ compile(MOVE, s1, $()_{nop}$, $(ALU)_{int}$, $(cont, ' ')_{flw}$);
            compile(opr, $(ALU)_{int}$, s2, d, ff);
          ⟧
    ▯ packed(s1, readAccess) ∧ ¬packed(s2, readAccess)→
          ⟦ unpack(s1, $(ALU)_{int}$);
            compile(opr, $(ALU)_{int}$, s2, d, ff);
            ]|
    [] ¬packed(s1, readAccess) ∧ packed(s2, readAccess)→
            |[ unpack(s2, $(ALU)_{int}$);
               compile(opr, s1, $(ALU)_{int}$, d, ff);
            ]|
    [] packed(s1, readAccess) ∧ packed(s2, readAccess)→
            |[ unpack(s1, $(TEMPREG)_{int}$);
               unpack(s2, $(ALU)_{int}$);
               compile(opr, $(TEMPREG)_{int}$, $(ALU)_{int}$, d, ff);
            ]|
    fi
```

**F)** **class**(s1) $\in$ set1 $\cup$ set2 $\wedge$ **class**(s2) $\in$ set3

This case is identical to **C** when s1 is swapped with s2.

```
compile (opr, s1, s2, d, ff) =
    |[ s2 := setupaltbase(s2, $()_{nop}$, rdb)[[1]] ;
       compile(opr, s1, s2, d, ff);
    ]|
```

**G)** **class**(s1) $\in$ set3 $\wedge$ **class**(s2) $\in$ set3

```
compile (opr, s1, s2, d, ff) =
    if ¬packed(s1, readAccess)→
            |[ s1 := setupaltbase(s1, $()_{nop}$, rdb)[[1]] ;
               compile(MOVE, s1, $()_{nop}$, $(TEMPREG)_{int}$, $(cont, ‘ ’)_{flw}$);
               compile(opr, $(TEMPREG)_{int}$, s2, d, ff);
            ]|
    [] packed(s1, readAccess)→
            |[ s1 := setupaltbase(s1, $()_{nop}$, rdb)[[1]] ;
               unpack(s1, $(TEMPREG)_{int}$);
               compile(opr, $(TEMPREG)_{int}$, s2, d, ff);
            ]|
    fi
```

This concludes a full list of all possible combinations of packed and unpacked operands of all classes. The code generating function *unpack* can now be defined in terms of *compile*. The expansion of a 'v' bit variable into a full word length is done using a function called *expand*, which will also be defined in terms of *compile*.

### 3.4.9 Unpacking operands

For unsigned sources, expand uses a masking function to force all bits of a word that do not belong to the target attribute to zero, and for signed sources, it sign extends the source to a full word length.

```
mask:  Integer × Integer → Integer
       mask(a, b) ≡ ((1 << b) - 1) << a{ bits a...a+b-1 are logic 1 }

signed: VarID → Boolean
       signed(vn) ⇔ 'vn is a signed variable'

expand: operand_ref × operand_ref × Integer × Boolean → microcode

expand(s, d, nrbits, s_signed) =
if s_signed      → compile(XSGN, s, $(\text{nrbits} - 1)_{con}$, d, $(\text{cont}, \text{' '})_{flw}$);
[] ¬ s_signed    → compile(AND, s, mask(0, nrbits), d, $(\text{cont}, \text{' '})_{flw}$);
fi
```

```
Unpack($(\text{vn}, \text{vo}, \text{s}, \text{v}, \text{t}, \text{a}, .)_{ext}$, d) =
    if s mod syswidth ≠ 0 →
        [[ h: Integer |
           h := s mod syswidth + ((v - 1) << syswidth div 2) ;
           if signed(vn)   →   h := h + (1 << syswidth - 1) ;
           [] ¬signed(vn)  →   skip
           fi ;
           compile(UNPK, $(\text{vn}, \text{vo}, \text{s}, \text{v}, \text{t}, \text{a}, \text{false})_{ext}$, $(\text{h})_{con}$, d, $(\text{cont}, \text{' '})_{flw}$);
        ]]
    [] s mod syswidth = 0 →
        [[ expand($(\text{vn}, \text{vo}, \text{s}, \text{v}, \text{t}, \text{a}, \text{false})_{ext}$, d, v, signed(vn))
        ]]
    fi
```

As can be seen from this code, unpack only uses the internal ALU bus for its operation and therefore does not change the value of the internal temp_reg register, nor does it use any temporary variables. The special opcode UNPK has the effect as described in table 6 as a function **unpk** with a number of arguments:

```
unpk: Integer × Boolean × Integer × Integer → microcode

unpk(val, signed, firstbit, highbit) =
        if signed      →     xsgn(val >> firstbit, highbit)
        [] ¬signed     →     bitand(val >> firstbit, mask(0, highbit + 1))
        fi
```

where '>>' represents a shift right operation and **xsgn** is the sign extension operation as defined by the corresponding ALU function.

### 3.4.10 Class dependent destination operand handling

As stated at the introduction of the compile function, it can only handle unpacked (word sized) destination operands, because the code it generates simply writes the result to the destination as a whole word. Therefore, some preprocessing is required if the destination operand is anything else but unpacked and word sized. The destination operand can never be a multiple word sized operand because those types of expressions are handled separately, as explained before. The only cases in which preprocessing is required are therefore those where the destination is packed or where its access mode is deref (or both). Note that internal registers and temporary variables are never packed.

In case the destination's access mode is *deref*, the only preprocessing required is to set up the alternate base register for write access to the start location of the destination operand. The call to *GenInstr* in the deepest level of compile will then automatically compile a write access relative to the alternate write base register with offset 0.

In case the destination operand is packed but does not have access mode deref, it is first changed into the operand $(\text{TEMPREG})_{int}$, which means that the result of the operation will be left in that internal register. From there, it is packed into the real destination by a special function called pack. This function will be defined below.

Finally, in case the destination is a packed operand with access mode deref, the same actions are taken as in the previous case, but just before calling *pack* the alternate write and read base registers are set up to the start of the destination operand. Packing requires both a read and a write access to the destination, which is why both alternate registers must be set up.

### 3.4.11 Packing operands

The pack procedure is defined as follows. It takes a source operand_ref (src) and a starting bit number in src, a destination operand_ref and (since it is often the last step for an expression) final flow control data, and generates microcode for the packing of 'src' into 'dst'.

**pack**: operand_ref $\times$ Integer $\times$ operand_ref $\times$ FlowCtl $\rightarrow$ microcode

pack(src, fb, dst, ff):

$\{ \neg$ packed(src) $\}$
**execute**(**pack**(src, fb, dst, ff))
$\{$ packed(dst) $\wedge$ value(dst) = value(src) $\}$

The exact operation of pack depends on the types of the variables it operates on. In general, unpacking and packing variables is an expensive operation, which is why it must be performed as little as possible. Packing is only necessary for assigning values, whereas unpacking is necessary for every read access to a short variable. Although packed variables must always be unpacked, there is a way to dramatically reduce the necessity of unpacking the short variables (those variables that occupy a whole word in memory, but actually use less). If it is guaranteed that accessing short variables as words will return the same value as accessing through unpack, the unpack is no longer necessary for these cases. This guarantee can in fact be made, if for every assignment to such a variable, the value to be stored is expanded first. The pack procedure will do this for short variables.

### 3.4.12 Automatic assignment value wrapping

It was also stated that it would be impossible to assign values to an attribute that are outside the specified range for that attribute. For example, assigning -4 to a range variable <-2..+6> would actually assign +5 (wrapping). This is generalization of the well known modulo N arithmetic. In this case, N is the cardinality of the range (the number of integer values within the range), and the base is not 0, but the lowest value in the range. The ALU has a function called WRAP that can force a value to be in the range <0..K-1> when given a source operand, K and a destination operand. Pack will use this function to enforce the more general wrapping technique for the appropriate attributes.

For assignments to packed variables, the pack procedure will also generate code to shift the source value to the correct bit position, and to replace the corresponding bits in the destination by the new source value.

### 3.4.13 Top level instruction generation

Now, the top level microcode generation function can be defined, that handles all cases, independent of operand categories and types. The function *Generate* takes an ALU function, 2 source operands, one destination operand and flow control for the last microcode instruction to be generated as arguments and generates microcode to perform the given operation.

```
Generate: ALU function ×
          operand_ref × operand_ref × operand_ref ×
          FlowCtl
          →   microcode
```

If the destination is packed for write access then the **compile** function is called with destination TEMPREG. Next, if the destination is an array element or has access mode deref, then the alternate read and write base registers are setup to point to the destination variable. Finally **pack** is called to pack the value now in TEMPREG into the real destination.

If the destination is not packed for write access then the alternate write base register is setup if (and only if) the destination is an array element or has access mode deref, after which the **compile** function is called directly with unchanged arguments.

### 3.4.14 High level expression compilation

Compiling the high level expressions into microcode now simply becomes a matter of a treewalk, while calling the *Generate* function at every level. At the same time, intermediate results have to be stored somewhere. The internal registers are used by compile and can not be used for temporary result storage on this level. The number of intermediate values to store depends on the width of the expression tree, and can therefore have almost any value. That is why these results will be stored in attribute RAM at positions above the current work space. These locations are called TMPV's (TeMPorary Var's). Their number and position are computed at compile time and hardcoded into the expressions. Keeping track of these TMPV's is another task to be performed during treewalk. The last function to be performed during this phase is evaluation of constant expressions.

# Bibliography

| | |
|---|---|
| Aho72 | Aho, A.V. and J.D. Ullman<br>The theory of parsing, translation, and compiling. Vol. 1: Parsing.<br>Englewood Cliffs: Prentice Hall, 1972-1973. |
| Aho86 | Aho, A.V. et al.<br>Compilers: Principles, techniques and tools.<br>Amsterdam: Addison Wesley, 1986. |
| Anderson85a | Anderson, D.P. and L.H. Landweber<br>Protocol specification by real-time attribute grammars.<br>In: Protocol specification, testing and verification. Proc. of the IFIPWG6.1 4th int. conf., Skytop Lodge, Pennsylvania, USA, June 11-14, 1985. Ed. by Y. Yemini et al.<br>Amsterdam: North-Holland, 1985. P. 457-465. |
| Anderson85b | Anderson, D.P. and L.H. Landweber<br>A grammar-based methodology for protocol specification and implementation.<br>In: Data communications. Proc. of the 9th symp., Whistler Mountain, British Columbia, Sept. 10-13, 1985.<br>Washington DC: IEEE Computer Society, 1985. P. 63-70. |
| Bloks91 | Bloks, R.H.J.<br>A protocol engine architecture.<br>In: Computer systems. Proc. of the 4th workshop, Amsterdam, Okt. 11, 1991. Ed. by P. Hartel and H. Muller.<br>Amsterdam: Computer Systems Dept., Univ. of Amsterdam, 1991. P. 43-62. |
| Bloks92 | Bloks, R.H.J.<br>A metagrammar for ProGrIL.<br>Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, November 1992.<br>Internal Report no. PRO/EB/9202. |
| Bloks93a | Bloks, R.H.J.<br>ProGrIL: A language for the definition of protocol grammars.<br>Eindhoven: Faculty of Electrical Engineering, Eindhoven University of Technology, 1993. EUT Report 93-E-270. |
| Bloks93b | Bloks, R.H.J.<br>A grammar based approach towards the automatic implementation of data communication protocols in hardware.<br>Dissertation, Eindhoven University of Technology.<br>To be published approx. May 1993. |
| Denning78 | Denning, P.J. et al.<br>Machines, languages and computation.<br>Englewood Cliffs: Prentice-Hall, 1978. |
| Fisher88 | Fisher, C.N. and R.J. Leblanc<br>Crafting a compiler.<br>Amsterdam: Benjamin/Cummings, 1988. |
| Haas85 | Haas, O.<br>Spezifikation von Kommunikationsprotokollen auf der Basis attributierter Grammatiken. (in German).<br>Dissertation, Technische Universität München, 1985. |

| | |
|---|---|
| Haas86 | Haas, O.<br>Formal protocol specification based on attribute grammars.<br>In: Protocol specification, testing and verification. Proc. of the IFIP WG 6.1 5th int. conf., Toulouse-Moissac, France, June 10-13, 1985. Ed. by M. Diaz.<br>Amsterdam: Elsevier,1986. P. 39-48. |
| Kain72 | Kain, R.Y.<br>Automata theory: Machines and languages.<br>London: McGraw-Hill, 1972. |
| Knuth68 | Knuth, D.E.<br>Semantics of context-free languages.<br>Mathematical Systems Theory, vol. 2 (1968), no. 2, p. 127-145. |
| Lewis81 | Lewis, H.R. and C.H. Papadimitriou<br>Elements of the theory of computation.<br>Englewood Cliffs: Prentice-Hall, 1981. |
| Linn83 | Linn, R.J. and W.H. McCoy<br>Producing tests for implementations of OSI protocols.<br>In: Protocol specification, testing and verification. Proc. of the IFIP WG 6.1 3rd int. conf., Rüschlikon, Switzerland, May 31-June 2, 1983. Ed. by H. Rudin and C.H. West.<br>Amsterdam: North-Holland, 1983. P. 505-520. |
| Lunteren91 | Lunteren, J. van<br>Ontwerp van een attribuut evaluator als onderdeel van een grammatica processor.<br>Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, 1991. Student report (in Dutch). |
| Sunshine82 | Sunshine, C.A.<br>Formal modeling of communication protocols.<br>In: Computer networks and simulation II. Ed. by S. Schoemaker.<br>Amsterdam: North-Holland, 1982. |
| Ural84 | Ural, H. and R.L. Probert<br>Automated testing of protocol specifications and their implementations.<br>In: Communications architectures & protocols. Proc. ACM Sigcomm '84 symp., June 6-8, 1984, Montreal, Quebec, Canada 1984 in Computer Communications Review, vol. 14 (1984), no. 2, p.149-155.<br>New York: ACM, 1984. |

Eindhoven University of Technology Research Reports ISSN 0167-9708
Coden: TEUEDE

Faculty of Electrical Engineering

(247) Józwiak, L. and T. Spassova-Kwaaitaal
DECOMPOSITIONAL STATE ASSIGNMENT WITH REUSE OF STANDARD DESIGNS: Using counters as sub-machines and using the method of maximal adjacensies to select the state chains and the state codes.
EUT Report 90-E-247. 1990. ISBN 90-6144-247-8

(248) Hoeijmakers, M.J. and J.M. Vleeshouwers
DERIVATION AND VERIFICATION OF A MODEL OF THE SYNCHRONOUS MACHINE WITH RECTIFIER WITH TWO DAMPER WINDINGS ON THE DIRECT AXIS.
EUT Report 90-E-248. 1990. ISBN 90-6144-248-6

(249) Zhu, Y.C. and A.C.P.M. Backx, P. Eykhoff
MULTIVARIABLE PROCESS IDENTIFICATION FOR ROBUST CONTROL.
EUT Report 91-E-249. 1991. ISBN 90-6144-249-4

(250) Pfaffenhöfer, F.M. and P.J.M. Cluitmans, H.M. Kuipers
EMDABS: Design and formal specification of a datamodel for a clinical research database system.
EUT Report 91-E-250. 1991. ISBN 90-6144-250-8

(251) Eijndhoven, J.T.J. van and G.G. de Jong, L. Stok
THE ASCIS DATA FLOW GRAPH: Semantics and textual format.
EUT Report 91-E-251. 1991. ISBN 90-6144-251-6

(252) Chen, J. and P.J.I. de Maagt, M.H.A.J. Herben
WIDE-ANGLE RADIATION PATTERN CALCULATION OF PARABOLOIDAL REFLECTOR ANTENNAS: A comparative study.
EUT Report 91-E-252. 1991. ISBN 90-6144-252-4

(253) Haan, S.W.H. de
A PWM CURRENT-SOURCE INVERTER FOR INTERCONNECTION BETWEEN A PHOTOVOLTAIC ARRAY AND THE UTILITY LINE.
EUT Report 91-E-253. 1991. ISBN 90-6144-253-2

(254) Velde, M. van de and P.J.M. Cluitmans
EEG ANALYSIS FOR MONITORING OF ANESTHETIC DEPTH.
EUT Report 91-E-254. 1991. ISBN 90-6144-254-0

(255) Smolders, A.B.
AN EFFICIENT METHOD FOR ANALYZING MICROSTRIP ANTENNAS WITH A DIELECTRIC COVER USING A SPECTRAL DOMAIN MOMENT METHOD.
EUT Report 91-E-255. 1991. ISBN 90-6144-255-9

(256) Backx, A.C.P.M. and A.A.H. Damen
IDENTIFICATION FOR THE CONTROL OF MIMO INDUSTRIAL PROCESSES.
EUT Report 91-E-256. 1991. ISBN 90-6144-256-7

(257) Maagt, P.J.I. de and H.G. ter Morsche, J.L.M. van den Broek
A SPATIAL RECONSTRUCTION TECHNIQUE APPLICABLE TO MICROWAVE RADIOMETRY.
EUT Report 92-E-257. 1992. ISBN 90-6144-257-5

(258) Vleeshouwers, J.M.
DERIVATION OF A MODEL OF THE EXCITER OF A BRUSHLESS SYNCHRONOUS MACHINE.
EUT Report 92-E-258. 1992. ISBN 90-6144-258-3

(259) Orlov, V.B.
DEFECT MOTION AS THE ORIGIN OF THE 1/F CONDUCTANCE NOISE IN SOLIDS.
EUT Report 92-E-259. 1992. ISBN 90-6144-259-1

(260) Rooijackers, J.E.
ALGORITHMS FOR SPEECH CODING SYSTEMS BASED ON LINEAR PREDICTION.
EUT Report 92-E-260. 1992. ISBN 90-6144-260-5

(261) Boom, T.J.J. van den and A.A.H. Damen, Martin Klompstra
IDENTIFICATION FOR ROBUST CONTROL USING AN H-infinity NORM.
EUT Report 92-E-261. 1992. ISBN 90-6144-261-3

(262) Groten, M. and W. van Etten
LASER LINEWIDTH MEASUREMENT IN THE PRESENCE OF RIN AND USING THE RECIRCULATING SELF HETERODYNE METHOD.
EUT Report 92-E-262. 1992. ISBN 90-6144-262-1

(263) Smolders, A.B.
RIGOROUS ANALYSIS OF THICK MICROSTRIP ANTENNAS AND WIRE ANTENNAS EMBEDDED IN A SUBSTRATE.
EUT Report 92-E-263. 1992. ISBN 90-6144-263-X

(264) Freriks, L.W. and P.J.M. Cluitmans, M.J. van Gils
THE ADAPTIVE RESONANCE THEORY NETWORK: (Clustering-) behaviour in relation with brainstem auditory evoked potential patterns.
EUT Report 92-E-264. 1992. ISBN 90-6144-264-8

(265) Wellen, J.S. and F. Karouta, M.F.C. Schemmann, E. Smalbrugge, L.M.F. Kaufmann
MANUFACTURING AND CHARACTERIZATION OF GAAS/ALGAAS MULTIPLE QUANTUMWELL RIDGE WAVEGUIDE LASERS.
EUT Report 92-E-265. 1992. ISBN 90-6144-265-6

(266) Cluitmans, L.J.M.
USING GENETIC ALGORITHMS FOR SCHEDULING DATA FLOW GRAPHS.
EUT Report 92-E-266. 1992 ISBN 90-6144-266-4

(267) Józwiak, L. and A.P.H. van Dijk
A METHOD FOR GENERAL SIMULTANEOUS FULL DECOMPOSITION OF SEQUENTIAL MACHINES: Algorithms and implementation.
EUT Report 92-E-267. 1992. ISBN 90-6144-267-2

(268) Boom, H. van den and W. van Etten, W.H.C. de Krom, P. van Bennekom, F. Huijskens, L. Niessen, F. de Leijer
AN OPTICAL ASK AND FSK PHASE DIVERSITY TRANSMISSION SYSTEM.
EUT Report 92-E-268. 1992. ISBN 90-6144-268-0

(269) Putten, P.H.A. van der
MULTIDISCIPLINAIR SPECIFICEREN EN ONTWERPEN VAN MICROELEKTRONICA IN PRODUKTEN (in Dutch).
EUT Report 93-E-269. 1993. ISBN 90-6144-269-9

(270) Bloks, R.H.J.
PROGRIL: A language for the definition of protocol grammars.
EUT Report 93-E-270. 1993. ISBN 90-6144-270-2

(271) Bloks, R.H.J.
CODE GENERATION FOR THE ATTRIBUTE EVALUATOR OF THE PROTOCOL ENGINE GRAMMAR PROCESSOR UNIT.
EUT Report 93-E-271. 1993. ISBN 90-6144-271-0