AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 187/81 DECEMBER

H.J. SINT

MIDL-A MICROINSTRUCTION DESCRIPTION LANGUAGE

Preprint

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

1980 Mathematics subject classification: 68AØ5

ACM-Computing Reviews-category: 6.20

MIDL - A Microinstruction Description Language*)

by

Marleen Sint

ABSTRACT

A microinstruction description language called MIDL is introduced. A MIDL description of a microarchitecture defines the semantics and triggering conditions of all microoperations. It also defines operand selection. MIDL incorporates a timing model that allows detailed specification of the timing of each microoperation, and a sequencing model that allows the description of many different sequencing schemes.

KEY WORDS & PHRASES: MICROARCHITECTURES, HARDWARE DESCRIPTION LANGUAGES

*) This paper is not for review; it has been submitted for publication elsewhere.

1. BACKGROUND AND MOTIVATION

User microprogramming is not really widespread. The microarchitectures of user microprogrammable machines tend to be complicated; as a result, the speed gain to be achieved (often claimed to be a factor of ten*) is seldom worth the trouble. If microprograms could be written in a (preferably machine independent) high level language instead of a microassembly language, microprogramming would become considerably easier. Several such languages have been designed and even experimentally implemented, but to date none of them has been generally accepted. I have argued elsewhere [Sin80] that this is not primarily caused by the absence of suitable languages, but by the lack of adequate implementation techniques. The parallelism and the inhomogeneity that are characteristic of microarchitectures make it difficult to produce even moderately efficient microcode.

A general code generation technique has to assume some structure common to all machines for which it can produce code. It needs a detailed model of each individual machine for which it is used. Therefore, some formalism to specify such a model has to be provided. A formalism general enough to model a microarchitecture at an abstraction level suitable for code generation does not yet exist. The evidence for this conclusion will be presented in section 2.

This observation has led to the initiation of a research project aiming at the development of such a formalism. A machine independent microcode generation system will be developed to test its suitability. Initially, requirements on the efficiency of the generated code will be modest. The (conveniently simple) language YALLL [Tuc79, Pat79] will be used as the source language in this project.

The code generation system will be modeled after the one Cattell has developed for conventional machines [Cat77, Cat80]. The structure of this system is shown in figure 1.

In Cattell´s method, it is essential that semantics of machine operations are defined by statements from the source language. For my project, this implies that the microarchitecture description, labeled (1) in the diagram, must define the effect of microoperations on the machine state in terms of YALLL statements. The task of the table-generator (2) is to reverse this definition: its output is a table (3) that maps each YALLL construct into a sequence of microoperations. This table in turn can be used to drive a code generator (4), which translates a YALLL program into microcode for the machine described by (1).

*) This claim seems open to doubt, however. Some organizational features, like the availability of an instruction or a data cache, are likely to reduce this factor seriously.

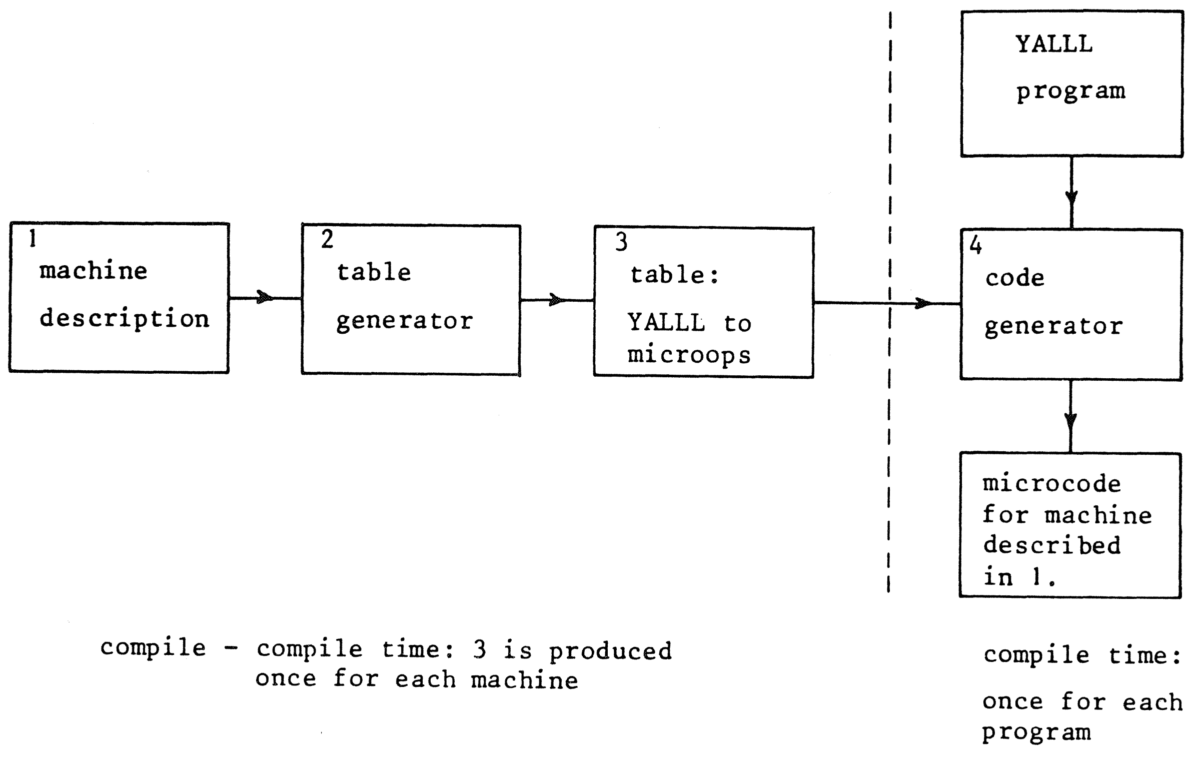

FIGURE 1: The envisaged code generation system

This paper presents a proposal for a microarchitecture description language called MIDL (Micro Instruction Description Language). The definition of this language is the first stage of the project just sketched; its suitability for code generation cannot be determined until the generator is implemented. Its implementation will probably reveal some as yet overlooked deficiencies in MIDL that will induce changes; this paper must therefore be considered as a progress report.

The next section will be devoted to existing models. In section 3 a fragment of a sample architecture will be informally presented. In section 4 MIDL will be introduced, using the fragment from section 3 as an illustration. Section 5 contains some concluding remarks.

2. RELATED RESEARCH

MIDL has been designed as a compromise between microinstruction models designed specifically to be used by microcode improvement techniques, and general purpose hardware description languages. In this section I will discuss the advantages and disadvantages of both, and try to formulate some design goals on the basis of this.

Special purpose models [DeW75, Tok77, Lan80] have the advantage that they are used by existing algorithms that compose a compact set of microinstructions from a given straight line sequence of microoperations (see [Lan80] for an overview), or even perform global optimizations [Fis79, Poe80]. Because these compaction techniques should preferably remain applicable, MIDL should have the same conceptual foundation as existing models: its central concept should be the microinstruction viewed as a collection of nonconflicting microoperations.

However, none of these models constitutes a suitable microarchitecture description language. They were designed for just one purpose (compaction of straight line microprograms), and all machine characteristics that are irrelevant to that purpose were left out. Microoperation semantics are not modeled beyond mere resource usage; asynchronous events cannot be modeled at all; and finally, sequencing is not considered. One aspect related to sequencing but commonly ignored by compaction algorithms is microprogram composition, i.e. the allocation of instructions to addresses in the control store. This problem is far from trivial because the calculation of successor addresses often leads to constraints. For example, if the destination of a 2-way branch is calculated by OR-ing a given register with a condition code, the two successors must have adjacent addresses.

The main advantage of general hardware description languages like ISPS [Bar77] or CDL [Chu72] is their power. ISPS, for example, is able to model a machine at each desired level of abstraction, from the purely functional level of the macroarchitecture down to the level of individual gates. An obvious choice for a microarchitecture description language is therefore a subset of such a language, tailored to the required level of abstraction. There are several reasons why such a subset is not satisfactory.

First, a minor change is necessitated by the choice to model the envisaged code generation system after the one described in [Cat77]. For reasons already explained in the introduction, the operators used in the definition of microoperation semantics should be those of the source language YALLL.

Secondly, the sequencing operations provided by hardware description languages cannot be easily tailored to the description of microsequencing. It is, for example, desirable to separate successor selection from successor initialization.

Finally, the timing primitives of hardware description languages are inadequate. In most languages, one can only distinguish sequential and parallel execution of operations. For a description language that, in accordance with the goal stated above, describes microoperations as indivisible units this is insufficient, since their execution can overlap without being fully parallel. Hence, a new scheme to model timing had to be developed.

A project with similar goals (development of an automated microprogram synthesis technique) is described in [Mue80]. This project differs from the one outlined in the introduction in two respects. First, it focuses on the synthesis process and uses a machine description that corresponds to a subset of ISP. The inadequacies of such a subset, especially the limitations imposed by the timing primitives, are noted but accepted. Secondly, the approach to the synthesis itself differs. The techniques used by Mueller have their origin in the domain of program verification by stepwise refinement; see for example [Man79]. The techniques that will be employed in my project are based on conventional, table driven code generation.

3. AN EXAMPLE ARCHITECTURE

The (informal) definition of MIDL in section 4 will be illustrated by the systematic development of an example, which defines a small part (some ALU-operations and some sequencing operations) of a microarchitecture, subsequently to be called EXMP. It is somewhat similar to the microarchitecture of the DEC PDP11/60. The section of the datapath considered here is shown in figure 2.

Operations

The EXMP ALU can perform 16 different operations, but only the following three will be considered:

1. Two's complement addition of the inputs;
2. Two's complement subtraction of the inputs;
3. Bitwise implication (A => B). This is equivalent to (NOT A) OR B.

Arithmetic operations set the N (negative ALU result), Z (zero ALU result), C (carry) and V (overflow) flag. Logical operations set only the N and Z flag, but leave the C and V flag undefined.

Three left ALU inputs will be considered: a 16-register scratchpad A; a 16-bit register X whose further purpose does not concern us here; and the low byte of X with the high byte replaced by zero's. Only one possible right input will be considered: A 16-register scratchpad B. The ALU output is connected to a register D; loading that register is under microprogram control.

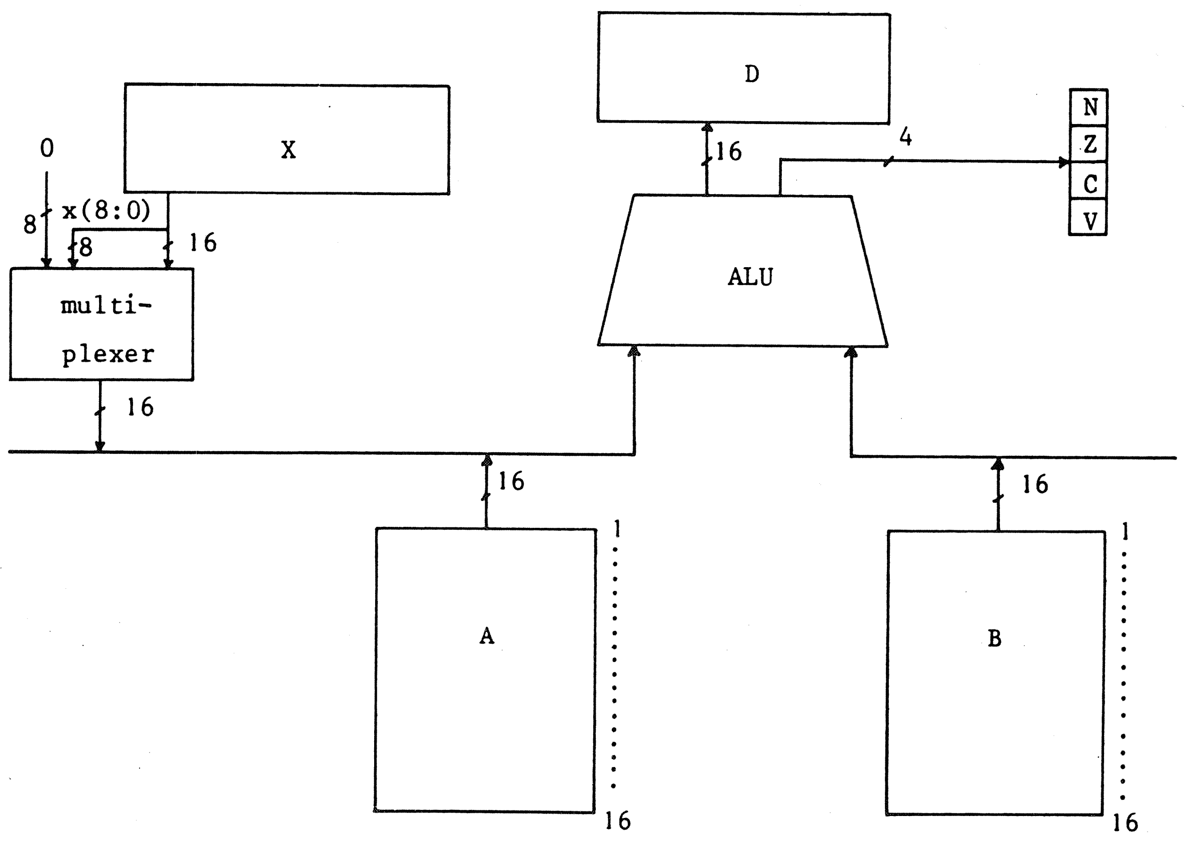

FIGURE 2: Part of the EXMP datapath

Timing

Each microcycle is divided into four phases. The ALU-operation starts execution in the first phase; the D-register can be clocked either in the second or in the third phase. The latter possibility should be used only if the ALU performs a logical operation, because the result of an arithmetical operation is not available until the third phase.

Sequencing

The EXMP microarchitecture has four sequencing operations:

- proceed with the next instruction;
- skip the next instruction;
- jump to the instruction following the one whose address is in the AMPC register (AMPC standing for alternative microprogram counter);
- likewise, but also load AMPC with the address of the current instruction.

Each microinstruction specifies a condition to be tested (N, Z, C, or V), a successor if that condition is true, and a successor if the condition is false. Specifying for example, "N, next, skip" will execute the next instruction if N equals 1, and skip it if N equals 0. Specifying "N, jump, jump" will jump regardless the value of N.

Micro-instruction fields

```
                                                   WHEN
                                                    |  CLKD
..__________________________________________________|__|__..________________________
  | AIN  | BIN  |  AB  |        ALU      | ASEL| BSEL|  |  |  | COND| TRS  | FAS |
..|______|______|______|_________________|_____|_____|__|__|..|_____|______|_____|
   49 48 47 46  45 44  43 42 41 40 39 38 37 36 35 34        5  4   3  2   1  0
```

ALU operation selection is controlled by the ALU field. ALU operand selection is fairly complicated. The left input is primarily controlled by the field AIN. If AIN designates the X register, bit 39 determines whether its high byte is masked off. If it designates the A-scratchpad, the ASEL field provides the 2 most significant bits of the index, and the complement of the AB field provides the 2 least significant bits. Selection of the right input is likewise controlled by BIN, BSEL and AB. It follows that the A and B indices are interdependent: If one of them is fixed only four possible choices are left for the other. The CLKD field denotes whether the ALU result should be clocked into the D register. The WHEN field denotes whether clocking the D register occurs during phase 2 or phase 3; this field is relevant only if CLKD is set to 1. The COND field specifies the flag that governs the choice of a successor. TRS (TRue Successor) specifies the sequencing operation executed if the flag equals 1; FAS (FAlse Successor) specifies the operation executed if the flag equals zero.

4. THE MICRO INSTRUCTION DESCRIPTION LANGUAGE

4.1. Overview

In this subsection the global structure of a MIDL definition is explained, guided by an abridged BNF syntax of the language. Details are postponed until later subsections. In addition to standard BNF notation, some non-standard notational conventions are used:

- Terminal keywords are underlined; other terminal symbols are quoted.

- { A ´s´}+ denotes a list of one or more A´s separated by ´s´ symbols.
  { A ´s´}* denotes a similar list, except that it may also be empty.

MIDL is a declarative language. It defines the semantics of microoperations in terms of YALLL operators. Furthermore, it defines triggering conditions for microoperations and their operands, and it defines how the operations are timed. The specifics of YALLL do not affect the structure of MIDL. Replacement of YALLL by some other language would merely result in another dialect.

The organization of a MIDL description is similar to that of a conventional program. It consists of a list of declarations, followed by a privileged declaration called the "root".

A MIDL description consists of the following kinds of declarations:

```
<MIDL_description>:      {<declaration> ';'}+ <root> .

<declaration>:             <resource_declaration>
                         | <field_declaration>
                         | <operation_selection_rule>
                         | <operand_selection_rule>
                         | <schedule_declaration> .
```

Resource declarations describe machine resources such as registers, along with some of their properties. Field declarations describe the lay-out of microinstructions. The details of these two kinds of declarations are given in section 4.2 and 4.3.

Operator selection rules and operand selection rules define microoperations and their operands; schedule declarations specify the timing of operations. Rules of these three types may contain references to other rules. Such a reference has the semantics of a macro-call. The rules thus form a hierarchy. The top of that hierarchy is the root. Operation selection rules, operand selection rules and timing declarations will now be considered in turn.

<u>Operator</u> <u>selection</u> <u>rules</u>.

```
<root>:                        root <identifier>
                                     <operation_expressions>
                               end .

<operation_selection_rule>: operation <identifier>
                                    <operation_expressions>
                                end .

<operation_expressions>:    {<operation_expression> ';'}+ .
```

```
<operation_expression>:       <microoperation>
                             | if <condition>
                               then <operation_expressions>
                               [else <operation_expressions>]
                               fi
                             | case <operand>
                               of ( (<integer> ':')+
                                      (<operation_expression> ';')+
                                  )+
                               esac
                             | during <schedule>
                               do <operation_expressions> od
                             | <identifier> .
```

Root and operation selection rules have the same structure: a list of operation expressions. The order of these expressions is irrelevant; the execution order of microoperations is only dictated by the timing information.

A microoperation in this context is a small YALLL program, which defines the effect of the operation on the machine state. If an operation involves operand selection, its declaration refers to an operand selection rule elsewhere in the same description. Details of microoperation definitions are considered in sections 4.4 and 4.6. The latter subsection is devoted to sequencing operations.

The *if*- and *case*-constructs are familiar from conventional programming languages. They specify triggering conditions for microoperations.

The *during*-construct associates timing information with all operations within its scope. Inner *during*-constructs take precedence over outer ones. This allows the easy modeling of local exceptions to a global timing scheme. Each microoperation should be within the scope of at least one *during*-construct. If all operations are uniformly timed, one such construct at the outermost level in the root declaration suffices. Schedules are either defined in line, or refer to a schedule declaration.

An operation expression may contain references to other operation selection rules.

Operand selection rules

```
<operand_selection_rule>:    operand <identifier>
                                     <operand_expression>
                             end .
```

```
<operand_expression>:        <operand>
                           | if <condition>
                             then <operand_expression>
                             [else <operand_expression> ]
                             fi
                           | case <operand>
                             of ( (<integer> ':')+
                                    <operand_expression>
                                )+
                             esac .
```

The body of an operand selection rule may contain (possibly nested) _if_- and _case_-constructs. Note the absence of the _during_-construct here: in MIDL all timing information is associated with microoperations, even if it concerns individual operands. Details of operands and operand selection are presented in section 4.5.

Schedule declarations

```
<schedule_declaration>:    schedule <identifier>
                                    <schedule_expression>
                           end .

<schedule_expression>:       <schedule>
                           | <identifier>
                           | if <condition>
                             then <schedule_expression>
                             [else <schedule_expression> ]
                             fi
                           | case <operand>
                             of ( (<integer> ':')+
                                     <schedule_expression>
                                )+
                             esac .
```

A schedule defines the timing of a microoperation. It specifies the phase or phases in which the operation is executed. If necessary, it can be broken up to reveal the phases during which individual resources are involved in the operation.

_If_- and _case_-constructs are allowed in schedule declarations to model the (rare) cases in which the timing of a microoperation is under microprogram control. The details of schedules will be considered in section 4.7.

The following subsections will be devoted to a more detailed discussion of the various MIDL constructs. Because examples are more suitable for a detailed explanation than syntax rules, the MIDL definition of EXMP will be used throughout.

4.2. Resource declarations

Like the microinstruction model presented in [Lan80], MIDL divides resources into two classes. Permanent resources model registers; they retain their value until it is explicitly replaced. Transient resources model buses, multiplexers, and the like; they retain their value for only a limited number of (sub)cycles.

With each resource a bit-dimension (denoted by (b1:b2)) and a block-dimension (denoted by [b1:b2]) can be associated. If a resource is dimensionless in either direction, the corresponding specification can be omitted. The bounds b1 and b2 may be any pair of nonnegative integers; 1:b2 can be abbreviated to b2.

Certain MIDL constructs require operands with a specific function, e.g. LOAD and STORE operations require a memory address and a memory buffer register. Other constructs have side-effects on resources with a specific function, e.g. the operator CALL pushes a return address on a microprocedure call stack. Such a function has to be specified in the declaration of the resource. Other functions to be specified are residual control registers, main memory, control store, microprogram counter, and four kinds of flags: zero, negative, carry and overflow. The purpose of each function indication is explained along with the construct for which it is relevant.

The declaration of the EXMP resources looks as follows:

```
permanent D(15:0),                                # The EXMP D register #
          X(15:0),                                # The X register  #
          A[16](15:0),                            # The A scratchpad #
          B[16](15:0),                            # The B scratchpad #
          WCS[0:1023](79:0) = control store,      # The EXMP 1K by #
                                                  # 80 bit control store #
          AMPC(9:0)                               # Holds jump addresses #
end;


transient (2) alu_out(15:0),                      # The ALU result #
          (1) Xtemp(15:0),                        # See section 4.5 #
          (4) Z = zero,                           # The four flags #
          (4) N = negative,
          (4) C = carry,
          (4) V = overflow
end;
```

The declaration of a transient resource specifies how long that resource retains its value. For example, the register alu_out, which represents the ALU result, must be clocked into the D register within 2 subcycles after an ALU operation. Likewise, the flags (Z, N, C and V) hold their value for only four subcycles.

4.3. Field declarations

A field declaration names microinstruction fields and enumerates their constituent bits. These do not have to be consecutive. A bit can be preceded by "~", to denote that it has to be complemented. The order in which the bits are enumerated is important, the leftmost bit is always the most significant one. Fields are allowed to overlap.

Fields are explicitly associated with a resource of type control store, mentioned at the start of the declaration. This explicit association allows the description of machines with split-level control store organization like the Burroughs Interpreter [Rei72], in which microinstructions from different memories have different lengths and formats.

These features are illustrated by the EXMP field declaration:

```
fields in WCS
        field_ain       (49:48),            # AIN #
        field_bin       (47:46),            # BIN #
        field_alu       (43:40),            # ALU. 43-40 is equivalent #
                                            # 43.42.41.40            #
        field_maskX     (39),               # To mask off X high byte #
        field_Aindex    (39.38.~45.~44),    # ASEL and AB combined #
        field_Bindex    (37.36.~45.~44),    # BSEL and AB combined #
        field_when      (35),               # WHEN #
        field_clockD    (34),               # CLKD #
        field_cond      (5:4),              # condition tested #
        field_strue     (3:2),              # TRUE successor #
        field_sfalse    (1:0)               # FALSE successor #
end;
```

The declaration of field_Aindex and field_Bindex, which represent the A and B scratchpad indices, may need clarification. The most significant half of, for example, the A index is provided by bits 39 and 38 of the microword, while its least significant half is provided by the complements of bits 45 and 44. A condition like "field_Aindex = 6" (0110 in binary) is thus to be interpreted as "bit 39 equals 0, bit 38 equals 1, bit 45 equals 0, bit 44 equals 1".

4.4. Operation selection rules

Operation selection rules serve two functions: they define the semantics of microoperations, and they associate triggering conditions and timing information with them. This section is devoted to the definition of microoperations that do not change the flow of control in the microprogram.

Microoperation semantics are defined by small YALLL programs surrounded by BEGIN and END statements If such a program consists of only one statement, the BEGIN and END brackets may be omitted. MIDL provides all register transfer operations provided by YALLL: addition, subtraction, logical AND, OR, XOR (exclusive or), CMPL (complement), several shifts, MOVE for transfer between registers, and LOAD and STORE for memory references. For more details the reader is referred to [Pat79] or [Tuc79]. The format of the operations is

```
OPCODE destination sources(s) (FLAGS)
```

FLAGS specifies which condition codes are defined by the operation.*) The resources mentioned in FLAGS must have been declared with a function indication zero, negative, carry or overflow. That function indication determines by which condition the flag is governed.

Three operation selection rules define the EXMP datapath operations: the first one describes how the ALU operation is selected, the second one models the clocking of the D register when field_clockD is set, and the third one describes the replacement of the high byte of the X registers by zeros. I will consider each rule in turn.

```
operation alu_op                  # ALU operation selection #
        during 2-3
        do case field_alu
           of   0:      ADD_2   alu_out, a_in, b_in <Z,N,C,V>;
                1:      SUB_2   alu_out, a_in, b_in <Z,N,C,V>;
                2:      during 2
                        do      BEGIN    loc;
                                CMPL     loc, a_in;
                                OR       alu_out, loc, b_in <N,Z>;
                                END
                        od
                    ...
           esac
        od
end;
```

*) This differs slightly from the original YALLL definition.

ADD_2 and SUB_2 denote two's complement addition and subtraction, respectively. The sources a_in and b_in refer to operand selection rules defined in the next subsection. YALLL has no implication operation, therefore the EXMP implication is defined using the CMPL and OR operations. The local variable loc does not correspond to either a machine resource or an operand selection rule, but is only introduced for the sake of the description. It provides communication between YALLL statements.

The schedules in alu_op are simple: the addition and subtraction operations are executed during phases 2 and 3, while the implication operation is executed during phase 2.

```
operation clockD                 # Clocking of D register #
        if field_clockD = 1
        then    during phase_clockD do MOVE D, alu_out od
        fi
end;
```

The only remarkable thing about this declaration is the during construct, which refers to a schedule phase_clockD. The declaration of this schedule will be shown in section 4.7.

```
operation maskX          # Mask operation on X register #
        during 1
        do AND Xtemp, X, %377 od          # %377 means 377 octal #
end;
```

Xtemp corresponds to the bus that receives the masked X register value. According to this declaration, the X register is masked even if the result is not selected as an ALU-input. Because Xtemp is a transient resource, the result, if not selected, is lost at the end of the cycle. This description is therefore equivalent to one that associates the same conditions with the masking operation and with the selection of the result.

4.5. Operand selection rules

Operand selection rules model the selection of operands. MIDL allows five operand forms:

1. A register. If selected from a register file, the index can be any other legal operand form; not only A[field_Aindex] but also A[D(8:6)] would be a legal operand.
2. A register field; e.g. D(8:6).

3. A reference to another operand selection rule;
4. A decimal, binary, octal or hexadecimal constant;
5. A microinstruction field.

The following rules model the selection of the EXMP ALU inputs:

```
operand a_in               # Left ALU input #
        case field_ain
        of      0:         case field_maskX
                           of      0:      X;
                                   1:      Xtemp
                           esac;
                1:         ...
                2:         ...
                3:         A[field_Aindex]
        esac
end;

operand b_in               # Right ALU input #
        case field_bin
        of      0:         B[field_Bindex]
                     ...
        esac
end;
```

Note, that thanks to the encoding of the index fields as specified in the field declarations (4.3), indexing of the A and B scratchpads is now completely straightforward.

4.6. Sequencing operations

The MIDL sequencing operations are based on the following conceptual model. At each moment, all known successors of the current microinstruction are held in a sufficiently large buffer. There are four operations that fetch instructions from control store and put them in the buffer: JUMP, CALL, RETURN and MULTIJUMP (for details see below). These operations have random write access to the buffer. By default, an instruction to be written to the buffer is taken to be the immediate successor of the currently executing one and is put into the first buffer location. Another operation, called INIT, signals the start of the execution of a new instruction. This operation always takes the instruction to be initialized from the first location of the buffer, and at the same time moves all remaining instructions up by one position.

Some examples will illustrate this model. In all these examples, the initial situation is the same. The microinstruction with address 512 is executing, and the buffer is empty. Figures 3a, 3b and 3c picture the successive states of the buffer for three different microoperation sequences. The instructions that are executing are also shown. After initialization of a successor, the previous instruction does not necessarily terminate: their execution may overlap. This should be reflected in the timing specifications.

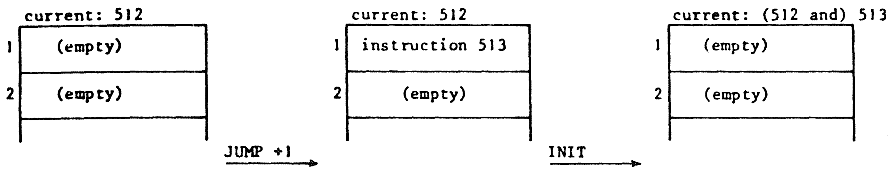

FIGURE 3a: straight line successor

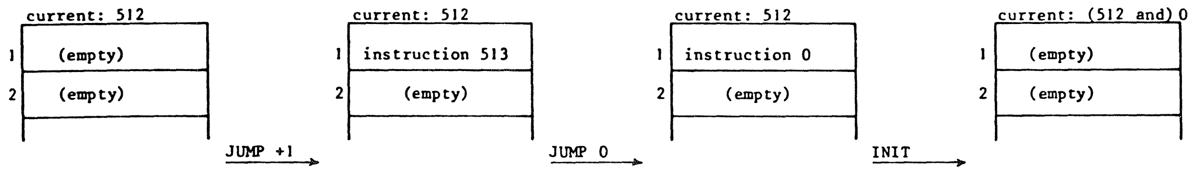

FIGURE 3b: straight line default replaced by branch

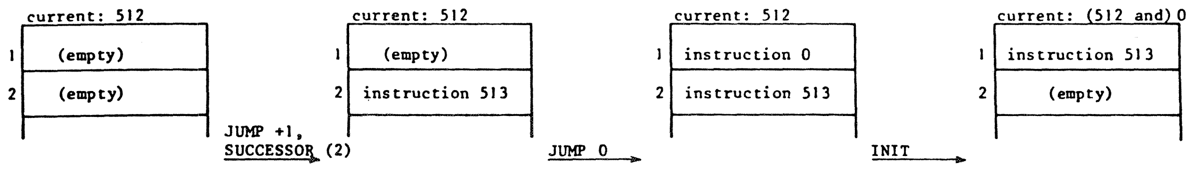

FIGURE 3c: out-of-line execution of instruction 0.

(a) JUMP +1
    INIT

The first microoperation indicates that the next instruction in control store is the immediate successor; the second one initializes its execution.

(b)
```
JUMP +1
JUMP 0
INIT
```

In this example, the second JUMP to absolute address 0 overrules the previously specified JUMP to the next instruction, as both are (by default) loaded into the first location of the buffer.

(c)
```
JUMP +1, SUCCESSOR(2)
JUMP 0
INIT
```

In this example, the default successor (the next instruction in control store) is placed second in the buffer, which is indicated by the operand SUCCESSOR(2). The instruction with absolute address 0 is loaded into the first location, and is therefore initialized first. If that instruction does not specify another JUMP, execution will afterwards continue with instruction 513. Hence, this operation sequence shows how out-of-line execution of a single instruction can be modeled.

Of the four sequencing operations, the first three are straightforward. JUMP specifies a successor address, CALL does the same but also saves an address in an appointed register. The address to be saved and the location where it is to be saved are the third and second parameter of CALL. RETURN jumps back to the instruction pointed to by the register most recently saved. If the second operand of CALL is a register declared with function indication <u>address-stack</u>, CALL performs a push instead of simply loading the register; a subsequent RETURN then automatically performs a pop.

All sequencing operations will load a resource declared with function indication <u>mpc</u> (microprogram counter) with the address of the instruction most recently fetched.

The addresses required by sequencing operations are specified as BASE +/- OFFSET, where BASE admits any operand form described in the previous subsection, and OFFSET is some constant. If the address of the current instruction serves as base address, it can be omitted: JUMP +1 appoints the next instruction as the successor. If a base is specified, a zero offset may be omitted.

MULTIJUMP is less straightforward. It specifies a dynamically calculated successor address, which is the result of OR-ing a base address (often provided by a field of the current microinstruction) with a mask composed of one or more conditions. This multiway jump is included as a separate operation, even though its effect could be modeled in terms of the ordinary JUMP. This particular way of address calculation is fairly common, however. It complicates microprogram composition (that is, the assignment of addresses to otherwise complete microinstructions), because it leads to restrictions on the successor addresses. Such a complication is better handled if its cause is explicitly modeled.

Sequencing operations can have two more operands. If more than one resource is declared with function indication control store, the store in which the successor resides must be specified in addition to its address. As already shown in example 3c, the successor's location in the buffer is indicated by SUCCESSOR(n). SUCCESSOR(1) may be omitted.

I will now show the MIDL description of the sequencing of EXMP. An EXMP microinstruction always specifies two possible successors and a flag whose value determines which one is executed.

```
operation successor
        during 4
        do      if flag = 1
                then    case field_strue
                        of      0:      JUMP +1;
                                1:      JUMP +2;
                                2:      JUMP AMPC+1;
                                3:      CALL AMPC, AMPC, +1
                        esac
                else    case field_sfalse
                        of      0:      JUMP +1;
                                1:      JUMP +2;
                                2:      JUMP AMPC+1;
                                3:      CALL AMPC, AMPC, +1
                        esac
                fi
        od &
        during 5 do INIT od
              # The 5-th phase of the current instruction coincides #
              # with the first phase of its successor; see also 4.7. #
end;

operand flag
        case field_cond
        of      0:      Z;
                1:      N;
                2:      V;
                3:      C
        esac
end;
```

The choice of a successor is determined by the condition "flag = 1". The flag tested is selected by field_cond, as described in selection rule "flag".

4.7. The MIDL timing model

The MIDL timing model is based on the one presented in [Tok77]. This model describes the timing of a microoperation by specifying during which phases each individual resource is used. The MIDL model allows a similar specification, but a linear notation was devised to replace the 2-dimensional diagrams used by Tokoro. There are some differences. First, MIDL allows the modeling of operations whose timing is under microprogram control. Secondly, MIDL provides an asynchronous timing primitive. Finally, the possibility to declare transient resources adds descriptive power, because it leads to constraints on the timing of microoperations using such resources: their execution cannot be further apart than the lifetime of the transient value.

MIDL assumes that a microcycle is divided into a number of phases. Each microoperation except INIT (see below) takes at least one phase to complete. If the architecture is not pipelined, it is not necessary to describe the exact timing of each individual resource. In that case, a schedule is a simple integer or range of integers, as was illustrated by the during-constructs in the EXMP rules considered so far.

A first refinement consists in the distinction between sources and destinations. Consider for example this pipelined timing scheme:

```
                          1         2         3         4         5
instruction 1:      |   read    | execute |  write   |
instruction 2:                  |  read   | execute  |  write   |
instruction 3:                            |   read   | execute  |  write   |
```

This scheme is described by the following schedule.

```
schedule pipeline
        read    1,
        exec    2,
        write   3
end;
```

If necessary, this can be even further refined by separately specifying the timing of individual resources. Suppose, that reading a source ROM takes longer than reading other sources. The timing of a microoperation using ROM as an operand is modeled by either of the following schedules.

```
schedule exception1                     schedule exception2
        1 (ROM: 1-2)                            read    1 (ROM: 1-2),
end;                                            exec    2,
                                                write   3
                                        end;
```

There is a subtle difference between these two schedules. The first one states that the associated operation takes longer to execute if ROM is selected as an operand. The second one states that reading ROM itself takes longer. In the above example (exception2), it even takes too long: it will cause illegal overlap between the read and the execution phase of the operation.

Schedule declarations are allowed to contain if- and case-constructs just like operation and operand selection rules. This is necessary to model, for instance, the timing of the EXMP clockD operation, which is under microprogram control (see section 4.4):

```
schedule phase_clockD
        case field_clockD
        of      0:      2;
                1:      3
        esac
end;
```

Asynchronous operations are considered to be operations whose completion is determined by conditions outside the scope of the MIDL description. It is assumed, that completion is signaled by some flag, set by the same outside agent that governs the operation. A memory reference could, for example, be described as follows:

```
during
        read    2,
        exec    until F,
        write   until F
do
        LOAD MBR, MAR
od
```

where MBR and MAR stand for the memory address and the memory buffer register respectively. Execution of the LOAD is completed when the flag F is set.

A final remark should be made about the INIT operation. Execution of INIT does not take any time. It is a synchronization primitive, designating the phase of the current microinstruction that coincides with the first phase of the next. This is not necessarily always the same phase, e.g.

```
if (some branching condition)
then    during 2 do JUMP alternative    od &
        during 3 do INIT                od
else    during 1 do JUMP +1             od &
        during 2 do INIT                od
fi
```

describes a sequencing scheme where a prefetched, straight-line successor starts execution in the second phase of its predecessor, while the first instruction following a branch is delayed.

4.8. The root

The *root* is a special operation selection rule, serving a dual purpose. First, it designates the top of the rule hierarchy. Secondly, it allows the association of defaults with all operations. For example, a *during* construct surrounding the root body specifies a default timing for all microoperations. The EXMP root consists only of references to all other operation selection rules:

```
root EXMP
        alu_op &
        clockD &
        maskX &
        successor
end;
```

5. CONCLUSIONS

MIDL has two design goals. It must be possible to model most existing microarchitectures in MIDL, and the resulting descriptions must be suitable to drive a machine independent code generator. Whether MIDL meets the latter requirement cannot yet be decided; but some remarks can be made about its modeling power.

MIDL combines features from declarative hardware description languages (general structure) with features from special purpose microinstruction models (timing, the concept of transient resources). In addition, it has some new features, notably the field encoding mechanism (originally suggested to me by A. Tanenbaum) and the sequencing model.

One input to the design of MIDL was [Agr76], in which some 13 microarchitectures are described. While working on the design, I kept a list of problematic features encountered there, and occasionally checked whether MIDL would be able to handle them. With one exception (the QM1, see below) this was indeed the case. This indicates that the modeling power of MIDL is satisfactory.

MIDL also has some weak points. These are a consequence of two decisions that have facilitated its design but limited its power.

First, MIDL models only one architectural level, without regard to others. This implies, that a code generator based on MIDL will generate code that completely ignores the existence of a conventional machine level, and therefore does not guarantee that the contents of conventional machine registers will be saved: this will have to be taken care of separately. It also implies that MIDL cannot model the two-level structure of the Nanodata QM-1, or at least, cannot model the nanolevel and the (interpreted) microlevel of that machine at the same time.

Secondly, MIDL ignores the existence of the outside world. Especially microtraps present difficult problems to microcode generators. Although these problems may provide an interesting research topic, I have initially decided to treat them like virtually all other papers on design or implementation of higher level microprogramming languages: I have ignored them.

Acknowledgements

A. van de Goor and A. Tanenbaum have made valuable contributions to the design of MIDL. Critical reading of this manuscript by J. Heering, P. Klint and A. Veen markedly improved the presentation.

Literature

[Agr76] A.K. Agrawala & T.G. Rauscher, "Foundations of Microprogramming", Academic Press, (1976).

[Bar77] M.R. Barbacci, G.E. Barnes, R.G. Cattell & D.P. Siewiorek, "The ISPS Computer Description Language", CSD, Technical Report, Carnegie Mellon University, (1977).

[Cat77] D.G. Cattell, "Formalization and Automatic Derivation of Code Generators", Ph.D. Thesis, Tech. Report TR 78-115, Computer Science, Carnegie Mellon University, Pittsburgh Pa, (1977).

[Cat80] R.G.G. Cattell, "Automatic Derivation of Code Generators from Machine Descriptions", ACM Transactions on Programming Languages and Systems, Vol. 2, No. 2, pp 173-190, (1980).

[Chu72] Y. Chu, "Introducing the Computer Design Language", IEEE Computer Conference COMPCON72, San Francisco, pp. 215-218, (1972).

[DeW75] D. J. DeWitt, "A Control Word Model for Detecting Conflicts Between Microprograms", Proceedings of the 8-th Annual Workshop on Microprogramming, pp. 6-12, (1975).

[Fis79] J.A. Fisher, "The Optimization of Horizontal Microprograms within and beyond Basic Blocks: An Application of Processor Scheduling with Resources", Ph.D. Thesis, Department of Mathematics and Computing, New York University, (1979).

[Lan80] D. Landskov, S. Davidson, B, Shriver & P.W. Mallett, "Local Microcode Compaction Techniques", Computing Surveys, Vol. 12, no. 3, pp. 261-294, (1980).

[Man79] Z. Manna & R. Waldinger, "Synthesis: Dreams -> Programs", IEEE Transactions on Software Engineering, Vol. SE-5, No.4, pp. 294-327, (1979).

[Mue80] R.A. Mueller, "Formalization and Automated Synthesis of Microprograms" Proceedings of the 13-th Annual Workshop on Microprogramming, pp. 45-53, (1980).

[Pat79] D. Patterson, K. Lew & R. Tuck, "Towards an Efficient, Machine-Independent Language for Microprogramming", Proceedings of the 12-th Annual Workshop on Microprogramming, pp. 22-35, (1979).

[Poe80] M.D. Poe, "Heuristics for the Global Optimization of Microprograms", Proceedings of the 13-th Annual Workshop on Microprogramming, pp. 13-22, (1980).

[Rei72] W. Reigel, V. Farber & D.A. Fisher, "The Interpreter - A Microprogrammable Building Block System", AFIPS Conference Proceedings, Vol. 40, pp. 705-723, (1972).

[Sin80] H.J. Sint, "A Survey of High Level Microprogramming Languages", Proceedings of the 13-th Annual Workshop on Microprogramming, pp. 141-153, (1980).

[Tok77] M. Tokoro, E. Tamura, K. Takase & K. Tamaru, "An Approach To Microprogram Optimization Considering Resource Occupancy and Instruction Formats", Proceedings of the 10-th Annual Workshop on Microprogramming, pp. 92-108, (1977).

[Tuc79] R.D. Tuck, "Software Microprogramming Tools for the VAX-11/780", Memorandum No. UCB/ERL M79/65, Electronics Research Laboratory, University of California, Berkeley, (1979).