A Formal Specification of Some User Mode Instructions for the Motorola 68020

Robert S. Boyer and *Yuan Yu*

Technical Report TR-92-04
Computer Sciences Department
University of Texas at Austin
Austin, Texas 78712

February, 1992

Contents

A Formal Specification of Some User Mode Instructions for the Motorola 68020[1]

Robert S. Boyer and *Yuan Yu*

Computer Sciences and Mathematics Departments
University of Texas at Austin
Austin, Texas 78712

February, 1992

telephone: (512) 471-9745
email: boyer@cs.utexas.edu or yuan@cs.utexas.edu

Abstract. We present a formal specification of approximately 80% of the 'user mode' instructions of the Motorola MC68020 microprocessor. The specification is given in the form of definitions in the logic of Nqthm, the Boyer-Moore system. The definitions are displayed in a conventional mathematical syntax. The specification has been used in the mechanical verification of several dozen machine code programs, whose binary was generated by 'industrial strength' C and Ada compilers.

1 Introduction

This report contains a formal specification of approximately 80% of the 'user mode' instructions of the Motorola MC68020 microprocessor. An earlier report [3] describes how we have used this specification to prove mechanically the correctness of several dozen machine code programs, most of them generated by 'industrial strength' compilers for C or Ada. Our specification is based upon the user's manual for the MC68020 [4].

The function definitions below are ordered so that a function is defined before it is referenced by another function. One of the very last functions defined, 'stepn', p. 109, emulates the MC68020. Like all the functions in this specification, 'stepn' is a recursive and hence computable function. Approximately speaking, if we are given an MC68020 state s and a positive integer n, we can

[1]The work described here was supported in part by NSF Grant MIP-9017499.

compute the state *s'* that results from executing an MC68020 for *n* instructions, starting in state *s*, by applying 'stepn' to *s* and *n*. If an illegal instruction or an instruction not among those covered in this specification is encountered during execution, then *s'* will exhibit an indication of the error. If no such error indication is exhibited, then the returned state correctly represents the state that a 'real' MC68020 would have after running *n* instructions provided that (i) the caches are initially consistent with memory, (ii) no interrupts happen during execution, and, of course, (iii) no externally caused changes to the state occur during execution. In Section 15 is a theorem that illustrates the use of 'stepn' to emulate an MC68020 on a specific state, one that contains machine code for Euclid's GCD algorithm.

Disclaimer: The development of this formal specification is part of a small scientific project aimed at examining the feasibility of mechanically checking the correctness of machine code programs that run on widely-used microprocessors. The accuracy with which the specification presented here represents a 'real' MC68020 is something we do not know how to ascertain with the certainty of a mathematical proof. One can only become increasingly confident by such activities as critical reading, testing, and bug fixing. It is in a spirit of scientific cooperation that we distribute this specification, but we distribute it without any warranty of any kind, on an 'as is' basis.

The definitions below were written in the logic described in *A Computational Logic Handbook,* [2], with syntactic extensions for 'let' and 'cond'. The definitions have been admitted under the definitional principle described in that book, using the mechanical theorem prover also described in that book. Although the logic and prover use the prefix, parenthesized notation of Church's lambda calculus and McCarthy's Lisp, in this report, we use a notation that is conventional. This new syntax is summarized in Section 17.

Our principal purpose in writing this technical report is to communicate precisely the formal, mathematical definitions of our specification of the MC68020. This report is decidedly not a tutorial on the MC68020 or on our specification of it. The reader will find it easier to read this specification after having read [3]. The reader will also find it invaluable to have a copy of [4] handy. Readers in search of a less stark introduction to this specification will find it in the forthcoming Ph. D. dissertation of Yuan Yu. There also will be found a review of the related scientific literature.

2 A Few Basic Functions

The objects we use in this specification are truth values, integers, ordered pairs, and symbols. The precise axioms and notations for these objects and the built-in functions that operate on these objects may be found in [2]. Here are a few brief remarks about some of these objects and functions.

- The constant 'true', abbreviated **t**, is the true truth value.

- The constant 'false', abbreviated **f**, is the false truth value.
- 'if' is a function of three arguments. **if** x **then** y **else** z **endif** returns z if x is equal to **f**, and y otherwise.
- 'cons' is a function of two arguments. cons (x, y) returns an ordered pair whose first component is x and whose second is y.
- 'car' is a function of one argument. car (x) returns the first component if x is an ordered pair; otherwise, it returns 0.
- 'cdr' is a function of one argument. cdr (x) returns the second component if x is an ordered pair; otherwise, it returns 0.
- 'list' is a function of any number of arguments. list (x, y, z) is cons (x, cons (y, cons (z, **nil**))). **nil** is a symbol, and is used to denote the empty list.

Except for **nil**, symbols are printed in a typewriter font, preceded by a single quotation mark, e.g., `'running` and `'read_unavailable_memory`.

3 Start Up

EVENT: Start with the library `"mc20-0"`.

Our initial library `mc20-0` contains (i) the basic axioms and definitions of Nqthm, which are described in Chapter 4 of [2], (ii) some proved arithmetic lemmas that are used to help in the admission of the following definitions, and (iii) a definition of the nonnegative integer exponentiation function 'exp', which is defined as: exp (x, y) = **if** $y \simeq 0$ **then** 1 **else** $x *$ exp (x, $y - 1$) **endif**.

4 Some Constants

We first define a few constants.

In the MC68020, a "byte" is 8 bits long. A "word" is 16 bits long. A "long word" is 32 bits long. A "quad word" is 64 bits long.

DEFINITION: B = 8

DEFINITION: W = 16

DEFINITION: L = 32

DEFINITION: Q = 64

DEFINITION: BSZ = `1`

DEFINITION: WSZ = `2`

DEFINITION: LSZ = `4`

DEFINITION: QSZ = `8`

Some error signals.

DEFINITION: READ-SIGNAL = `'read_unavailable_memory`

DEFINITION: WRITE-SIGNAL = `'write_rom_or_unavailable_memory`

DEFINITION:
RESERVED-SIGNAL = `'motorola_reserved_for_future_development`

DEFINITION: PC-SIGNAL = `'pc_outside_rom`

DEFINITION: PC-ODD-SIGNAL = `'pc_at_odd_address`

DEFINITION:
MODE-SIGNAL
=
`'illegal_addressing_mode_in_current_instruction`

Throughout our specification, we have frequent need to refer to bits and bit-vectors. In our model, bit ::= 0 | 1, and bit-vectors ::= nonnegative integers. If the operation is signed, we use the two conversion functions 'nat-to-int' and 'int-to-nat.'

'bitp' is a function of one argument, x. 'bitp' returns **t** or **f** according to whether x is a bit or not.

DEFINITION: bitp (x) = ((x = `0`) $\vee$ (x = `1`))

We frequently use the bits 0 and 1. For clarity, to identify informally when we are using these integers as bits, we use the two constants 'b1' and 'b0.'

DEFINITION: B1 = `1`

DEFINITION: B0 = `0`

We frequently test a bit to see whether it is 0 or 1. We define the functions 'b1p' and 'b0p' to return **t** or **f** according to whether their arguments are 0 or non-0 respectively.

DEFINITION: b0p (x) = (x = B0)

Definition: b1p$(x) = (x \neq$ B0$)$

Definition:
```
fix-bit (c)
   =
if b0p (c)  then B0
else B1 endif
```

Here are the definitions of some operators for logical arithmetic on bits.

'b-not' returns the complement of its argument.

Definition:
```
b-not (x)
   =
if b0p (x)  then B1
else B0 endif
```

'b-and' returns the logical and of its two arguments.

Definition:
```
b-and (x, y)
   =
if b0p (x)  then B0
elseif b0p (y)  then B0
else B1 endif
```

'b-or' returns the logical or of its two arguments.

Definition:
```
b-or (x, y)
   =
if b0p (x)
then if b0p (y)  then B0
      else B1 endif
else B1 endif
```

'b-nor' returns the logical nor of its two arguments.

Definition:
```
b-nor (x, y)
   =
if b0p (x)
then if b0p (y)  then B1
      else B0 endif
else B0 endif
```

'b-nand' returns the logical nand of its two arguments.

DEFINITION:
```
b-nand ($x$, $y$)
    =
**if** b0p ($x$) **then** B1
**elseif** b0p ($y$) **then** B1
**else** B0 **endif**
```

'b-eor' returns the exclusive or of its two arguments.

DEFINITION:
```
b-eor ($x$, $y$)
    =
if b0p ($x$)
then if b0p ($y$) then B0
       else B1 endif
elseif b0p ($y$) then B1
else B0 endif
```

'b-equal' returns the logical equal of its two arguments.

DEFINITION:
```
b-equal ($x$, $y$)
    =
if b0p ($x$) then b0p ($y$)
else b1p ($y$) endif
```

5 Bit Vector Arithmetic

'bcar' returns the first bit of x.

DEFINITION: $\mathrm{bcar}(x) = (x \ \mathbf{mod}\ 2)$

'bcdr' returns a natural number by cutting off the first bit of x. For any natural number x, $(\mathrm{bcar}(x) + (x * \mathrm{bcdr}(x))) = x$.

DEFINITION: $\mathrm{bcdr}(x) = (x \div 2)$

'head' is a function of two arguments, x and n. x and n should be nonnegative integers. 'head' returns the remainder of x divided by 2^n.

DEFINITION: $\mathrm{head}(x, n) = (x \ \mathbf{mod}\ \exp(2, n))$

'tail' is a function of two arguments, x and n. x and n should be nonnegative integers. 'tail' returns the quotient of x divided by 2^n.

DEFINITION: $\mathrm{tail}(x, n) = (x \div \exp(2, n))$

We next define some logical operations on bit-vectors. 'lognot' takes two naturals as its arguments and returns the logical complement of its second argument.

DEFINITION: $\text{lognot}(n, x) = ((\exp(2, n) - \text{head}(x, n)) - 1)$

'logand' takes two naturals as arguments and returns their logical and.

DEFINITION:
$\text{logand}(x, y)$
  $=$
if $(x \simeq 0) \vee (y \simeq 0)$ **then** 0
else $\text{b-and}(\text{bcar}(x), \text{bcar}(y)) + (2 * \text{logand}(\text{bcdr}(x), \text{bcdr}(y)))$ **endif**

'logor' takes two naturals as arguments and returns the logical (inclusive) or of the two arguments.

DEFINITION:
$\text{logor}(x, y)$
  $=$
if $x \simeq 0$ **then** $\text{fix}(y)$
elseif $y \simeq 0$ **then** $\text{fix}(x)$
else $\text{b-or}(\text{bcar}(x), \text{bcar}(y)) + (2 * \text{logor}(\text{bcdr}(x), \text{bcdr}(y)))$ **endif**

'logeor' takes two naturals as arguments and returns the logical exclusive or of the two arguments.

DEFINITION:
$\text{logeor}(x, y)$
  $=$
if $(x \simeq 0) \wedge (y \simeq 0)$ **then** 0
else $\text{b-eor}(\text{bcar}(x), \text{bcar}(y)) + (2 * \text{logeor}(\text{bcdr}(x), \text{bcdr}(y)))$ **endif**

'bitn' retrieves the nth bit of x. Indexing is 0-based.

DEFINITION: $\text{bitn}(x, n) = \text{bcar}(\text{tail}(x, n))$

'bits' returns bits i through j as a natural number. 'bits' is a function of three arguments, x, i, and j. x, i, and j should be natural numbers. Intuitively, bits extracts bits of x from bit i to bit j. Normally, i should be less than or equal to j.

DEFINITION: $\text{bits}(x, i, j) = \text{head}(\text{tail}(x, i), 1 + (j - i))$

'setn' updates the n^{th} bit of x by the given value c. Indexing is 0-based.

DEFINITION:
$\text{setn}(x, n, c)$
  $=$
if $n \simeq 0$ **then** $\text{fix-bit}(c) + (2 * \text{bcdr}(x))$
else $\text{bcar}(x) + (2 * \text{setn}(\text{bcdr}(x), n - 1, c))$ **endif**

'adder' takes four arguments and returns the addition of x, y, and c modulo 2^n. That is, $(x + y + c)$ **mod** $\exp(n, 2)$. Typically, c is either 0 or 1.

DEFINITION: $\text{adder}(n, c, x, y) = \text{head}(c + x + y, n)$

'add' takes three arguments and returns the addition of x and y modulo 2^n. That is, $(x + y)$ **mod** $\exp(n, 2)$.

DEFINITION: $\text{add}(n, x, y) = \text{head}(x + y, n)$

'subtracter' takes four arguments and returns the subtraction of y and $(x + c)$ **mod** $\exp(n, 2)$. That is, $(y - (x + c))$ **mod** $\exp(n, 2)$. Typically, c is either 0 or 1.

DEFINITION:
$\text{subtracter}(n, c, x, y) = \text{adder}(n, \text{b-not}(c), y, \text{lognot}(n, x))$

'sub' takes three arguments and returns, in the form of 2's complement, the subtraction of y and x. That is, $(y - x)$ **mod** $\exp(n, 2)$.

DEFINITION:
$\text{sub}(n, x, y) = \text{head}(y + (\exp(2, n) - \text{head}(x, n)), n)$

'replace' replaces x partially by y in the head. 'replace' is a function of three arguments, n, x and y, all of which should be naturals. 'replace' is frequently used when updating only one byte or one word in a register, leaving the other bytes alone.

DEFINITION:
$\text{replace}(n, x, y) = (\text{head}(x, n) + (\text{tail}(y, n) * \exp(2, n)))$

'app' "appends" two naturals. 'app' takes three arguments, n, x, and y.

DEFINITION: $\text{app}(n, x, y) = (\text{head}(x, n) + (y * \exp(2, n)))$

'ext' is a function of three arguments, n, x and *size*. 'ext' is used frequently to do "sign-extension". For instance, in the MC68020, we often extract a byte or word and wish to add it into a 32-bit sum, but we first sign-extend the extracted quantity to obtain a meaningful sum.

DEFINITION:
```
ext (n, x, size)
  =
if n < size
then if b0p (bitn (x, n - 1)) then head (x, n)
      else app (n, x, exp (2, size - n) - 1) endif
else head (x, size) endif
```

Shift operations.
Logical shift left.

DEFINITION: $\text{lsl}(len, x, cnt) = \text{head}(x * \exp(2, cnt), len)$

Arithmetic shift left.

DEFINITION: $\text{asl}(len, x, cnt) = \text{head}(x * \exp(2, cnt), len)$

Logical shift right.

DEFINITION: $\text{lsr}(x, cnt) = \text{tail}(x, cnt)$

Arithmetic shift right.

DEFINITION:
```
asr(n, x, cnt)
  =
if $x < \exp(2, n - 1)$ then $\text{tail}(x, cnt)$
elseif $n < cnt$ then $\exp(2, n) - 1$
else $\text{tail}(x, cnt) + ((\exp(2, cnt) - 1) * \exp(2, n - cnt))$ endif
```

6 Integer Arithmetic

Throughout most of this MC68020 specification, we restrict our attention to arithmetic on the nonnegative integers. However, in the definition of two machine instructions, those for signed multiplication and division, we also consider all of the integers, both nonnegative and negative. The Nqthm logic adds the negative integers almost as an afterthought, and the basic, built-in arithmetic operations of the Nqthm logic work only for nonnegative integers. To do arithmetic on all the integers, we must define appropriate operations explicitly, as we do below.

The Nqthm logic has the peculiarity that $-\ 0$ is not the same as 0. However, we will restrict our domain so that $-\ 0$ is not considered. A negative integer is defined to be of the form $-\ x$ with x nonzero.

DEFINITION: $\text{negp}(i) = (\text{negativep}(i) \land (i \neq (-\ 0)))$

x is an integer iff x is either a nonnegative number or a negative number.

DEFINITION: $\text{integerp}(x) = ((x \in \mathbf{N}) \lor \text{negp}(x))$

DEFINITION:
```
fix-int(x)
  =
if integerp(x) then x
else 0 endif
```

DEFINITION: $\text{izerop}(x) = (\text{fix-int}(x) = 0)$

Definition:
```
abs ($x$)
    =
if negp ($x$)  then negative-guts ($x$)
else fix ($x$) endif
```

Definition:
```
ilessp ($i$, $j$)
    =
if negp ($i$)
then if negp ($j$)  then negative-guts ($j$) $<$ negative-guts ($i$)
        else t endif
elseif negp ($j$)  then f
else $i < j$ endif
```

Definition: ileq (i, j) = ($\neg$ ilessp (j, i))

Definition:
```
iplus ($x$, $y$)
    =
if negp ($x$)
then if negp ($y$)  then $-$ (negative-guts ($x$) $+$ negative-guts ($y$))
        elseif $y <$ negative-guts ($x$)  then $-$ (negative-guts ($x$) $-$ $y$)
        else $y$ $-$ negative-guts ($x$) endif
elseif negp ($y$)
then if $x <$ negative-guts ($y$)  then $-$ (negative-guts ($y$) $-$ $x$)
        else $x$ $-$ negative-guts ($y$) endif
else $x + y$ endif
```

Definition:
```
ineg ($x$)
    =
if izerop ($x$)  then 0
elseif negp ($x$)  then negative-guts ($x$)
else $-$ $x$ endif
```

Definition: idifference (x, y) = iplus (x, ineg (y))

Definition:
```
itimes ($x$, $y$)
    =
if negp ($x$)
then if negp ($y$)  then negative-guts ($x$) $*$ negative-guts ($y$)
        else fix-int ($-$ (negative-guts ($x$) $*$ $y$)) endif
elseif negp ($y$)  then fix-int ($-$ ($x$ $*$ negative-guts ($y$)))
else $x * y$ endif
```

DEFINITION:
```
iremainder (x, y)
  =
if negp (x)  then fix-int (− (negative-guts (x) mod abs (y)))
else x mod abs (y) endif
```

DEFINITION:
```
iquotient (x, y)
  =
if negp (x)
then if negp (y)  then negative-guts (x) ÷ negative-guts (y)
       else fix-int (− (negative-guts (x) ÷ y)) endif
elseif negp (y)  then fix-int (− (x ÷ negative-guts (y)))
else x ÷ y endif
```

The size of bit vectors. 'nat-rangep' returns **t**, if nat < exp (n, 2), but returns **f**, otherwise.

DEFINITION: nat-rangep (nat, n) = (nat < exp (2, n))

The size of an unsigned integer. 'uint-rangep' returns **t**, if 0 $\leq x$ $\leq$ exp (n, 2), and returns **f**, otherwise.

DEFINITION: uint-rangep (x, n) = (x < exp (2, n))

Two conversion functions for unsigned integer interpretation.

DEFINITION: nat-to-uint (x) = fix (x)

DEFINITION: uint-to-nat (x) = fix (x)

The size of an integer. 'int-rangep' returns **t**, if ((− exp (2, n − 1)) $\leq$ int) $\wedge$ (int < exp (2, n − 1)), and returns **f**, otherwise.

DEFINITION:
```
int-rangep (int, n)
  =
if n ≃ 0  then fix-int (int) = 0
elseif negativep (int)  then negative-guts (int) ≤ exp (2, n − 1)
else int < exp (2, n − 1) endif
```

Two conversion functions for signed integer interpretation. 'nat-to-int' converts natural numbers to integers, 'int-to-nat' converts integers to natural numbers.

DEFINITION:
```
nat-to-int (x, n)
  =
if x < exp (2, n − 1)  then fix (x)
else − (exp (2, n) − x) endif
```

DEFINITION:
int-to-nat $(x,\ size)$
    =
if negativep (x) **then** exp $(2,\ size) -$ negative-guts (x)
else fix (x) **endif**

7 Binary Trees for Memory

A binary tree is either **nil** or an object of the form (*value bt0 . bt1*), where *bt0* and *bt1* are binary trees and *value* is any object stored at that node.

'value-field' is a function of one argument. 'value-field' returns the object stored at the current node, i.e., the 'car.'

DEFINITION:
value-field (bt)
    =
if listp (bt) **then** car (bt)
else 0 **endif**

'branch0' is a function of one argument, which should be a non-**nil** binary tree. 'branch0' returns the left branch, i.e., the 'cadr.'

DEFINITION:
branch0 (bt)
    =
if listp (bt) **then** cadr (bt)
else nil endif

'branch1' is a function of one argument, which should be a non-**nil** bin-tree. 'branch1' returns the right branch, i.e., the 'cddr.'

DEFINITION:
branch1 (bt)
    =
if listp $(bt) \wedge$ listp (cdr (bt)) **then** cddr (bt)
else nil endif

Construct a binary tree (*value br0 . br1*).

DEFINITION: make-bt $(value,\ br0,\ br1) =$ cons $(value,$ cons $(br0,\ br1))$

In order to execute MC68020 instructions reasonably efficiently in an applicative programming language, we implement memory using binary trees rather than simple linear lists or association lists. Binary trees give us logarithmic access and change times.

A memory state in this specification is actually given by a 'cons' of two binary trees, one that tells us 'protection' information about each byte of the memory and one that is the 'physical' memory, i.e., the byte of data stored at each 32-bit address.

A completely 'full' binary tree would contain 2^{32} tips, and the explicit representation of such a tree would vastly exceed the memory capacity of any known implementation of Nqthm. Therefore, we assign meaning to non-full, i.e., partially full, binary trees, both for protection and for data.

To characterize, informally, the content and protection of an address in memory, let us momentarily view an address as a sequence of 32 bits, most significant bit on the left. By an 'initial sequence' of an address x, we mean a sequence to which one can append another possibly empty sequence on the right to obtain x. Thus `001` is an initial sequence of `0010011`. For a given memory data tree *bt* and address x, what is the content of *bt* at x? Answer: if the subtree of *bt* obtained by taking the path through *bt* determined by any initial sequence of x is **nil**, then the content of *bt* at x is `0`. Otherwise, the content is the value field at the subtree of *bt* determined by x. In other words, if *bt* is not sufficiently deep along the path x, then the content of *bt* at x is `0`.

A memory protection tree *map* is a binary tree which has stored at each node, in the value cell, either **nil**, `'(unavailable)`, `'(rom)`, or `'(unavailable rom)`. (The last of these has the same meaning as `'(unavailable)`.) For a given memory protection tree *map* and address x, what is the protection status of *map* at x? Answer: if `'unavailable` is a member of the value cell at any subtree of *map* obtained by taking the path through *map* determined by any initial subsequence of x, then the address x is said to be unavailable, and it may not be read or written (even as part of a word or long word operation) by any instruction. Moreover, if an address x is not unavailable by the preceding rule, but `'rom` is a member of any such value cell, then the address is said to be ROM and may not be written by any instruction. Instructions must come entirely from such ROM addresses. Finally, if an address is not unavailable or ROM by the preceding rules, we say that it is RAM, and it may be read or written by any instruction.

'readp' is a function of three arguments, x, *map*, and n. *map* should be a memory protection binary tree. x should be a natural number. n is the index of the 'next bit' to select upon in x while walking the x path through *map*. Typically 'readp' is called with n initially equal to 32 and *map* equal to the current memory protection map. 'readp' returns **f** if it encounters an `'unavailable` at a node on the x path through *map* (considering only the least n significant bits of x), and otherwise *readp* returns **t**.

DEFINITION:
readp (x, n, *map*)
  =
if `'unavailable` $\in$ value-field (*map*) **then f**
elseif (*map* $\simeq$ **nil**) $\vee$ ($n \simeq$ `0`) **then t**
elseif b0p (bitn (x, $n - 1$)) **then** readp (x, $n - 1$, branch0 (*map*))

```
else readp$(x,\ n-1,\ \text{branch1}(map))$ endif
```

In our specification, programs can only be stored in ROM. The function 'pc-readp' returns **t** only when it hits a `'rom` at a node on the path x through *map* and only if there is no `'unavailable` at each node on the path x. n serves the same role it does in 'readp', as an index into x.

Definition:

```
pc-readp$(x,\ n,\ map)$
    =
if 'unavailable $\in$ value-field$(map)$ then f
elseif 'rom $\in$ value-field$(map)$ then readp$(x,\ n,\ map)$
elseif $(map \simeq \textbf{nil}) \vee (n \simeq 0)$ then f
elseif b0p$(\text{bitn}(x,\ n-1))$ then pc-readp$(x,\ n-1,\ \text{branch0}(map))$
else pc-readp$(x,\ n-1,\ \text{branch1}(map))$ endif
```

'writep' is a function of three arguments, x, n, and *map*. 'map' should be a memory protection binary tree. x should be a natural number. 'writep' returns **t** if it never encounters `'unavailable` or `'rom` at a node on the path x through *map*, otherwise **f**. n serves the same role it does in 'readp', as an index into x.

Definition:

```
writep$(x,\ n,\ map)$
    =
if ('unavailable $\in$ value-field$(map)$) $\vee$ ('rom $\in$ value-field$(map)$)
then f
elseif $(map \simeq \textbf{nil}) \vee (n \simeq 0)$ then t
elseif b0p$(\text{bitn}(x,\ n-1))$ then writep$(x,\ n-1,\ \text{branch0}(map))$
else writep$(x,\ n-1,\ \text{branch1}(map))$ endif
```

'read' is a function of three arguments, x, n, and *bt*. *bt* should be a binary tree, x and n should be natural numbers. 'read' returns the value component at the node reached by taking the path x through *bt*. n serves the same role it does in 'readp', as an index into x.

Definition:

```
read$(x,\ n,\ bt)$
    =
if $n \simeq 0$ then value-field$(bt)$
elseif b0p$(\text{bitn}(x,\ n-1))$ then read$(x,\ n-1,\ \text{branch0}(bt))$
else read$(x,\ n-1,\ \text{branch1}(bt))$ endif
```

'pc-read' acts the same as read. But it is used in a quite different sense. So we introduce this dummy function.

Definition: pc-read$(x,\ n,\ bt)$ = read$(x,\ n,\ bt)$

'write' is a function of four arguments, *value*, x, n, and *bt*. *value*, x, and n should be nonnegative integers, and *bt* should be a binary tree. 'write' returns the binary tree obtained by updating *bt* at the address x. n serves the same role it does in 'readp', as an index into x.

DEFINITION:
```
write (value, x, n, bt)
   =
if n ≃ 0 then make-bt (value, branch0 (bt), branch1 (bt))
elseif b0p (bitn (x, n − 1))
then make-bt (value-field (bt), write (value, x, n − 1, branch0 (bt)), branch1 (bt))
else make-bt (value-field (bt),
              branch0 (bt),
              write (value, x, n − 1, branch1 (bt))) endif
```

'get-nth' is a function of two arguments. The first should be a nonnegative integer and the second should be a list. 'get-nth' returns the n^{th} element of *lst*. Indexing is 0-based. For example, get-nth (0, list (a, b, c)) = a.

DEFINITION:
```
get-nth (n, lst)
   =
if n ≃ 0 then car (lst)
else get-nth (n − 1, cdr (lst)) endif
```

'put-nth' is a function of three arguments: *value*, n, and *lst*. *value* and n should be natural numbers, and *lst* should be a list. 'put-nth' returns a list like *lst* except that the n^{th} element has been changed to be *value*. Indexing is 0-based, e.g., put-nth (d, 1, list (a, b, c)) = list (a, d, c).

DEFINITION:
```
put-nth (value, n, lst)
   =
if n ≃ 0 then cons (value, cdr (lst))
else cons (car (lst), put-nth (value, n − 1, cdr (lst))) endif
```

The size of the operand, given the operation length.

DEFINITION: op-sz (*oplen*) = (*oplen* ÷ B)

'read-rn' and 'write-rn' are two functions used to fetch and modify the register *rn* in the register file *regs*.

DEFINITION:
read-rn (*oplen*, *rn*, *regs*) = head (get-nth (*rn*, *regs*), *oplen*)

DEFINITION:
write-rn (*oplen*, *value*, *rn*, *regs*)
=
put-nth (replace (*oplen*, *value*, get-nth (*rn*, *regs*)), *rn*, *regs*)

A machine state is defined to be a list of length 5, say (*status regs pc ccr memory*), whose components have the following purposes: *status*, if it is not `'running`, is the reason that execution was stopped; *regs* holds the data registers and the address registers; *pc* is the program counter; *ccr* is the 16-bit condition code register; and *memory* is the memory, including protection information. The status field is set when we encounter an instruction which we do not choose to handle for some reason. Among the many reasons that might arise for setting the status field are (1) an illegal instruction, (2) a legal MC68020 instruction (e.g., CALLM) that this specification does not handle, and (3) an illegal addressing mode. To construct a state one uses the 5 argument function 'mc-state', giving it as arguments, in order, the halt-reason, the data and address registers, the pc, the ccr, and the memory. The five fields of a state can be accessed with the five accessor functions 'mc-status', 'mc-rfile', 'mc-pc', 'mc-ccr', and 'mc-mem.'

DEFINITION:
mc-state (*status*, *regs*, *pc*, *ccr*, *mem*) = list (*status*, *regs*, *pc*, *ccr*, *mem*)

DEFINITION: mc-status (*s*) = car (*s*)

DEFINITION: mc-rfile (*s*) = cadr (*s*)

DEFINITION: mc-pc (*s*) = head (caddr (*s*), `32`)

DEFINITION: mc-ccr (*s*) = head (cadddr (*s*), `8`)

DEFINITION: mc-mem (*s*) = caddddr (*s*)

'len' is a function of one argument, *lst*, which should be a proper list. 'len' returns the length of *lst*, i.e., the number of elements in *lst.*

DEFINITION:
len (*lst*)
=
if *lst* $\simeq$ **nil then** `0`
else 1 + len (cdr (*lst*)) **endif**

'mc-haltp' returns **t** if some halting condition has been satisfied.

DEFINITION: mc-haltp (*s*) = (mc-status (*s*) $\neq$ `'running`)

8 Operands from Memory

Everything in this section is machine dependent. We assume the memory capacity is 2^{32}. In our specification, the memory is a binary tree with depth 32.

DEFINITION: byte-readp $(x,\ mem)$ = readp $(x,\ \texttt{32},\ \text{car}\,(mem))$

'read-memp' returns **t** if the k consecutive bytes in memory starting at x are readable, but returns **f** otherwise.

DEFINITION:

```
read-memp (x, mem, k)
   =
if k ≃ 0  then t
else byte-readp (add (32, x, k − 1), mem) ∧ read-memp (x, mem, k − 1) endif
```

'word-readp' determines whether both bytes of the word at the memory address x are readable.

DEFINITION: word-readp $(x,\ mem)$ = read-memp $(x,\ mem,\ \text{WSZ})$

'long-readp' determines whether all four bytes of the longword at the memory address x are readable.

DEFINITION: long-readp $(x,\ mem)$ = read-memp $(x,\ mem,\ \text{LSZ})$

Programs can only be stored in ROM. Assume that x is a pointer in some program segment. 'pc-read-memp' returns **t** if the next k consecutive bytes are ROM.

DEFINITION: pc-byte-readp $(x,\ mem)$ = pc-readp $(x,\ \texttt{32},\ \text{car}\,(mem))$

DEFINITION:

```
pc-read-memp (x, mem, k)
   =
if k ≃ 0  then t
else pc-byte-readp (add (32, x, k − 1), mem)
          ∧
     pc-read-memp (x, mem, k − 1) endif
```

DEFINITION: pc-word-readp $(x,\ mem)$ = pc-read-memp $(x,\ mem,\ \text{WSZ})$

DEFINITION: pc-long-readp $(x,\ mem)$ = pc-read-memp $(x,\ mem,\ \text{LSZ})$

Read from the memory. 'byte-read' reads a byte from the memory.

DEFINITION: byte-read $(x,\ mem)$ = head (read $(x,\ \texttt{32},\ \text{cdr}\,(mem)),\ \text{B})$

Read k consecutive bytes from the memory at x to form a natural number. 'read-mem' is a function of three arguments, x, mem, and k. 'read-mem' returns the natural number obtained by 'appending' together the n bytes that are obtained by reading from mem at locations $addr$, ..., $addr + n - 1$. The most significant byte is the one with the lowest memory address, and conversely, the least significant byte is the one with the highest memory address. This is known as the 'Big Endian' scheme of memory.

DEFINITION:
read-mem (x, mem, k)
  =
if $k \simeq$ 0 **then** 0
else app (B, byte-read (add (32, x, $k - 1$), mem), read-mem (x, mem, $k - 1$)) **endif**

The two functions 'word-read' and 'long-read' use the function 'read-mem' to obtain a word or a long word from the memory.

DEFINITION: word-read (x, mem) = read-mem (x, mem, WSZ)

DEFINITION: long-read (x, mem) = read-mem (x, mem, LSZ)

Fetch instructions, by fetching bytes pointed to by the pc. This is the same as reading from memory. But we define a separate set of functions because we use them in a very different sense in our specification. 'pc-byte-read' reads a byte from the memory at pc.

DEFINITION:
pc-byte-read (pc, mem) = head (pc-read (pc, 32, cdr (mem)), B)

DEFINITION:
pc-read-mem (pc, mem, k)
  =
```
if $k \simeq$ 0 then 0
else app (B,
            pc-byte-read (add (32, $pc$, $k - 1$), $mem$),
            pc-read-mem ($pc$, $mem$, $k - 1$)) endif
```

'pc-word-read' reads a word from the memory at pc.

DEFINITION: pc-word-read (pc, mem) = pc-read-mem (pc, mem, WSZ)

'pc-long-read' reads a longword from the memory at pc.

DEFINITION: pc-long-read (pc, mem) = pc-read-mem (pc, mem, LSZ)

We define some bit field extractors. The function names reflect the meanings of the fields for MC68020 instructions.

The source register field. 's_rn' is a function of one argument, ins, which should be a word, i.e., a 16-bit bit-vector. Nonnegative integer value of bits 0..2 of ins.

DEFINITION: $\text{s\_rn}(ins) = \text{bits}(ins, \texttt{0}, \texttt{2})$

The source mode field. Integer value of bits 3..5 of ins.

DEFINITION: $\text{s\_mode}(ins) = \text{bits}(ins, \texttt{3}, \texttt{5})$

The destination mode field. Integer value of bits 6..8 of ins.

DEFINITION: $\text{d\_mode}(ins) = \text{bits}(ins, \texttt{6}, \texttt{8})$

The destination register field. Integer value of bits 9..11 of ins.

DEFINITION: $\text{d\_rn}(ins) = \text{bits}(ins, \texttt{9}, \texttt{11})$

The op-mode field. Integer value of bits 6..8 of ins.

DEFINITION: $\text{opmode-field}(ins) = \text{bits}(ins, \texttt{6}, \texttt{8})$

The condition field. Integer value of bits 8..11 of ins.

DEFINITION: $\text{cond-field}(ins) = \text{bits}(ins, \texttt{8}, \texttt{11})$

By the "oplen" of an instruction we mean whether an instruction deals with a byte, word, long word, or quad word operation.

The oplen of the operation is normally determined by bits 6 and 7. 'op-len' is a function of one argument, *ins*, which normally is the first word of an instruction.

67	(common bit numbers)
00	byte
10	word
01	long word
11	illegal, but we return (qsz).

DEFINITION: $\text{op-len}(ins) = (\texttt{B} * \exp(\texttt{2}, \text{bits}(ins, \texttt{6}, \texttt{7})))$

9 Storing the Result

'byte-writep' determines whether the location x is writable with respect to the current memory.

DEFINITION: $\text{byte-writep}(x, mem) = \text{writep}(x, \texttt{32}, \text{car}(mem))$

'write-memp' determines whether the k consecutive bytes starting at address x in the memory are writable.

DEFINITION:
$\text{write-memp}(x, mem, k)$
$=$
if $k \simeq \texttt{0}$ **then t**
else $\text{byte-writep}(\text{add}(\texttt{32}, x, k-1), mem) \wedge \text{write-memp}(x, mem, k-1)$ **endif**

'write-mem' is a function of four arguments, *value*, x, *mem*, and k. *value* should be a natural number, namely the thing we are storing; x should be a natural number, namely the address at which to store *value*; *mem* is the memory; k is the number of bytes to store. We store the bytes one byte at a time, storing the most significant byte of *value* first, at location x, and storing subsequently, decreasingly significant bytes at increasing addresses.

DEFINITION:
byte-write (*value*, x, *mem*)
  =
cons (car (*mem*), write (head (*value*, `B`), x, `32`, cdr (*mem*)))

DEFINITION:
```
write-mem(value, x, mem, k)
  =
if k ≃ 0 then mem
else write-mem(tail(value, B),
               x,
               byte-write(value, add(32, x, k − 1), mem),
               k − 1) endif
```

Obtain c, v, z, n, and x from CCR. The following five functions 'ccr-c', 'ccr-v', 'ccr-z', 'ccr-n', and 'ccr-x' simply access the five correspondingly named bits of the CCR. We use them to specify the condition cc in the bcc instruction.

DEFINITION: ccr-c (*ccr*) = bitn (*ccr*, `0`)

DEFINITION: ccr-v (*ccr*) = bitn (*ccr*, `1`)

DEFINITION: ccr-z (*ccr*) = bitn (*ccr*, `2`)

DEFINITION: ccr-n (*ccr*) = bitn (*ccr*, `3`)

DEFINITION: ccr-x (*ccr*) = bitn (*ccr*, `4`)

Whenever instructions update the CCR, 'cvznx' simply generates a new partial CCR consisting of the new cvznx-flags.

DEFINITION:
```
cvznx(c, v, z, n, x)
  =
(fix-bit(c)
    +
 ((2 * fix-bit(v))
     +
  ((4 * fix-bit(z)) + ((8 * fix-bit(n)) + (16 * fix-bit(x))))))
```

'set-cvznx' replaces the old flags in CCR by the given flags.

DEFINITION: set-cvznx ($cvznx$, ccr) = replace (`5`, $cvznx$, ccr)

DEFINITION:
set-c (c, ccr)
=
set-cvznx (cvznx (c, ccr-v (ccr), ccr-z (ccr), ccr-n (ccr), ccr-x (ccr)), ccr)

DEFINITION:
set-v (v, ccr)
=
set-cvznx (cvznx (ccr-c (ccr), v, ccr-z (ccr), ccr-n (ccr), ccr-x (ccr)), ccr)

DEFINITION:
set-z (z, ccr)
=
set-cvznx (cvznx (ccr-c (ccr), ccr-v (ccr), z, ccr-n (ccr), ccr-x (ccr)), ccr)

DEFINITION:
set-n (n, ccr)
=
set-cvznx (cvznx (ccr-c (ccr), ccr-v (ccr), ccr-z (ccr), n, ccr-x (ccr)), ccr)

DEFINITION:
set-x (x, ccr)
=
set-cvznx (cvznx (ccr-c (ccr), ccr-v (ccr), ccr-z (ccr), ccr-n (ccr), x), ccr)

To halt the machine, we simply put the halting reason "signal" in the machine state.

DEFINITION:
halt ($signal$, s)
=
mc-state ($signal$, mc-rfile (s), mc-pc (s), mc-ccr (s), mc-mem (s))

To update the register file in the state s.

DEFINITION:
update-rfile (*new-rfile*, s)
=
mc-state (mc-status (s), *new-rfile*, mc-pc (s), mc-ccr (s), mc-mem (s))

To update the program counter in the state s.

DEFINITION:
update-pc ($new\text{-}pc$, s)
=
mc-state (mc-status (s), mc-rfile (s), $new\text{-}pc$, mc-ccr (s), mc-mem (s))

To update the condition code in the state s.

DEFINITION:
update-ccr ($new\text{-}ccr$, s)
=
mc-state (mc-status (s),
    mc-rfile (s),
    mc-pc (s),
    set-cvznx ($new\text{-}ccr$, mc-ccr (s)),
    mc-mem (s))

To update the memory in the state s.

DEFINITION:
update-mem ($new\text{-}mem$, s)
=
mc-state (mc-status (s), mc-rfile (s), mc-pc (s), mc-ccr (s), $new\text{-}mem$)

'read-dn' and 'read-an' are used to fetch data and address registers in the machine state s.

DEFINITION: read-dn ($oplen$, dn, s) = read-rn ($oplen$, dn, mc-rfile (s))

DEFINITION:
read-an ($oplen$, an, s) = read-rn ($oplen$, 8 + an, mc-rfile (s))

'write-dn' and 'write-an' are used to modify data and address registers in the machine state s. They return the modified machine state.

DEFINITION:
write-dn ($oplen$, $value$, dn, s)
=
update-rfile (write-rn ($oplen$, $value$, dn, mc-rfile (s)), s)

DEFINITION:
write-an ($oplen$, $value$, an, s)
=
update-rfile (write-rn ($oplen$, $value$, 8 + an, mc-rfile (s)), s)

'sp' is the constant 7, which refers to the stack pointer sp(a7) in the address register file.

DEFINITION: SP = 7

'read-sp' is a function that fetches the stack pointer in the given state s.

DEFINITION: read-sp (s) = read-an (L, SP, s)

'write-sp' is a function of two arguments, *value* and *s*. It returns a new machine state with the stack pointer updated to value.

DEFINITION: write-sp (*value*, s) = write-an (L, *value*, SP, s)

'push-up' pushes *value* onto the sp stack and increments sp.

DEFINITION:
push-sp (*opsz*, *value*, s)
  =
let *sp* **be** sub (L, *opsz*, read-sp (s))
in
if write-memp (*sp*, mc-mem (s), *opsz*)
then update-mem (write-mem (*value*, *sp*, mc-mem (s), *opsz*), write-sp (*sp*, s))
else halt (WRITE-SIGNAL, s) **endif endlet**

10 Retrieving the Operand According to Oplen

The function 'operand' returns the operand based on the given addr. *addr* should be a cons; the 'car' tells us where to retrieve the operand, the 'cdr' provides the real address.

DEFINITION:
operand (*oplen*, *addr*, s)
  =
if car (*addr*) = `'d` **then** read-dn (*oplen*, cdr (*addr*), s)
elseif car (*addr*) = `'a` **then** read-an (*oplen*, cdr (*addr*), s)
elseif car (*addr*) = `'m` **then** read-mem (cdr (*addr*), mc-mem (s), op-sz (*oplen*))
else cdr (*addr*) **endif**

11 Effective Address Calculation

We now begin the definition of a collection of functions culminating in the function 'effec-addr', which computes "the effective address" for MC68020 instructions. (Actually, some instructions, e.g., the MOVE instruction, compute two effective addresses.)

In his Ph. D. thesis, Warren Hunt specified the FM8502 microprocessor in the Nqthm logic [1]. In Hunt's FM8502 there is only one instruction format. Therefore in the FM8502 "soft-machine" specification one can compute the effective addresses before looking at the op-code. But in the MC68020, there are several instruction formats, and the algorithm for computing effective addresses depends upon what the op-code is. So we cannot handle instructions as uniformly as in FM8502. We have to know what the op-code is at a very early stage in the implementation.

Pre-effect and post-effect are two functions used in address register predecrement and postincrement.

DEFINITION:
post-effect (*oplen*, *rn*, *addr*)
  =
if (*rn* = SP) $\wedge$ (*oplen* = B) **then** add (L, *addr*, WSZ)
else add (L, *addr*, op-sz (*oplen*)) **endif**

DEFINITION:
pre-effect (*oplen*, *rn*, *addr*)
  =
if (*rn* = SP) $\wedge$ (*oplen* = B) **then** sub (L, WSZ, *addr*)
else sub (L, op-sz (*oplen*), *addr*) **endif**

For each of the different effective addressing modes, we define a function that "does the work." In each case, the function takes as its argument the current value of the state, *s*. Some may take other parameters. In each case a 'cons' is returned, consisting of (a) an internal state with possible an and pc updates after the effective address calculation; (b) the effective address, normally another cons indicating where to look and where to get the operands.

Register direct modes. Data register direct (000) and address register direct (001). Number of extension words: 0.

'dn-direct' is a function of two arguments, *rn* and *s*. *rn* should be a natural number and *s* should be an mc-state. Mode 000.

DEFINITION: dn-direct (*rn*, *s*) = cons (*s*, cons (`'d`, *rn*))

'an-direct' is a function of two arguments, *rn* and *s*. *rn* should be a natural number and *s* should be an mc-state. Mode 001.

DEFINITION: an-direct (*rn*, *s*) = cons (*s*, cons (`'a`, *rn*))

Memory address modes. The pc argument to these effective address subroutines need not be the actual pc of the instruction. In the case of the MOVE instruction, which involves two effective address calculations, the pc will point to the word before the "next" possible byte in the memory which is to be used as an extension word. For example, the instruction

i: move (1,a0) (2,a2)

i.e., move the word at 1 + (a0) to 2 + (a2), requires altogether 3 words because two extension words are required, one for each of the displacements (1 and 2). When we invoke the function ‘addr-disp’ for the calculation of the first effective address, the pc will be i. But when we again invoke the function ‘addr-disp’ for the calculation of the second effective address, the pc will be i+2.

A subtlety about pc displacement. The one MC68020 instruction that involves two effective address calculations, the MOVE instruction, will have its second effective address calculation performed by us with the pc not pointing necessarily to the MOVE instruction but rather (possibly) pointing to the next word after the calculation of the first effective address. However, this discrepancy does not cause a problem with pc relative addressing because pc relative addressing is prohibited in the second effective address calculation.

Address register indirect, mode 010. Number of extension words: 0. ‘addr-indirect’ is a function of two arguments, *rn* and *s*. *rn* should be a natural number and *s* should be a machine state. It returns the contents of the *rn* element of the address register file.

DEFINITION:
addr-indirect (*rn*, *s*) = cons (*s*, cons (`'m`, read-an (L, *rn*, *s*)))

Address register indirect with postincrement, mode 011. Number of extension words: 0.

DEFINITION:
addr-postinc (*oplen*, *rn*, *s*)
=
let *addr* **be** read-an (L, *rn*, *s*)
in
cons (write-an (L, post-effect (*oplen*, *rn*, *addr*), *rn*, *s*), cons (`'m`, *addr*)) **endlet**

Address register indirect with predecrement, mode 100. Number of extension words: 0. The function ‘addr-predec’ returns a cons of the given state *s* and the contents of the *rn* element of the register file after the register has been predecremented.

DEFINITION:
addr-predec (*oplen*, *rn*, *s*)
=
let *addr* **be** read-an (L, *rn*, *s*)
in
cons (write-an (L, pre-effect (*oplen*, *rn*, *addr*), *rn*, *s*),
cons (`'m`, pre-effect (*oplen*, *rn*, *addr*))) **endlet**

Address register indirect with index, mode 101. Number of extension words: 1. We now begin handling an effective address calculation which involves an extension word. In this mode, we add in the sign-extended 16-bit quantity in the word after the pc. We return a cons with (a) the state with pc incremented and (b) the sum of the address register *rn* and the sign-extended contents of the next word.

```
DEFINITION:
addr-disp (pc, rn, s)
   =
if pc-word-readp (pc, mc-mem (s))
then cons (update-pc (add (L, pc, WSZ), s),
           cons ('m,
                 add (L, read-an (L, rn, s), ext (W, pc-word-read (pc, mc-mem (s)), L))))
else cons (halt (PC-SIGNAL, s), nil) endif
```

Address register indirect with index (8-bit displacement), mode 110. Number of extension words: 1.

```
DEFINITION:  index-rn (indexwd) = bits (indexwd, 12, 14)
```

```
DEFINITION:
index-register (indexwd, s)
   =
if b0p (bitn (indexwd, 15))
then if b0p (bitn (indexwd, 11))  then ext (W, read-dn (W, index-rn (indexwd), s), L)
     else read-dn (L, index-rn (indexwd), s) endif
elseif b0p (bitn (indexwd, 11))  then ext (W, read-an (W, index-rn (indexwd), s), L)
else read-an (L, index-rn (indexwd), s) endif
```

```
DEFINITION:
ir-scaled (indexwd, s)
   =
asl (L, index-register (indexwd, s), bits (indexwd, 9, 10))
```

```
DEFINITION:
addr-index-disp (pc, rn, indexwd, s)
   =
cons (update-pc (pc, s),
      cons ('m,
            add (L,
                 add (L, read-an (L, rn, s), ext (B, head (indexwd, B), L)),
                 ir-scaled (indexwd, s))))
```

Address register indirect with index (base displacement), mode 110. Number of extension words: 1, 2, or 3.

```
DEFINITION:
addr-index-bd (pc, addr, indexwd, s)
    =
cons (update-pc (pc, s), cons ('m, add (L, addr, ir-scaled (indexwd, s))))
```

Memory indirect without index, mode 110. Number of extension words: 1, 2, 3, 4, or 5.

```
DEFINITION:
mem-indirect (pc, addr, olen, s)
    =
if long-readp (addr, mc-mem (s))
then if pc-read-memp (pc, mc-mem (s), op-sz (olen))
       then cons (update-pc (add (L, pc, op-sz (olen)), s),
                  cons ('m,
                        add (L,
                             long-read (addr, mc-mem (s)),
                             ext (olen, pc-read-mem (pc, mc-mem (s), op-sz (olen)), L))))
       else cons (halt (PC-SIGNAL, s), nil) endif
else cons (halt (READ-SIGNAL, s), nil) endif
```

Memory indirect postindexed mode.

```
DEFINITION:
mem-postindex (pc, addr, indexwd, olen, s)
    =
if long-readp (addr, mc-mem (s))
then if pc-read-memp (pc, mc-mem (s), op-sz (olen))
       then cons (update-pc (add (L, pc, op-sz (olen)), s),
                  cons ('m,
                        add (L,
                             add (L, long-read (addr, mc-mem (s)), ir-scaled (indexwd, s)),
                             ext (olen, pc-read-mem (pc, mc-mem (s), op-sz (olen)), L))))
       else cons (halt (PC-SIGNAL, s), nil) endif
else cons (halt (READ-SIGNAL, s), nil) endif
```

Memory indirect preindexed mode.

```
DEFINITION:
mem-preindex (pc, addr, indexwd, olen, s)
    =
mem-indirect (pc, add (L, addr, ir-scaled (indexwd, s)), olen, s)
```

```
DEFINITION:  i-is (indexwd) = bits (indexwd, 0, 2)
```

The base displacement has been added to *addr*, if necessary. 'addr-index3' is to consider the index register and index/indirect selection.

DEFINITION:

```
addr-index3 (pc, addr, indexwd, s)
  =
if b0p (bitn (indexwd, 6))
then if i-is (indexwd) < 4
     then if i-is (indexwd) < 2
          then if i-is (indexwd) = 0 then addr-index-bd (pc, addr, indexwd, s)
               else mem-preindex (pc, addr, indexwd, 0, s) endif
          elseif i-is (indexwd) = 2 then mem-preindex (pc, addr, indexwd, W, s)
          else mem-preindex (pc, addr, indexwd, L, s) endif
     elseif i-is (indexwd) < 6
     then if i-is (indexwd) = 4 then cons (halt (RESERVED-SIGNAL, s), nil)
          else mem-postindex (pc, addr, indexwd, 0, s) endif
     elseif i-is (indexwd) = 6 then mem-postindex (pc, addr, indexwd, W, s)
     else mem-postindex (pc, addr, indexwd, L, s) endif
elseif i-is (indexwd) < 4
then if i-is (indexwd) < 2
     then if i-is (indexwd) = 0 then cons (update-pc (pc, s), cons ('m, addr))
          else mem-indirect (pc, addr, 0, s) endif
     elseif i-is (indexwd) = 2 then mem-indirect (pc, addr, W, s)
     else mem-indirect (pc, addr, L, s) endif
else cons (halt (RESERVED-SIGNAL, s), nil) endif
```

DEFINITION: bd-sz (*indexwd*) = bits (*indexwd*, 4, 5)

The address register (base register) has been added to addr, if necessary. 'addr-index2' is to consider the base displacement.

DEFINITION:

```
addr-index2 (pc, addr, indexwd, s)
  =
if bd-sz (indexwd) < 2
then if bd-sz (indexwd) = 0 then cons (halt (RESERVED-SIGNAL, s), nil)
     else addr-index3 (pc, addr, indexwd, s) endif
elseif bd-sz (indexwd) = 2
then if pc-word-readp (pc, mc-mem (s))
     then addr-index3 (add (L, pc, WSZ),
                       add (L, addr, ext (W, pc-word-read (pc, mc-mem (s)), L)),
                       indexwd,
                       s)
     else cons (halt (PC-SIGNAL, s), nil) endif
elseif pc-long-readp (pc, mc-mem (s))
then addr-index3 (add (L, pc, LSZ),
                  add (L, addr, pc-long-read (pc, mc-mem (s))),
                  indexwd,
                  s)
else cons (halt (PC-SIGNAL, s), nil) endif
```

DEFINITION:

```
bs-register (rn, indexwd, s)
  =
if b0p (bitn (indexwd, 7))  then read-an (LSZ, rn, s)
else 0 endif
```

'addr-index1' is to consider the address register (base register).

DEFINITION:

```
addr-index1 (pc, rn, indexwd, s)
  =
if b0p (bitn (indexwd, 8))  then addr-index-disp (pc, rn, indexwd, s)
elseif b0p (bitn (indexwd, 3))
then addr-index2 (pc, bs-register (rn, indexwd, s), indexwd, s)
else cons (halt (RESERVED-SIGNAL, s), nil) endif
```

DEFINITION:

```
addr-index (pc, rn, s)
  =
if pc-word-readp (pc, mc-mem (s))
then addr-index1 (add (L, pc, WSZ), rn, pc-word-read (pc, mc-mem (s)), s)
else cons (halt (PC-SIGNAL, s), nil) endif
```

Absolute short address. Mode 111, rn 000.

DEFINITION:

```
absolute-short (pc, s)
  =
if pc-word-readp (pc, mc-mem (s))
then cons (update-pc (add (L, pc, WSZ), s),
           cons ('m, ext (W, pc-word-read (pc, mc-mem(s)), L)))
else cons (halt (PC-SIGNAL, s), nil) endif
```

Absolute long address. Mode 111, rn 001.

DEFINITION:

```
absolute-long (pc, s)
  =
if pc-long-readp (pc, mc-mem (s))
then cons (update-pc (add (L, pc, LSZ), s), cons ('m, pc-long-read (pc, mc-mem (s))))
else cons (halt (PC-SIGNAL, s), nil) endif
```

Surprisingly, the design of the MC68020 deliberately avoids having two program counter addressing modes. This specification here relies on this very fact.

Program counter indirect with displacement. Mode 111, rn 010. Number of extension words: 1.

DEFINITION:
```
pc-disp (pc, s)
   =
if pc-word-readp (pc, mc-mem (s))
then cons (update-pc (add (L, pc, WSZ), s),
              cons ('m, add (L, pc, ext (W, pc-word-read (pc, mc-mem (s)), L))))
else cons (halt (PC-SIGNAL, s), nil) endif
```

Program counter indirect with index (8-bit displacement). mode 111, rn 011.

DEFINITION:
```
pc-index-disp (pc, indexwd, s)
   =
cons (update-pc (add (L, pc, WSZ), s),
        cons ('m, add (L, add (L, pc, ext (B, head (indexwd, B), L)), ir-scaled (indexwd, s))))
```

Program counter indirect with index (base displacement) mode.
Program counter memory indirect postindexed mode.
Program counter memory indirect preindexed mode.

DEFINITION:
```
bs-pc (pc, indexwd)
   =
if b0p (bitn (indexwd, 7))  then pc
else 0 endif
```

DEFINITION:
```
pc-index1 (pc, indexwd, s)
   =
if b0p (bitn (indexwd, 8))  then pc-index-disp (pc, indexwd, s)
elseif b0p (bitn (indexwd, 3))
then addr-index2 (add (L, pc, WSZ), bs-pc (pc, indexwd), indexwd, s)
else cons (halt (RESERVED-SIGNAL, s), nil) endif
```

DEFINITION:
```
pc-index (pc, s)
   =
if pc-word-readp (pc, mc-mem (s))
then pc-index1 (pc, pc-word-read (pc, mc-mem (s)), s)
else cons (halt (PC-SIGNAL, s), nil) endif
```

Immediate data. Mode 111, rn 100. Number of extension words: 1 or 2.

```
DEFINITION:
immediate(oplen, pc, s)
   =
if oplen = B
then if pc-word-readp (pc, mc-mem(s))
        then cons (update-pc (add (L, pc, WSZ), s),
                   cons ('i, pc-byte-read (add (L, pc, BSZ), mc-mem(s))))
        else cons (halt (PC-SIGNAL, s), nil) endif
elseif pc-read-memp (pc, mc-mem(s), op-sz (oplen))
then cons (update-pc (add (L, pc, op-sz (oplen)), s),
           cons ('i, pc-read-mem (pc, mc-mem(s), op-sz (oplen))))
else cons (halt (PC-SIGNAL, s), nil) endif
```

Effective address calculation. 'effec-addr' is a function of four arguments, *oplen*, *mode*, *rn*, and *s*. 'oplen' should be B, W, or L; it is the size of the datum we are computing the effective address of. *mode* is a natural number extracted from the first word of the instruction; *mode* indicates pre-decrement, post-increment, etc. *rn* is a natural number extracted from the first word of the instruction; *rn* designates a register. *s* the current machine state. 'effec-addr' returns a pair, or 'cons' as it is called in Lisp and Nqthm. The first element (or 'car') of this pair is an internal state after this effective address calculation. The second element (or cdr) is another 'cons' consisting of the direction ('d, 'a, 'm, or 'i), and the effective address (a nonnegative integer). Because MC68020 instructions can be as many as 11 words long, the calculation of the next pc is intimately tied to the effective address calculation.

```
DEFINITION:
effec-addr (oplen, mode, rn, s)
   =
if mode < 4
then if mode < 2
        then if mode = 0 then dn-direct (rn, s)
                else an-direct (rn, s) endif
        elseif mode = 2 then addr-indirect (rn, s)
        else addr-postinc (oplen, rn, s) endif
elseif mode < 6
then if mode = 4 then addr-predec (oplen, rn, s)
        else addr-disp (mc-pc (s), rn, s) endif
elseif mode = 6 then addr-index (mc-pc (s), rn, s)
elseif rn < 4
then if rn < 2
        then if rn = 0 then absolute-short (mc-pc (s), s)
                else absolute-long (mc-pc (s), s) endif
```

```
    elseif $rn = 2$ then pc-disp (mc-pc ($s$), $s$)
    else pc-index (mc-pc ($s$), $s$) endif
else immediate($oplen$, mc-pc ($s$), $s$) endif
```

Given an effective address field, test if it is one of the existing addressing modes.

DEFINITION:
addr-modep ($mode$, rn)
  =
if $mode = 7$ **then** $rn \leq 4$
else t endif

Given an effective address field, test if it is a data addressing mode.

DEFINITION:
data-addr-modep ($mode$, rn)
  =
if $mode = 7$ **then** $rn \leq 4$
else $mode \neq 1$ **endif**

Given an effective address field, test if it is a memory addressing mode.

DEFINITION:
memory-addr-modep ($mode$, rn)
  =
if $mode = 7$ **then** $rn \leq 4$
else $mode \geq 2$ **endif**

Given an effective address field, test if it is a control addressing mode.

DEFINITION:
control-addr-modep ($mode$, rn)
  =
if $mode = 7$ **then** $rn \leq 3$
else $(mode = 2) \vee (mode \geq 5)$ **endif**

Given an effective address field, test if it is an alterable addressing mode.

DEFINITION:
alterable-addr-modep ($mode$, rn)
  =
$((mode \neq 7) \vee (rn = 0) \vee (rn = 1))$

'dn-direct-modep' returns **t** if the addressing mode is a data register direct. Returns **f** otherwise.

DEFINITION: dn-direct-modep ($mode$) = $(mode = 0)$

'an-direct-modep' returns **t** if the addressing mode is an address register direct, and returns **f** otherwise.

DEFINITION: an-direct-modep (*mode*) = (*mode* = `1`)

Postincrement.

DEFINITION: postinc-modep (*mode*) = (*mode* = `3`)

Predecrement.

DEFINITION: predec-modep (*mode*) = (*mode* = `4`)

DEFINITION:
idata-modep (*mode*, *rn*) = ((*mode* = `7`) $\wedge$ (*rn* = `4`))

In address register direct (001), a byte size operation is not allowed.

DEFINITION:
byte-an-direct-modep (*oplen*, *mode*)
    =
((*oplen* = B) $\wedge$ an-direct-modep (*mode*))

An internal state in the execution of one instruction.

DEFINITION:
```
mc-instate (oplen, ins, s)
    =
let s&addr  be  effec-addr (oplen, s_mode (ins), s_rn (ins), s)
in
if cadr (s&addr) = 'm
then if read-memp (cddr (s&addr), mc-mem (s), op-sz (oplen))  then s&addr
        else cons (halt (READ-SIGNAL, s), nil) endif
else s&addr endif endlet
```

Mapping functions. 'mapping' finishes the execution of instructions. 'mapping' maps a machine state into the next state.

DEFINITION:
```
d-mapping (oplen, v&cvznx, addr, s)
    =
mc-state (mc-status (s),
            write-rn (oplen, car (v&cvznx), addr, mc-rfile (s)),
            mc-pc (s),
            set-cvznx (cdr (v&cvznx), mc-ccr (s)),
            mc-mem (s))
```

```
DEFINITION:
a-mapping(oplen, v&cvznx, addr, s)
  =
mc-state(mc-status(s),
         write-rn(oplen, car(v&cvznx), 8 + addr, mc-rfile(s)),
         mc-pc(s),
         set-cvznx(cdr(v&cvznx), mc-ccr(s)),
         mc-mem(s))
```

```
DEFINITION:
m-mapping(oplen, v&cvznx, addr, s)
  =
if write-memp(addr, mc-mem(s), op-sz(oplen))
then mc-state(mc-status(s),
              mc-rfile(s),
              mc-pc(s),
              set-cvznx(cdr(v&cvznx), mc-ccr(s)),
              write-mem(car(v&cvznx), addr, mc-mem(s), op-sz(oplen)))
else halt(WRITE-SIGNAL, s) endif
```

```
DEFINITION:
mapping(oplen, v&cvznx, s&addr)
  =
if cadr(s&addr) = 'd
then d-mapping(oplen, v&cvznx, cddr(s&addr), car(s&addr))
elseif cadr(s&addr) = 'a
then a-mapping(oplen, v&cvznx, cddr(s&addr), car(s&addr))
else m-mapping(oplen, v&cvznx, cddr(s&addr), car(s&addr)) endif
```

12 The Individual Instructions

ADD instruction. The computation of the condition code register(CCR).

```
DEFINITION:
add-c(n, sopd, dopd)
  =
let result be add(n, sopd, dopd)
in
b-or(b-or(b-and(bitn(sopd, n - 1), bitn(dopd, n - 1)),
          b-and(b-not(bitn(result, n - 1)), bitn(dopd, n - 1))),
     b-and(bitn(sopd, n - 1), b-not(bitn(result, n - 1)))) endlet
```

```
DEFINITION:
add-v(n, sopd, dopd)
```

```
    =
let result be add(n, sopd, dopd)
in
b-or(b-and(b-and(bitn(sopd, n − 1), bitn(dopd, n − 1)),
              b-not(bitn(result, n − 1))),
      b-and(b-and(b-not(bitn(sopd, n − 1)), b-not(bitn(dopd, n − 1))),
              bitn(result, n − 1))) endlet
```

DEFINITION:
add-z(*oplen*, *sopd*, *dopd*)

```
    =
if add(oplen, dopd, sopd) = 0 then B1
else B0 endif
```

DEFINITION:
add-n(*oplen*, *sopd*, *dopd*)

```
    =
if add(oplen, dopd, sopd) < exp(2, oplen − 1) then B0
else B1 endif
```

DEFINITION:
add-cvznx(*oplen*, *sopd*, *dopd*)

```
    =
cvznx(add-c(oplen, sopd, dopd),
        add-v(oplen, sopd, dopd),
        add-z(oplen, sopd, dopd),
        add-n(oplen, sopd, dopd),
        add-c(oplen, sopd, dopd))
```

The effects of the execution of an ADD instruction are given as follows.

DEFINITION:
add-effect(*oplen*, *sopd*, *dopd*)

```
    =
cons(add(oplen, dopd, sopd), add-cvznx(oplen, sopd, dopd))
```

Test if the addressing mode is legal.

DEFINITION:
add-addr-modep1(*oplen*, *ins*)

```
    =
(addr-modep(s_mode(ins), s_rn(ins))
     ∧
 (¬ byte-an-direct-modep(oplen, s_mode(ins))))
```

DEFINITION:
```
add-addr-modep2 (ins)
    =
(alterable-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 memory-addr-modep (s_mode (ins), s_rn (ins)))
```

An execution of an ADD instruction.

DEFINITION:
```
add-ins1 (oplen, ins, s)
    =
if add-addr-modep1 (oplen, ins)
then let s&addr be mc-instate (oplen, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else d-mapping (oplen,
                           add-effect (oplen,
                                       operand (oplen, cdr (s&addr), s),
                                       read-dn (oplen, d_rn (ins), s)),
                           d_rn (ins),
                           car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

DEFINITION:
```
add-mapping (opd, oplen, ins, s)
    =
let s&addr be mc-instate (oplen, ins, s)
in
if mc-haltp (car (s&addr)) then car (s&addr)
else mapping (oplen,
                add-effect (oplen, opd, operand (oplen, cdr (s&addr), s)),
                s&addr) endif endlet
```

DEFINITION:
```
add-ins2 (oplen, ins, s)
    =
if add-addr-modep2 (ins)
then add-mapping (read-dn (oplen, d_rn (ins), s), oplen, ins, s)
else halt (MODE-SIGNAL, s) endif
```

ADDA instruction.

DEFINITION:
```
adda-addr-modep (ins) = addr-modep (s_mode (ins), s_rn (ins))
```

Notice that the ADDA instruction does not affect CCR.

DEFINITION:
```
adda-ins (oplen, ins, s)
    =
if adda-addr-modep (ins)
then let s&addr  be  mc-instate (oplen, ins, s)
        in
        if mc-haltp (car (s&addr))  then  car (s&addr)
        else write-an (L,
                           add (L,
                                  read-an (L, d_rn (ins), s),
                                  ext (oplen, operand (oplen, cdr (s&addr), s), L)),
                           d_rn (ins),
                           car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

ADDX instruction.

DEFINITION:
```
addx-c (n, x, sopd, dopd)
    =
let result  be  adder (n, x, sopd, dopd)
in
b-or (b-or (b-and (bitn (sopd, n − 1), bitn (dopd, n − 1)),
             b-and (b-not (bitn (result, n − 1)), bitn (dopd, n − 1))),
       b-and (bitn (sopd, n − 1), b-not (bitn (result, n − 1)))) endlet
```

DEFINITION:
```
addx-v (n, x, sopd, dopd)
    =
let result  be  adder (n, x, sopd, dopd)
in
b-or (b-and (b-and (bitn (sopd, n − 1), bitn (dopd, n − 1)),
               b-not (bitn (result, n − 1))),
       b-and (b-and (b-not (bitn (sopd, n − 1)), b-not (bitn (dopd, n − 1))),
               bitn (result, n − 1))) endlet
```

DEFINITION:
```
addx-z (oplen, z, x, sopd, dopd)
    =
b-and (z,
        if adder (oplen, x, dopd, sopd) = 0  then  B1
        else B0 endif)
```

```
DEFINITION:
addx-n (oplen, x, sopd, dopd)
   =
if adder (oplen, x, dopd, sopd) < exp (2, oplen - 1) then B0
else B1 endif

DEFINITION:
addx-cvznx (oplen, z, x, sopd, dopd)
   =
cvznx (addx-c (oplen, x, sopd, dopd),
       addx-v (oplen, x, sopd, dopd),
       addx-z (oplen, z, x, sopd, dopd),
       addx-n (oplen, x, sopd, dopd),
       addx-c (oplen, x, sopd, dopd))

DEFINITION:
addx-effect (oplen, sopd, dopd, ccr)
   =
cons (adder (oplen, ccr-x (ccr), dopd, sopd),
      addx-cvznx (oplen, ccr-z (ccr), ccr-x (ccr), sopd, dopd))

DEFINITION:
addx-ins1 (oplen, ins, s)
   =
d-mapping (oplen,
           addx-effect (oplen,
                        read-dn (oplen, s_rn (ins), s),
                        read-dn (oplen, d_rn (ins), s),
                        mc-ccr (s)),
           d_rn (ins),
           s)

DEFINITION:
addx-ins2 (oplen, ins, s)
   =
let s&addr0 be addr-predec (oplen, s_rn (ins), s)
in
if read-memp (cddr (s&addr0), mc-mem (s), op-sz (oplen))
then let s&addr be addr-predec (oplen, d_rn (ins), car (s&addr0))
     in
     if read-memp (cddr (s&addr), mc-mem (s), op-sz (oplen))
     then mapping (oplen,
                   addx-effect (oplen,
                                operand (oplen,
                                         cdr (s&addr0),
                                    car (s&addr0)),
                              operand (oplen,
                                    cdr (s&addr),
                                    cdr (s&addr)),
                              mc-ccr (s)),
                   s&addr)
      else halt (READ-SIGNAL, s) endif endlet
else halt (READ-SIGNAL, s) endif endlet
```

Opcode 1101. The ADD instruction group includes three instructions ADD, ADDA, and ADDX.

DEFINITION:

```
add-group (opmode, ins, s)
   =
if opmode < 4
then if opmode = 3 then adda-ins (W, ins, s)
      else add-ins1 (op-len (ins), ins, s) endif
elseif opmode = 7 then adda-ins (L, ins, s)
elseif s_mode (ins) = 0 then addx-ins1 (op-len (ins), ins, s)
elseif s_mode (ins) = 1 then addx-ins2 (op-len (ins), ins, s)
else add-ins2 (op-len (ins), ins, s) endif
```

SUB instruction. The computation of the condition code register (ccr).

DEFINITION:

```
sub-c (n, sopd, dopd)
   =
let result be sub (n, sopd, dopd)
in
b-or (b-or (b-and (bitn (sopd, n - 1), b-not (bitn (dopd, n - 1))),
            b-and (bitn (result, n - 1), b-not (bitn (dopd, n - 1)))),
      b-and (bitn (sopd, n - 1), bitn (result, n - 1))) endlet
```

DEFINITION:

```
sub-v (n, sopd, dopd)
   =
let result be sub (n, sopd, dopd)
in
b-or (b-and (b-and (b-not (bitn (sopd, n - 1)), bitn (dopd, n - 1)),
             b-not (bitn (result, n - 1))),
      b-and (b-and (bitn (sopd, n - 1), b-not (bitn (dopd, n - 1))),
             bitn (result, n - 1))) endlet
```

DEFINITION:

```
sub-z (oplen, sopd, dopd)
```

$=$

if sub $(oplen,\ sopd,\ dopd) = 0$ **then** B1
else B0 **endif**

DEFINITION:
sub-n $(oplen,\ sopd,\ dopd)$

$=$

if sub $(oplen,\ sopd,\ dopd) <$ exp $(2,\ oplen - 1)$ **then** B0
else B1 **endif**

DEFINITION:
sub-cvznx $(oplen,\ sopd,\ dopd)$

$=$

cvznx (sub-c $(oplen,\ sopd,\ dopd)$,
sub-v $(oplen,\ sopd,\ dopd)$,
sub-z $(oplen,\ sopd,\ dopd)$,
sub-n $(oplen,\ sopd,\ dopd)$,
sub-c $(oplen,\ sopd,\ dopd)$)

The effect of an execution of a SUB instruction.

DEFINITION:
sub-effect $(oplen,\ sopd,\ dopd)$

$=$

cons (sub $(oplen,\ sopd,\ dopd)$, sub-cvznx $(oplen,\ sopd,\ dopd)$)

Test if the addressing mode is illegal.

DEFINITION:
sub-addr-modep1 $(oplen,\ ins)$

$=$

(addr-modep (s_mode (ins), s_rn (ins))
$\wedge$
($\neg$ byte-an-direct-modep $(oplen$, s_mode (ins))))

DEFINITION:
sub-addr-modep2 (ins)

$=$

(alterable-addr-modep (s_mode (ins), s_rn (ins))
$\wedge$
memory-addr-modep (s_mode (ins), s_rn (ins)))

The execution of the SUB instruction.

DEFINITION:
sub-ins1 $(oplen,\ ins,\ s)$

```
    =
if sub-addr-modep1 (oplen, ins)
then let s&addr be mc-instate (oplen, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else d-mapping (oplen,
                            sub-effect (oplen,
                                           operand (oplen, cdr (s&addr), s),
                                           read-dn (oplen, d_rn (ins), s)),
                            d_rn (ins),
                            car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

DEFINITION:

```
sub-mapping (opd, oplen, ins, s)
    =
let s&addr be mc-instate (oplen, ins, s)
in
if mc-haltp (car (s&addr)) then car (s&addr)
else mapping (oplen,
                  sub-effect (oplen, opd, operand (oplen, cdr (s&addr), s)),
                  s&addr) endif endlet
```

DEFINITION:

```
sub-ins2 (oplen, ins, s)
    =
if sub-addr-modep2 (ins)
then sub-mapping (read-dn (oplen, d_rn (ins), s), oplen, ins, s)
else halt (MODE-SIGNAL, s) endif
```

SUBA instruction.

DEFINITION:

suba-addr-modep (*ins*) = addr-modep (s_mode (*ins*), s_rn (*ins*))

DEFINITION:

```
suba-ins (oplen, ins, s)
    =
if suba-addr-modep (ins)
then let s&addr be mc-instate (oplen, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else write-an (L,
                          sub (L,
                                 ext (oplen, operand (oplen, cdr (s&addr), s), L),
                    read-an (L, d_rn (ins), s)),
                d_rn (ins),
                car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

SUBX instruction.

DEFINITION:

```
subx-c (n, x, sopd, dopd)
   =
let result be subtracter (n, x, sopd, dopd)
in
b-or (b-or (b-and (bitn (sopd, n - 1), b-not (bitn (dopd, n - 1))),
            b-and (bitn (result, n - 1), b-not (bitn (dopd, n - 1)))),
      b-and (bitn (sopd, n - 1), bitn (result, n - 1))) endlet
```

DEFINITION:

```
subx-v (n, x, sopd, dopd)
   =
let result be subtracter (n, x, sopd, dopd)
in
b-or (b-and (b-and (b-not (bitn (sopd, n - 1)), bitn (dopd, n - 1)),
             b-not (bitn (result, n - 1))),
      b-and (b-and (bitn (sopd, n - 1), b-not (bitn (dopd, n - 1))),
             bitn (result, n - 1))) endlet
```

DEFINITION:

```
subx-z (oplen, z, x, sopd, dopd)
   =
b-and (z,
       if subtracter (oplen, x, sopd, dopd) = 0 then B1
       else B0 endif)
```

DEFINITION:

```
subx-n (oplen, x, sopd, dopd)
   =
if subtracter (oplen, x, sopd, dopd) < exp (2, oplen - 1) then B0
else B1 endif
```

DEFINITION:

```
subx-cvznx (oplen, z, x, sopd, dopd)
   =
cvznx (subx-c (oplen, x, sopd, dopd),
       subx-v (oplen, x, sopd, dopd),
       subx-z (oplen, z, x, sopd, dopd),
       subx-n (oplen, x, sopd, dopd),
       subx-c (oplen, x, sopd, dopd))
```

DEFINITION:
```
subx-effect (oplen, sopd, dopd, ccr)
    =
cons (subtracter (oplen, ccr-x (ccr), sopd, dopd),
      subx-cvznx (oplen, ccr-z (ccr), ccr-x (ccr), sopd, dopd))
```

DEFINITION:
```
subx-ins1 (oplen, ins, s)
    =
d-mapping (oplen,
           subx-effect (oplen,
                        read-dn (oplen, s_rn (ins), s),
                        read-dn (oplen, d_rn (ins), s),
                        mc-ccr (s)),
           d_rn (ins),
           s)
```

DEFINITION:
```
subx-ins2 (oplen, ins, s)
    =
let s&addr0 be addr-predec (oplen, s_rn (ins), s)
in
if read-memp (cddr (s&addr0), mc-mem(s), op-sz (oplen))
then let s&addr be addr-predec (oplen, d_rn (ins), car (s&addr0))
     in
     if read-memp (cddr (s&addr), mc-mem(s), op-sz (oplen))
     then mapping (oplen,
                   subx-effect (oplen,
                                operand (oplen,
                                         cdr (s&addr0),
                                         car (s&addr0)),
                                operand (oplen,
                                         cdr (s&addr),
                                         car (s&addr)),
                                mc-ccr (s)),
                   s&addr)
     else halt (READ-SIGNAL, s) endif endlet
else halt (READ-SIGNAL, s) endif endlet
```

Opcode 1001. The SUB instruction group includes three instructions SUB, SUBA, and SUBX.

DEFINITION:
```
sub-group (opmode, ins, s)
    =
if $opmode < 4$
then if $opmode = 3$ then suba-ins (W, $ins$, $s$)
       else sub-ins1 (op-len ($ins$), $ins$, $s$) endif
elseif $opmode = 7$ then suba-ins (L, $ins$, $s$)
elseif s_mode ($ins$) = 0 then subx-ins1 (op-len ($ins$), $ins$, $s$)
elseif s_mode ($ins$) = 1 then subx-ins2 (op-len ($ins$), $ins$, $s$)
else sub-ins2 (op-len ($ins$), $ins$, $s$) endif
```

AND instruction. The computation of the condition code register(CCR).

DEFINITION:
and-z ($sopd$, $dopd$)
=
```
if logand ($sopd$, $dopd$) = 0 then B1
else B0 endif
```

DEFINITION:
and-n ($oplen$, $sopd$, $dopd$)
=
```
if ($sopd < \exp(2, oplen - 1)$) $\vee$ ($dopd < \exp(2, oplen - 1)$) then B0
else B1 endif
```

DEFINITION:
and-cvznx ($oplen$, $sopd$, $dopd$, ccr)
=
cvznx (B0, B0, and-z ($sopd$, $dopd$), and-n ($oplen$, $sopd$, $dopd$), ccr-x (ccr))

The effect of the execution of the AND instruction.

DEFINITION:
and-effect ($oplen$, $sopd$, $dopd$, ccr)
=
cons (logand ($sopd$, $dopd$), and-cvznx ($oplen$, $sopd$, $dopd$, ccr))

Test if the addressing mode is legal.

DEFINITION:
and-addr-modep1 (ins) = data-addr-modep (s_mode (ins), s_rn (ins))

DEFINITION:
and-addr-modep2 (ins)
=
```
(alterable-addr-modep (s_mode ($ins$), s_rn ($ins$))
   $\wedge$
 memory-addr-modep (s_mode ($ins$), s_rn ($ins$)))
```

The execution of the AND instruction.

DEFINITION:

```
and-ins1 ($oplen$, $ins$, $s$)
    =
if and-addr-modep1 ($ins$)
then let $s\&addr$ be mc-instate ($oplen$, $ins$, $s$)
       in
       if mc-haltp (car ($s\&addr$)) then car ($s\&addr$)
       else d-mapping ($oplen$,
                        and-effect ($oplen$,
                                     operand ($oplen$, cdr ($s\&addr$), $s$),
                                     read-dn ($oplen$, d_rn ($ins$), $s$),
                                     mc-ccr ($s$)),
                        d_rn ($ins$),
                        car ($s\&addr$)) endif endlet
else halt (MODE-SIGNAL, $s$) endif
```

DEFINITION:

```
and-mapping ($sopd$, $oplen$, $ins$, $s$)
    =
let $s\&addr$ be mc-instate ($oplen$, $ins$, $s$)
in
if mc-haltp (car ($s\&addr$)) then car ($s\&addr$)
else mapping ($oplen$,
               and-effect ($oplen$,
                            $sopd$,
                            operand ($oplen$, cdr ($s\&addr$), $s$),
                            mc-ccr ($s$)),
               $s\&addr$) endif endlet
```

DEFINITION:

```
and-ins2 ($oplen$, $ins$, $s$)
    =
if and-addr-modep2 ($ins$)
then and-mapping (read-dn ($oplen$, d_rn ($ins$), $s$), $oplen$, $ins$, $s$)
else halt (MODE-SIGNAL, $s$) endif
```

MULU.W/MULS.W instruction. S * D $\rightarrow$ D. MULU expects x and y to be two natural numbers.

DEFINITION: mulu (n, x, y, i) = head ($x * y$, n)

MULS expects x and y to be two natural numbers.

DEFINITION:

```
muls ($n$, $x$, $y$, $i$)
    =
head (int-to-nat (itimes (nat-to-int ($x$, $i$), nat-to-int ($y$, $i$)), $2 * i$), $n$)
```

DEFINITION:
mul&div-addr-modep(ins) = data-addr-modep(s_mode(ins), s_rn(ins))

The condition codes for MULU.

```
DEFINITION:
mulu-v($n$, $sopd$, $dopd$, $i$)
    =
if ($sopd * dopd$) < exp(2, $n$) then B0
else B1 endif
```

```
DEFINITION:
mulu-z($n$, $sopd$, $dopd$, $i$)
    =
if mulu($n$, $sopd$, $dopd$, $i$) = 0 then B1
else B0 endif
```

```
DEFINITION:
mulu-n($n$, $sopd$, $dopd$, $i$)
    =
if mulu($n$, $sopd$, $dopd$, $i$) < exp(2, $n - 1$) then B0
else B1 endif
```

```
DEFINITION:
mulu-cvznx($n$, $sopd$, $dopd$, $i$, $ccr$)
    =
cvznx(B0,
        mulu-v($n$, $sopd$, $dopd$, $i$),
        mulu-z($n$, $sopd$, $dopd$, $i$),
        mulu-n($n$, $sopd$, $dopd$, $i$),
        ccr-x($ccr$))
```

```
DEFINITION:
mulu_w-ins($sopd$, $dn$, $s$)
    =
let $dopd$ be read-dn(W, $dn$, $s$)
in
update-ccr(mulu-cvznx(L, $sopd$, $dopd$, W, mc-ccr($s$)),
              write-dn(L, mulu(L, $sopd$, $dopd$, W), $dn$, $s$)) endlet
```

The condition codes for MULS.

```
DEFINITION:
muls-v($n$, $sopd$, $dopd$, $i$)
    =
if int-rangep(itimes(nat-to-int($sopd$, $i$), nat-to-int($dopd$, $i$)), $n$) then B0
else B1 endif
```

```
DEFINITION:
muls-z (n, sopd, dopd, i)
    =
if muls (n, sopd, dopd, i) = 0 then B1
else B0 endif

DEFINITION:
muls-n (n, sopd, dopd, i)
    =
if muls (n, sopd, dopd, i) < exp (2, n − 1) then B0
else B1 endif

DEFINITION:
muls-cvznx (n, sopd, dopd, i, ccr)
    =
cvznx (B0,
        muls-v (n, sopd, dopd, i),
        muls-z (n, sopd, dopd, i),
        muls-n (n, sopd, dopd, i),
        ccr-x (ccr))

DEFINITION:
muls_w-ins (sopd, dn, s)
    =
let dopd be read-dn (W, dn, s)
in
update-ccr (muls-cvznx (L, sopd, dopd, W, mc-ccr (s)),
              write-dn (L, muls (L, sopd, dopd, W), dn, s)) endlet
```

EXG instruction. Exchange the contents of two data registers.

```
DEFINITION:
exg-drdr-ins (ins, s)
    =
let dx be read-dn (L, d_rn (ins), s),
    dy be read-dn (L, s_rn (ins), s)
in
write-dn (L, dy, d_rn (ins), write-dn (L, dx, s_rn (ins), s)) endlet
```

Exchange the contents of two address registers.

```
DEFINITION:
exg-arar-ins (ins, s)
    =
let ax be read-an (L, d_rn (ins), s),
    ay be read-an (L, s_rn (ins), s)
```

```
in
write-an (L, ay, d_rn (ins), write-an (L, ax, s_rn (ins), s)) endlet
```

Exchange the contents of data and address registers.

```
DEFINITION:
exg-drar-ins (ins, s)
    =
let dx  be  read-dn (L, d_rn (ins), s),
    ay  be  read-an (L, s_rn (ins), s)
in
write-dn (L, ay, d_rn (ins), write-an (L, dx, s_rn (ins), s)) endlet
```

```
DEFINITION:
mul_w-ins (ins, s)
    =
if mul&div-addr-modep (ins)
then let s&addr  be  mc-instate (W, ins, s)
       in
       if mc-haltp (car (s&addr))  then car (s&addr)
       else let sopd  be  operand (W, cdr (s&addr), s)
              in
              if b0p (bitn (ins, 8))
              then mulu_w-ins (sopd, d_rn (ins), car (s&addr))
              else muls_w-ins (sopd,
                                   d_rn (ins),
                                   car (s&addr)) endif endlet endif endlet
else halt (MODE-SIGNAL, s) endif
```

Opcode 1100. The AND instruction group includes three instructions AND, MULS.W/MULU.W, and EXG. Detect ABCD.

```
DEFINITION:
and-group (oplen, ins, s)
    =
if oplen = Q  then mul_w-ins (ins, s)
elseif b0p (bitn (ins, 8))  then and-ins1 (oplen, ins, s)
elseif s_mode (ins) < 2
then if oplen = B  then halt ('abcd-unspecified, s)
       elseif oplen = W
       then if s_mode (ins) = 0  then exg-drdr-ins (ins, s)
              else exg-arar-ins (ins, s) endif
       elseif s_mode (ins) = 0  then halt (RESERVED-SIGNAL, s)
       else exg-drar-ins (ins, s) endif
else and-ins2 (oplen, ins, s) endif
```

OR instruction. The computation of the condition code register.

DEFINITION:
or-z (*sopd*, *dopd*)
    =
if $(sopd = 0) \land (dopd = 0)$ **then** B1
else B0 **endif**

DEFINITION:
or-n (*oplen*, *sopd*, *dopd*)
    =
if $(sopd < \exp(2, oplen - 1)) \land (dopd < \exp(2, oplen - 1))$ **then** B0
else B1 **endif**

DEFINITION:
or-cvznx (*oplen*, *sopd*, *dopd*, *ccr*)
    =
cvznx (B0, B0, or-z (*sopd*, *dopd*), or-n (*oplen*, *sopd*, *dopd*), ccr-x (*ccr*))

The effect of an execution of an OR instruction.

DEFINITION:
or-effect (*oplen*, *sopd*, *dopd*, *ccr*)
    =
cons (logor (*sopd*, *dopd*), or-cvznx (*oplen*, *sopd*, *dopd*, *ccr*))

Test if the addressing mode is illegal.

DEFINITION:
or-addr-modep1 (*ins*) = data-addr-modep (s_mode (*ins*), s_rn (*ins*))

DEFINITION:
or-addr-modep2 (*ins*)
    =
(alterable-addr-modep (s_mode (*ins*), s_rn (*ins*))
    $\land$
 memory-addr-modep (s_mode (*ins*), s_rn (*ins*)))

The execution of the OR instruction.

DEFINITION:
or-ins1 (*oplen*, *ins*, *s*)
    =
```
if or-addr-modep1 (ins)
then let s&addr  be  mc-instate (oplen, ins, s)
     in
     if mc-haltp (car (s&addr))  then car (s&addr)
```

```
        else d-mapping (oplen,
                        or-effect (oplen,
                                   operand (oplen, cdr (s&addr), s),
                                   read-dn (oplen, d_rn (ins), s),
                                   mc-ccr (s)),
                        d_rn (ins),
                        car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

DEFINITION:

```
or-mapping (sopd, oplen, ins, s)
    =
let s&addr  be  mc-instate (oplen, ins, s)
in
if mc-haltp (car (s&addr))  then car (s&addr)
else mapping (oplen,
              or-effect (oplen,
                         sopd,
                         operand (oplen, cdr (s&addr), s),
                         mc-ccr (s)),
              s&addr) endif endlet
```

DEFINITION:

```
or-ins2 (oplen, ins, s)
    =
if or-addr-modep2 (ins)
then or-mapping (read-dn (oplen, d_rn (ins), s), oplen, ins, s)
else halt (MODE-SIGNAL, s) endif
```

DIVU.W/DIVS.W instructions. D(32)/S(16) $\rightarrow$ D(16r:16q). 'iquot' and 'irem' expect that s and d are unsigned integers. They are used in the DIV instruction.

DEFINITION:

```
iquot (n, s, i, d, j)
    =
head (int-to-nat (iquotient (nat-to-int (d, j), nat-to-int (s, i)), j), n)
```

DEFINITION:

```
irem (n, s, i, d, j)
    =
head (int-to-nat (iremainder (nat-to-int (d, j), nat-to-int (s, i)), i), n)
```

DIVS.W instruction.

```
DEFINITION:
divs-v (n, sopd, i, dopd, j)
   =
if int-rangep (iquotient (nat-to-int (dopd, j), nat-to-int (sopd, i)), n)  then B0
else B1 endif
```

```
DEFINITION:
divs-z (n, sopd, i, dopd, j)
   =
if iquot (n, sopd, i, dopd, j) = 0  then B1
else B0 endif
```

```
DEFINITION:
divs-n (n, sopd, i, dopd, j)
   =
if iquot (n, sopd, i, dopd, j) < exp (2, n − 1)  then B0
else B1 endif
```

In our specification of DIV, we only make sure that the N and Z bits are set correctly when there is NO overflow. Since we test for overflow before this instruction is fully completed, the setting of CCR is actually the same as AND's if NO overflow occurs. When an overflow is detected, we simply halt the machine with an error signal.

If overflow or divide by zero happens during the DIV instructions, then the MC68020 manual states that values of N, Z, and V are undefined. Thus one should not count on the validity of these values in the error state returned by 'stepi.'

```
DEFINITION:
divs-cvznx (n, sopd, i, dopd, j, ccr)
   =
cvznx (B0, B0, divs-z (n, sopd, i, dopd, j), divs-n (n, sopd, i, dopd, j), ccr-x (ccr))
```

32/16 → 16r:16q.

```
DEFINITION:
divs_w-ins (sopd, dn, s)
   =
if nat-to-int (sopd, W) = 0  then halt ('trap-exception, s)
else let dopd  be  read-dn (L, dn, s)
     in
     if b0p (divs-v (W, sopd, W, dopd, L))
     then update-ccr (divs-cvznx (W, sopd, W, dopd, L, mc-ccr (s)),
                      write-dn (L,
                                app (W,
                              iquot (W, $sopd$, W, $sopd$, L),
                              irem (W, $sopd$, W, $dopd$, L)),
                          $dn$,
                          $s$))
   else halt ('divs-overflow,
          update-ccr (set-v (B1, mc-ccr ($s$)), $s$)) endif endlet endif
```

DIVU.W instruction.

DEFINITION: quot (n, x, y) = head $(y \div x, n)$

DEFINITION: rem (n, x, y) = head $(y$ **mod** $x, n)$

The condition codes for DIVU.

DEFINITION:
divu-v $(n, sopd, dopd)$
  =
if $(dopd \div sopd) <$ exp $(2, n)$ **then** B0
else B1 **endif**

DEFINITION:
divu-z $(n, sopd, dopd)$
  =
if quot $(n, sopd, dopd) = 0$ **then** B1
else B0 **endif**

DEFINITION:
divu-n $(n, sopd, dopd)$
  =
if quot $(n, sopd, dopd) <$ exp $(2, n - 1)$ **then** B0
else B1 **endif**

Same treatment as divs-cvznx.

DEFINITION:
divu-cvznx $(n, sopd, dopd, ccr)$
  =
cvznx (B0, B0, divu-z $(n, sopd, dopd)$, divu-n $(n, sopd, dopd)$, ccr-x (ccr))

32/16 $\rightarrow$ 16r:16q.

DEFINITION:
divu_w-ins $(sopd, dn, s)$
  =
if nat-to-uint $(sopd) = 0$ **then** halt ('trap-exception, s)
else let $dopd$ **be** read-dn (L, dn, s)

```
    in
    if b0p (divu-v (W, sopd, dopd))
    then update-ccr (divu-cvznx (W, sopd, dopd, mc-ccr (s)),
                     write-dn (L,
                               app (W,
                                    quot (W, sopd, dopd),
                                    rem (W, sopd, dopd)),
                               dn,
                               s))
    else halt ('divu-overflow,
               update-ccr (set-v (B1, mc-ccr (s)), s)) endif endlet endif
```

DEFINITION:

```
div_w-ins (ins, s)
   =
if mul&div-addr-modep (ins)
then let s&addr be mc-instate (W, ins, s)
      in
      if mc-haltp (car (s&addr)) then car (s&addr)
      else let sopd be operand (W, cdr (s&addr), s)
            in
            if b0p (bitn (ins, 8))
            then divu_w-ins (sopd, d_rn (ins), car (s&addr))
            else divs_w-ins (sopd,
                             d_rn (ins),
                             car (s&addr)) endif endlet endif endlet
else halt (MODE-SIGNAL, s) endif
```

Opcode 1000. The OR instruction group includes two instructions OR and DIVU.W/DIVS.W.

DEFINITION:

```
or-group (oplen, ins, s)
   =
if oplen = Q then div_w-ins (ins, s)
elseif b0p (bitn (ins, 8)) then or-ins1 (oplen, ins, s)
elseif s_mode (ins) < 2 then halt ('sbcd-pack-unpk-unspecified, s)
else or-ins2 (oplen, ins, s) endif
```

Rotate operations. Rotate left *cnt* times. *len* is supposed to be the length of x.

DEFINITION:

```
rol (len, x, cnt)
   =
```

let n **be** $len - (cnt$ **mod** $len)$
in
app$(n$, tail(x, n), head$(x, n))$ **endlet**

Rotate right cnt times. len is supposed to be the length of x.

DEFINITION:
ror(len, x, cnt)
=
let n **be** cnt **mod** len
in
app$(n$, tail(x, n), head$(x, n))$ **endlet**

For memory shift/rotate, only memory alterable addressing modes are allowed.

DEFINITION:
s&r-addr-modep(ins)
=
(memory-addr-modep(s_mode(ins), s_rn(ins))
$\wedge$
alterable-addr-modep(s_mode(ins), s_rn(ins)))

'i-data' returns a nonnegative integer. In register shift/rotate, it is the shift/rotate cnt. In ADDQ and SUBQ, it is the immediate data.

DEFINITION:
i-data(n)
=
if $n \simeq 0$ **then** 8
else n **endif**

DEFINITION:
sr-cnt(ins, s)
=
if b0p(bitn$(ins$, 5)) **then** i-data(d_rn(ins))
else read-dn(B, d_rn(ins), s) **mod** 64 **endif**

ROL and ROR instructions. We divide the ROL/ROR instruction into a few subinstructions.

Register ROL instruction. The setting of cvznx-flags for ROL.

DEFINITION:
rol-c(len, x, cnt)
=
if $cnt = 0$ **then** B0
else let n **be** cnt **mod** len

```
    in
    if $n \simeq 0$ then bcar$(x)$
    else bitn$(x,\ len - n)$ endif endlet endif
```

DEFINITION:
```
rol-z$(x)$
  =
if $x = 0$ then B1
else B0 endif
```

DEFINITION:
rol-n$(len,\ x,\ cnt) =$ bitn$(x,\ (len - (cnt$ **mod** $len)) - 1)$

DEFINITION:
```
rol-cvznx$(len,\ opd,\ cnt,\ ccr)$
  =
cvznx(rol-c$(len,\ opd,\ cnt)$, B0, rol-z$(opd)$, rol-n$(len,\ opd,\ cnt)$, ccr-x$(ccr)$)
```

DEFINITION:
```
rol-effect$(len,\ opd,\ cnt,\ ccr)$
  =
cons(rol$(len,\ opd,\ cnt)$, rol-cvznx$(len,\ opd,\ cnt,\ ccr)$)
```

DEFINITION:
```
register-rol-ins$(oplen,\ ins,\ s)$
  =
d-mapping$(oplen,$
          rol-effect$(oplen,$
                  read-dn$(oplen$, s_rn$(ins)$, $s)$,
                  sr-cnt$(ins,\ s)$,
                  mc-ccr$(s))$,
          s_rn$(ins)$,
          $s)$
```

Register ROR instruction.

DEFINITION:
```
ror-c$(len,\ x,\ cnt)$
  =
if $cnt = 0$ then B0
else let $n$ be $cnt$ mod $len$
    in
    if $n = 0$ then bitn$(x,\ len - 1)$
    else bitn$(x,\ n - 1)$ endif endlet endif
```

Definition:
```
ror-z ($opd$)
    =
if $opd = 0$ then B1
else B0 endif
```

Definition:
```
ror-n ($len$, $x$, $cnt$)
    =
let $n$ be $cnt$ mod $len$
in
if $n \simeq 0$ then bitn ($x$, $len - 1$)
else bitn ($x$, $n - 1$) endif endlet
```

Definition:
```
ror-cvznx ($len$, $opd$, $cnt$, $ccr$)
    =
cvznx (ror-c ($len$, $opd$, $cnt$), B0, ror-z ($opd$), ror-n ($len$, $opd$, $cnt$), ccr-x ($ccr$))
```

Definition:
```
ror-effect ($oplen$, $opd$, $cnt$, $ccr$)
    =
cons (ror ($oplen$, $opd$, $cnt$), ror-cvznx ($oplen$, $opd$, $cnt$, $ccr$))
```

Definition:
```
register-ror-ins ($oplen$, $ins$, $s$)
    =
d-mapping ($oplen$,
           ror-effect ($oplen$,
                       read-dn ($oplen$, s_rn ($ins$), $s$),
                       sr-cnt ($ins$, $s$),
                       mc-ccr ($s$)),
           s_rn ($ins$),
           $s$)
```

Memory ROL instruction. The operand size should be word, and the shift operation is one bit only.

Definition: mem-rol-effect (opd, ccr) = rol-effect (W, opd, 1, ccr)

Definition:
```
mem-rol-ins ($ins$, $s$)
    =
if s&r-addr-modep ($ins$)
then let $s\&addr$ be mc-instate (W, $ins$, $s$)
```

```
        in
        if mc-haltp (car (s&addr))  then car (s&addr)
        else mapping(W,
                     mem-rol-effect (operand (W, cdr (s&addr), s),
                                     mc-ccr (s)),
                     s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

Memory ROR instruction. The operand size should be word, and the shift operation is one bit only.

DEFINITION: mem-ror-effect (*opd*, *ccr*) = ror-effect (W, *opd*, 1, *ccr*)

DEFINITION:

```
mem-ror-ins (ins, s)
    =
if s&r-addr-modep (ins)
then let s&addr  be  mc-instate(W, ins, s)
        in
        if mc-haltp (car (s&addr))  then car (s&addr)
        else mapping(W,
                     mem-ror-effect (operand (W, cdr (s&addr), s),
                                     mc-ccr (s)),
                     s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

LSL and LSR instructions. We divided the LSL/LSR instruction into several subinstructions.

Register LSL instruction.

DEFINITION:

```
lsl-c (len, opd, cnt)
    =
if $cnt = 0$ then B0
elseif $cnt < len$ then bitn (opd, $len - cnt$)
else B0 endif
```

DEFINITION:

```
lsl-z (len, opd, cnt)
    =
if lsl (len, opd, cnt) = 0  then B1
else B0 endif
```

DEFINITION:

```
lsl-n (len, opd, cnt)
    =
**if** lsl ($len, opd, cnt$) $<$ exp ($\texttt{2}, len - 1$) **then** B0
**else** B1 **endif**
```

DEFINITION:

```
lsl-x ($len, opd, cnt, ccr$)
    =
**if** $cnt = \texttt{0}$ **then** ccr-x ($ccr$)
**else** lsl-c ($len, opd, cnt$) **endif**
```

DEFINITION:

```
lsl-cvznx ($len, opd, cnt, ccr$)
    =
cvznx (lsl-c ($len, opd, cnt$),
        B0,
        lsl-z ($len, opd, cnt$),
        lsl-n ($len, opd, cnt$),
        lsl-x ($len, opd, cnt, ccr$))
```

DEFINITION:

```
lsl-effect ($len, opd, cnt, ccr$)
    =
cons (lsl ($len, opd, cnt$), lsl-cvznx ($len, opd, cnt, ccr$))
```

DEFINITION:

```
register-lsl-ins ($oplen, ins, s$)
    =
d-mapping ($oplen$,
              lsl-effect ($oplen$,
                          read-dn ($oplen$, s_rn ($ins$), $s$),
                          sr-cnt ($ins, s$),
                          mc-ccr ($s$)),
              s_rn ($ins$),
              $s$)
```

Register LSR instruction.

DEFINITION:

```
lsr-c ($len, opd, cnt$)
    =
**if** $cnt = \texttt{0}$ **then** B0
**elseif** $len < cnt$ **then** B0
**else** bitn ($opd, cnt - 1$) **endif**
```

DEFINITION:

lsr-z (len, opd, cnt)

```
    =
if lsr (opd, cnt) = 0 then B1
else B0 endif
```

DEFINITION:

```
lsr-n (len, opd, cnt)
    =
if lsr (opd, cnt) < exp (2, len − 1) then B0
else B1 endif
```

DEFINITION:

```
lsr-x (len, opd, cnt, ccr)
    =
if cnt = 0 then ccr-x (ccr)
else lsr-c (len, opd, cnt) endif
```

DEFINITION:

```
lsr-cvznx (len, opd, cnt, ccr)
    =
cvznx (lsr-c (len, opd, cnt),
       B0,
       lsr-z (len, opd, cnt),
       lsr-n (len, opd, cnt),
       lsr-x (len, opd, cnt, ccr))
```

DEFINITION:

```
lsr-effect (len, opd, cnt, ccr)
    =
cons (lsr (opd, cnt), lsr-cvznx (len, opd, cnt, ccr))
```

DEFINITION:

```
register-lsr-ins (oplen, ins, s)
    =
d-mapping (oplen,
           lsr-effect (oplen,
                       read-dn (oplen, s_rn (ins), s),
                       sr-cnt (ins, s),
                       mc-ccr (s)),
           s_rn (ins),
           s)
```

Memory LSL instruction.

DEFINITION: mem-lsl-effect (*opd*, *ccr*) = lsl-effect (W, *opd*, 1, *ccr*)

DEFINITION:
```
mem-lsl-ins(ins, s)
    =
if s&r-addr-modep (ins)
then let s&addr be mc-instate(W, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else mapping(W,
                    mem-lsl-effect (operand (W, cdr (s&addr), s),
                                    mc-ccr (s)),
                    s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

Memory LSR instruction.

DEFINITION: mem-lsr-effect (*opd*, *ccr*) = lsr-effect (W, *opd*, 1, *ccr*)

DEFINITION:
```
mem-lsr-ins(ins, s)
    =
if s&r-addr-modep (ins)
then let s&addr be mc-instate(W, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else mapping(W,
                    mem-lsr-effect (operand (W, cdr (s&addr), s),
                                    mc-ccr (s)),
                    s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

ASL and ASR instructions.
Register ASL instruction.

DEFINITION: asl-c (*len*, *opd*, *cnt*) = lsl-c (*len*, *opd*, *cnt*)

DEFINITION:
```
asl-v (len, opd, cnt)
    =
if int-rangep (nat-to-int (opd, len), len − cnt) then B0
else B1 endif
```

DEFINITION:
```
asl-z (len, opd, cnt)
    =
if asl (len, opd, cnt) = 0 then B1
else B0 endif
```

DEFINITION:
```
asl-n ($len$, $opd$, $cnt$)
    =
if asl ($len$, $opd$, $cnt$) $<$ exp (2, $len - 1$) then B0
else B1 endif
```

DEFINITION:
```
asl-x ($len$, $opd$, $cnt$, $ccr$)
    =
if $cnt = 0$ then ccr-x ($ccr$)
else asl-c ($len$, $opd$, $cnt$) endif
```

DEFINITION:
```
asl-cvznx ($len$, $opd$, $cnt$, $ccr$)
    =
cvznx (asl-c ($len$, $opd$, $cnt$),
        asl-v ($len$, $opd$, $cnt$),
        asl-z ($len$, $opd$, $cnt$),
        asl-n ($len$, $opd$, $cnt$),
        asl-x ($len$, $opd$, $cnt$, $ccr$))
```

DEFINITION:
```
asl-effect ($len$, $opd$, $cnt$, $ccr$)
    =
cons (asl ($len$, $opd$, $cnt$), asl-cvznx ($len$, $opd$, $cnt$, $ccr$))
```

DEFINITION:
```
register-asl-ins ($oplen$, $ins$, $s$)
    =
d-mapping ($oplen$,
           asl-effect ($oplen$,
                       read-dn ($oplen$, s_rn ($ins$), $s$),
                       sr-cnt ($ins$, $s$),
                       mc-ccr ($s$)),
           s_rn ($ins$),
           $s$)
```

Register ASR instruction.

DEFINITION:
```
asr-c ($len$, $opd$, $cnt$)
    =
if $cnt = 0$ then B0
elseif $cnt < len$ then bitn ($opd$, $cnt - 1$)
else bitn ($opd$, $len - 1$) endif
```

DEFINITION:
asr-z (len, opd, cnt)
$=$
if asr $(len, opd, cnt) = \texttt{0}$ **then** $\textsc{b}1$
else $\textsc{b}0$ **endif**

DEFINITION:
asr-n (len, opd, cnt)
$=$
if asr $(len, opd, cnt) < \exp(\texttt{2}, len - 1)$ **then** $\textsc{b}0$
else $\textsc{b}1$ **endif**

DEFINITION:
asr-x (len, opd, cnt, ccr)
$=$
if $cnt = \texttt{0}$ **then** ccr-x (ccr)
else asr-c (len, opd, cnt) **endif**

DEFINITION:
asr-cvznx (len, opd, cnt, ccr)
$=$
```
cvznx (asr-c (len, opd, cnt),
       B0,
       asr-z (len, opd, cnt),
       asr-n (len, opd, cnt),
       asr-x (len, opd, cnt, ccr))
```

DEFINITION:
asr-effect (len, opd, cnt, ccr)
$=$
cons (asr (len, opd, cnt), asr-cvznx (len, opd, cnt, ccr))

DEFINITION:
register-asr-ins $(oplen, ins, s)$
$=$
```
d-mapping (oplen,
           asr-effect (oplen,
                       read-dn (oplen, s_rn (ins), s),
                       sr-cnt (ins, s),
                       mc-ccr (s)),
           s_rn (ins),
           s)
```

Memory ASL instruction.

DEFINITION: mem-asl-effect (opd, ccr) = asl-effect ($\textsc{w}$, opd, $\texttt{1}$, ccr)

DEFINITION:
```
mem-asl-ins($ins$, $s$)
   =
if s&r-addr-modep($ins$)
then let $s\&addr$ be mc-instate(W, $ins$, $s$)
      in
      if mc-haltp(car($s\&addr$)) then car($s\&addr$)
      else mapping(W,
                   mem-asl-effect(operand(W, cdr($s\&addr$), $s$),
                                  mc-ccr($s$)),
                   $s\&addr$) endif endlet
else halt(MODE-SIGNAL, $s$) endif
```

Memory ASR instruction.

DEFINITION: mem-asr-effect(opd, ccr) = asr-effect(W, opd, 1, ccr)

DEFINITION:
```
mem-asr-ins($ins$, $s$)
   =
if s&r-addr-modep($ins$)
then let $s\&addr$ be mc-instate(W, $ins$, $s$)
      in
      if mc-haltp(car($s\&addr$)) then car($s\&addr$)
      else mapping(W,
                   mem-asr-effect(operand(W, cdr($s\&addr$), $s$),
                                  mc-ccr($s$)),
                   $s\&addr$) endif endlet
else halt(MODE-SIGNAL, $s$) endif
```

ROXL and ROXR instructions.
'roxl' defines the rotate left with extend operation.

DEFINITION:
```
roxl($len$, $opd$, $cnt$, $x$)
   =
let $temp$ be $x + (2 * opd)$
in
rol(1 + $len$, $temp$, $cnt$) $\div$ 2 endlet
```

'roxr' defines the rotate right with extend operation.

DEFINITION:
```
roxr($len$, $opd$, $cnt$, $x$)
   =
let $temp$ be $opd + (x * \exp(2, len))$
```

```
in
ror (1 + $len$, $temp$, $cnt$) mod exp (2, $len$) endlet
```

Register ROXL instruction.

DEFINITION:

```
roxl-c ($len$, $opd$, $cnt$, $x$)
  =
let $tmp$ be $cnt$ mod (1 + $len$)
in
if $tmp$ = 0 then fix-bit ($x$)
else bitn ($opd$, $len - tmp$) endif endlet
```

DEFINITION:

```
roxl-z ($len$, $opd$, $cnt$, $x$)
  =
if roxl ($len$, $opd$, $cnt$, $x$) = 0 then B1
else B0 endif
```

DEFINITION:

```
roxl-n ($len$, $opd$, $cnt$, $x$) = bitn (roxl ($len$, $opd$, $cnt$, $x$), $len - 1$)
```

DEFINITION:

```
roxl-cvznx ($len$, $opd$, $cnt$, $x$)
  =
cvznx (roxl-c ($len$, $opd$, $cnt$, $x$),
       B0,
       roxl-z ($len$, $opd$, $cnt$, $x$),
       roxl-n ($len$, $opd$, $cnt$, $x$),
       roxl-c ($len$, $opd$, $cnt$, $x$))
```

DEFINITION:

```
roxl-effect ($len$, $opd$, $cnt$, $ccr$)
  =
cons (roxl ($len$, $opd$, $cnt$, ccr-x ($ccr$)), roxl-cvznx ($len$, $opd$, $cnt$, ccr-x ($ccr$)))
```

DEFINITION:

```
register-roxl-ins ($oplen$, $ins$, $s$)
  =
d-mapping ($oplen$,
           roxl-effect ($oplen$,
                        read-dn ($oplen$, s_rn ($ins$), $s$),
                        sr-cnt ($ins$, $s$),
                        mc-ccr ($s$)),
           s_rn ($ins$),
           $s$)
```

Register ROXR instruction.

DEFINITION:

```
roxr-c (len, opd, cnt, x)
    =
let tmp be cnt mod (1 + len)
in
if tmp = 0 then fix-bit (x)
else bitn (opd, tmp − 1) endif endlet
```

DEFINITION:

```
roxr-z (len, opd, cnt, x)
    =
if roxr (len, opd, cnt, x) = 0 then B1
else B0 endif
```

DEFINITION:

```
roxr-n (len, opd, cnt, x) = bitn (roxr (len, opd, cnt, x), len − 1)
```

DEFINITION:

```
roxr-cvznx (len, opd, cnt, x)
    =
cvznx (roxr-c (len, opd, cnt, x),
       B0,
       roxr-z (len, opd, cnt, x),
       roxr-n (len, opd, cnt, x),
       roxr-c (len, opd, cnt, x))
```

DEFINITION:

```
roxr-effect (len, opd, cnt, ccr)
    =
cons (roxr (len, opd, cnt, ccr-x (ccr)), roxr-cvznx (len, opd, cnt, ccr-x (ccr)))
```

DEFINITION:

```
register-roxr-ins (oplen, ins, s)
    =
d-mapping (oplen,
           roxr-effect (oplen,
                        read-dn (oplen, s_rn (ins), s),
                        sr-cnt (ins, s),
                        mc-ccr (s)),
           s_rn (ins),
           s)
```

Memory ROXL instruction.

DEFINITION:
mem-roxl-effect (*opd*, *ccr*) = roxl-effect (W, *opd*, `1`, *ccr*)

DEFINITION:
```
mem-roxl-ins(ins, s)
   =
if s&r-addr-modep (ins)
then let s&addr be mc-instate(W, ins, s)
      in
      if mc-haltp (car (s&addr)) then car (s&addr)
      else mapping(W,
                   mem-roxl-effect (operand (W, cdr (s&addr), s),
                                    mc-ccr (s)),
                   s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

Memory ROXR instruction.

DEFINITION:
mem-roxr-effect (*opd*, *ccr*) = roxr-effect (W, *opd*, `1`, *ccr*)

DEFINITION:
```
mem-roxr-ins(ins, s)
   =
if s&r-addr-modep (ins)
then let s&addr be mc-instate(W, ins, s)
      in
      if mc-haltp (car (s&addr)) then car (s&addr)
      else mapping(W,
                   mem-roxr-effect (operand (W, cdr (s&addr), s),
                                    mc-ccr (s)),
                   s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

Memory shift/rotate.

DEFINITION:
```
memory-shift-rotate(ins, s)
   =
if b0p (bitn (ins, 10))
then if b0p (bitn (ins, 9))
      then if b0p (bitn (ins, 8)) then mem-asr-ins(ins, s)
            else mem-asl-ins(ins, s) endif
      elseif b0p (bitn (ins, 8)) then mem-lsr-ins(ins, s)
      else mem-lsl-ins(ins, s) endif
```

```
elseif b0p (bitn (ins, 9))
then if b0p (bitn (ins, 8))  then mem-roxr-ins (ins, s)
       else mem-roxl-ins (ins, s) endif
elseif b0p (bitn (ins, 8))  then mem-ror-ins (ins, s)
else mem-rol-ins (ins, s) endif
```

Register shift/rotate.

DEFINITION:

```
register-shift-rotate (oplen, ins, s)
   =
if b0p (bitn (ins, 4))
then if b0p (bitn (ins, 3))
       then if b0p (bitn (ins, 8))  then register-asr-ins (oplen, ins, s)
              else register-asl-ins (oplen, ins, s) endif
       elseif b0p (bitn (ins, 8))  then register-lsr-ins (oplen, ins, s)
       else register-lsl-ins (oplen, ins, s) endif
elseif b0p (bitn (ins, 3))
then if b0p (bitn (ins, 8))  then register-roxr-ins (oplen, ins, s)
       else register-roxl-ins (oplen, ins, s) endif
elseif b0p (bitn (ins, 8))  then register-ror-ins (oplen, ins, s)
else register-rol-ins (oplen, ins, s) endif
```

The bit field instruction group consists of BFxxx instructions. All of these instructions are new in the MC68020. Note that bit 15 in the extension word has to be 0!

DEFINITION:
bf-subgroup (*ins*, *s*) = halt (`'i-will-do-it-later`, *s*)

Opcode 1110. The shift/rotate instruction group includes the ASL/ASR, LSL/LSR, ROL/ROR, ROXL/RORL, BFTST, BFEXTU, BFCHG, BFEXTS, BFCLR, BFFFO, BFSET, and BFINS instructions. But it actually divides into many varieties of these instructions.

DEFINITION:

```
s&r-group (ins, s)
   =
if op-len (ins) = Q
then if b0p (bitn (ins, 11))  then memory-shift-rotate (ins, s)
       else bf-subgroup (ins, s) endif
else register-shift-rotate (op-len (ins), ins, s) endif
```

MOVE instruction.

DEFINITION:
move-addr-modep (*oplen*, *ins*)

```
    =
(addr-modep (s_mode ($ins$), s_rn ($ins$))
     $\wedge$
 data-addr-modep (d_mode ($ins$), d_rn ($ins$))
     $\wedge$
 alterable-addr-modep (d_mode ($ins$), d_rn ($ins$))
     $\wedge$
 ($\neg$ byte-an-direct-modep ($oplen$, s_mode ($ins$))))
```

DEFINITION:

```
move-z ($oplen$, $sopd$)
    =
**if** head ($sopd$, $oplen$) = 0 **then** B1
**else** B0 **endif**
```

DEFINITION:

```
move-n ($oplen$, $sopd$)
    =
**if** $sopd$ < exp (2, $oplen$ − 1) **then** B0
**else** B1 **endif**
```

The definition of cvznx-flags of MOVE instruction. It is also used in TST and TAS instructions.

DEFINITION:

```
move-cvznx ($oplen$, $sopd$, $ccr$)
    =
cvznx (B0, B0, move-z ($oplen$, $sopd$), move-n ($oplen$, $sopd$), ccr-x ($ccr$))
```

DEFINITION:

```
move-effect ($oplen$, $sopd$, $ccr$) = cons ($sopd$, move-cvznx ($oplen$, $sopd$, $ccr$))
```

DEFINITION:

```
move-mapping($sopd$, $oplen$, $ins$, $s$)
    =
**let** $s\&addr$ **be** effec-addr ($oplen$, d_mode ($ins$), d_rn ($ins$), $s$)
**in**
**if** mc-haltp (car ($s\&addr$)) **then** car ($s\&addr$)
**else** mapping ($oplen$, move-effect ($oplen$, $sopd$, mc-ccr ($s$)), $s\&addr$) **endif** **endlet**
```

DEFINITION:

```
move-ins ($oplen$, $ins$, $s$)
    =
**if** move-addr-modep ($oplen$, $ins$)
**then let** $s\&addr$ **be** mc-instate ($oplen$, $ins$, $s$)
```

```
      in
      if mc-haltp (car (s&addr))  then car (s&addr)
      else move-mapping(operand (oplen, cdr (s&addr), s),
                        oplen,
                        ins,
                        car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

MOVEA instruction. MOVEA differs from MOVE in several ways: ccr is not affected and word operation is sign-extended. Therefore, we define it separately.

DEFINITION:
movea-addr-modep (*ins*) = addr-modep (s_mode (*ins*), s_rn (*ins*))

DEFINITION:

```
movea-ins(oplen, ins, s)
   =
if movea-addr-modep (ins)
then let s&addr  be  mc-instate(oplen, ins, s)
      in
      if mc-haltp (car (s&addr))  then car (s&addr)
      else write-an (oplen,
                    ext (oplen, operand (oplen, cdr (s&addr), s), L),
                    d_rn (ins),
                    car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

Opcode 0010 and 0011. The following definition is defined to distinguish MOVE and MOVEA instructions. This definition is only for word and long operations.

DEFINITION:

```
move-group(oplen, ins, s)
   =
if d_mode (ins) = 1  then movea-ins(oplen, ins, s)
else move-ins(oplen, ins, s) endif
```

LEA instruction.

DEFINITION:
lea-addr-modep (*ins*) = control-addr-modep (s_mode (*ins*), s_rn (*ins*))

'lea-ins' calls 'effec-addr', instead of 'mc-instate', since the effective address is JUST what we need. Notice that LEA and PEA only deal with memory address. The address direct modes are not allowed. Operation size: long.

```
DEFINITION:
lea-ins (smode, ins, s)
    =
if lea-addr-modep (ins)
then let s&addr be effec-addr (L, smode, s_rn (ins), s)
        in
        if mc-haltp (car (s&addr)) then car (s&addr)
        else write-an (L, cddr (s&addr), d_rn (ins), car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

EXTB instruction. Sign-extend a byte to a longword. It is new in the MC68020.

```
DEFINITION:
ext-z (opd)
    =
if opd = 0 then B1
else B0 endif
```

```
DEFINITION:
ext-n (n, opd)
    =
if negativep (nat-to-int (opd, n)) then B1
else B0 endif
```

```
DEFINITION:
ext-cvznx (n, opd, ccr) = cvznx (B0, B0, ext-z (opd), ext-n (n, opd), ccr-x (ccr))
```

```
DEFINITION:
ext-effect (n, opd, size, ccr) = cons (ext (n, opd, size), ext-cvznx (n, opd, ccr))
```

```
DEFINITION:
extb-ins (ins, s)
    =
d-mapping(L, ext-effect (B, read-dn (B, s_rn (ins), s), L, mc-ccr (s)), s_rn (ins), s)
```

The LEA instruction subgroup includes LEA and EXTB instructions.

```
DEFINITION:
lea-subgroup (ins, s)
    =
if s_mode (ins) = 0
then if bits (ins, 9, 11) = 0 then extb-ins (ins, s)
        else halt (RESERVED-SIGNAL, s) endif
else lea-ins (s_mode (ins), ins, s) endif
```

CLR instruction.

DEFINITION:

```
clr-addr-modep (ins)
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

DEFINITION: clr-cvznx (*ccr*) = cvznx (B0, B0, B1, B0, ccr-x (*ccr*))

DEFINITION: clr-effect (*ccr*) = cons (0, clr-cvznx (*ccr*))

DEFINITION:

```
clr-ins (oplen, ins, s)
    =
if clr-addr-modep (ins)
then let s&addr be effec-addr (oplen, s_mode (ins), s_rn (ins), s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else mapping (oplen, clr-effect (mc-ccr (s)), s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

MOVE from CCR instruction.

DEFINITION:

```
move-from-ccr-addr-modep (ins)
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

This instruction has no effect on CCR. Therefore, the original CCR is copied for the updating. This is intended to have a uniform treatment for cvznx-flags. It makes it possible to use one theorem to characterize the action.

DEFINITION: move-from-ccr-effect (*ccr*) = cons (*ccr*, *ccr*)

DEFINITION:

```
move-from-ccr-ins (ins, s)
    =
if move-from-ccr-addr-modep (ins)
then let s&addr be mc-instate (W, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else mapping (W, move-from-ccr-effect (mc-ccr (s)), s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

The CLR instruction subgroup includes CLR and MOVE from CCR instructions.

```
DEFINITION:
clr-subgroup (ins, s)
   =
if op-len (ins) = Q then move-from-ccr-ins (ins, s)
else clr-ins (op-len (ins), ins, s) endif
```

NEGX instruction.

```
DEFINITION:
negx-addr-modep (ins)
   =
(data-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

```
DEFINITION:
negx-ins (oplen, ins, s)
   =
if negx-addr-modep (ins)
then let s&addr be mc-instate (oplen, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else mapping (oplen,
                     subx-effect (oplen,
                                  operand (oplen, cdr (s&addr), s),
                                  0,
                                  mc-ccr (s)),
                     s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

The NEGX instruction subgroup includes the NEGX instruction. Detect MOVE from SR.

```
DEFINITION:
negx-subgroup (ins, s)
   =
if op-len (ins) = Q then halt ('move-from-sr-privileged, s)
else negx-ins (op-len (ins), ins, s) endif
```

NEG instruction.

```
DEFINITION:
neg-addr-modep (ins)
```

```
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

DEFINITION:

```
neg-ins (oplen, ins, s)
    =
if neg-addr-modep (ins)
then let s&addr  be  mc-instate (oplen, ins, s)
       in
       if mc-haltp (car (s&addr))  then car (s&addr)
       else mapping (oplen,
                         sub-effect (oplen,
                                       operand (oplen, cdr (s&addr), s),
                                       0),
                         s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

MOVE to CCR instruction.

DEFINITION:

```
move-to-ccr-addr-modep (ins) = data-addr-modep (s_mode (ins), s_rn (ins))
```

DEFINITION:

```
move-to-ccr-ins (ins, s)
    =
if move-to-ccr-addr-modep (ins)
then let s&addr  be  mc-instate (W, ins, s)
       in
       if mc-haltp (car (s&addr))  then car (s&addr)
       else update-ccr (head (operand (W, cdr (s&addr), s), B),
                          car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

The NEG instruction subgroup includes NEG and MOVE to CCR instructions.

DEFINITION:

```
neg-subgroup (ins, s)
    =
if op-len (ins) = Q  then move-to-ccr-ins (ins, s)
else neg-ins (op-len (ins), ins, s) endif
```

PEA instruction.

```
DEFINITION:
pea-addr-modep (ins) = control-addr-modep (s_mode (ins), s_rn (ins))

DEFINITION:
pea-ins (smode, ins, s)
   =
if pea-addr-modep (ins)
then let s&addr  be  effec-addr (L, smode, s_rn (ins), s)
       in
       if mc-haltp (car (s&addr))  then car (s&addr)
       else push-sp (LSZ, cddr (s&addr), car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

SWAP instruction.

```
DEFINITION:
swap-z (opd)
   =
if fix (opd) = 0  then B1
else B0 endif

DEFINITION:  swap-n (opd) = bitn (opd, 15)

DEFINITION:
swap-cvznx (opd, ccr) = cvznx (B0, B0, swap-z (opd), swap-n (opd), ccr-x (ccr))

DEFINITION:
swap-effect (opd, ccr)
   =
cons (app (W, tail (opd, W), head (opd, W)), swap-cvznx (opd, ccr))

DEFINITION:
swap-ins (ins, s)
   =
d-mapping (L, swap-effect (read-dn (L, s_rn (ins), s), mc-ccr (s)), s_rn (ins), s)
```

The PEA instruction subgroup includes PEA and SWAP. Detect BKPT.

```
DEFINITION:
pea-subgroup (ins, s)
   =
if s_mode (ins) < 2
then if s_mode (ins) = 0  then swap-ins (ins, s)
       else halt ('bkpt-unspecified, s) endif
else pea-ins (s_mode (ins), ins, s) endif
```

EXT instruction.

Definition:
```
ext-ins (ins, s)
   =
if b0p (bitn (ins, 6))
then d-mapping (W,
                 ext-effect (B, read-dn (B, s_rn (ins), s), W, mc-ccr (s)),
                 s_rn (ins),
                 s)
else d-mapping (L,
                ext-effect (W, read-dn (W, s_rn (ins), s), L, mc-ccr (s)),
                s_rn (ins),
                s) endif
```

MOVEM Rn to EA instruction. A pair of functions for multiple read/write on memory.

Definition:
```
readm-mem (opsz, addr, mem, n)
   =
if n ≃ 0 then nil
else cons (read-mem (addr, mem, opsz),
           readm-mem (opsz, add (L, addr, opsz), mem, n − 1)) endif
```

Definition:
```
writem-mem (opsz, vlst, addr, mem)
   =
if listp (vlst)
then writem-mem (opsz,
                 cdr (vlst),
                 add (L, addr, opsz),
                 write-mem (car (vlst), addr, mem, opsz))
else mem endif
```

A pair of functions for multiple read/write on the register file.

Definition:
```
readm-rn (oplen, rnlst, rfile)
   =
if listp (rnlst)
then cons (read-rn (oplen, car (rnlst), rfile), readm-rn (oplen, cdr (rnlst), rfile))
else nil endif
```

Definition:
```
writem-rn (oplen, vlst, rnlst, rfile)
   =
if listp (rnlst)
```

```
then writem-rn ($oplen$,
                cdr ($vlst$),
                cdr ($rnlst$),
                write-rn (L, ext ($oplen$, car ($vlst$), L), car ($rnlst$), $rfile$))
else $rfile$ endif
```

A list of the number of registers to be moved.

DEFINITION:

```
movem-rnlst ($mask$, $i$)
  =
if $mask \simeq$ 0 then nil
elseif b0p (bcar ($mask$)) then movem-rnlst(bcdr ($mask$), $1 + i$)
else cons ($i$, movem-rnlst(bcdr ($mask$), $1 + i$)) endif
```

DEFINITION:

```
movem-len ($mask$)
  =
if $mask \simeq$ 0 then 0
elseif b0p (bcar ($mask$)) then movem-len(bcdr ($mask$))
else 1 + movem-len(bcdr ($mask$)) endif
```

DEFINITION:

```
writemp ($mask$, $oplen$, $addr$, $mem$)
  =
write-memp ($addr$, $mem$, op-sz ($oplen$) $*$ movem-len ($mask$))
```

In the case of predecrement, there are a few things we have to treat separately. The order of the mask is the reverse of the other cases.

DEFINITION:

```
movem-pre-rnlst ($mask$, $i$, $lst$)
  =
if $mask \simeq$ 0 then $lst$
elseif b0p (bcar ($mask$)) then movem-pre-rnlst (bcdr ($mask$), $i - 1$, $lst$)
else movem-pre-rnlst (bcdr ($mask$), $i - 1$, cons ($i$, $lst$)) endif
```

The reason we modify the address register *rn* here is that if it is also moved to memory, it is changed before it is moved. This function returns a 'cons': the first element is an intermediate state with the address register *rn* changed, the second element is the starting memory address to move those registers.

DEFINITION:

```
movem-predec ($mask$, $oplen$, $rn$, $s$)
  =
let $addr$ be read-an (L, $rn$, $s$)
```

```
in
cons (write-an (L, pre-effect (oplen, rn, addr), rn, s),
      cons ('m, sub (L, op-sz (oplen) * movem-len (mask), addr))) endlet
```

The addressing modes are control alterable plus predecrement. We deal with -(An) separately.

DEFINITION:
movem-rn-ea-addr-modep (*ins*)

```
   =
(alterable-addr-modep (s_mode (ins), s_rn (ins))
    ∧
 control-addr-modep (s_mode (ins), s_rn (ins)))
```

Note in the predecrement mode, if mask = 0, there is no action on An.

DEFINITION:
movem-rn-ea-ins (*mask*, *oplen*, *ins*, *s*)

```
   =
if predec-modep (s_mode (ins))
then let s&addr be movem-predec (mask, oplen, s_rn (ins), s)
     in
     if writemp (mask, oplen, cddr (s&addr), mc-mem (s))
     then write-an (L,
                    cddr (s&addr),
                    s_rn (ins),
                    update-mem (writem-mem (op-sz (oplen),
                                            readm-rn (oplen,
                                                      movem-pre-rnlst (mask,
                                                                       15,
                                                                       nil),
                                                      mc-rfile (car (s&addr))),
                                            cddr (s&addr),
                                            mc-mem (s)),
                                car (s&addr)))
     else halt (WRITE-SIGNAL, s) endif endlet
elseif movem-rn-ea-addr-modep (ins)
then let s&addr be effec-addr (oplen, s_mode (ins), s_rn (ins), s)
     in
     if mc-haltp (car (s&addr)) then car (s&addr)
     elseif writemp (mask, oplen, cddr (s&addr), mc-mem (s))
     then update-mem (writem-mem (op-sz (oplen),
                                  readm-rn (oplen,
                                            movem-rnlst (mask, 0),
                                            mc-rfile (s)),
                                  cddr (s&addr),
                                  mc-mem (s)),
                         car (s&addr))
else halt (WRITE-SIGNAL, s) endif endlet
else halt (MODE-SIGNAL, s) endif
```

The EXT instruction subgroup includes EXT and MOVEM Rn to EA.

DEFINITION:

```
ext-subgroup (ins, s)
  =
if s_mode (ins) = 0  then ext-ins (ins, s)
elseif pc-word-readp (mc-pc (s), mc-mem (s))
then movem-rn-ea-ins (pc-word-read (mc-pc (s), mc-mem (s)),
                      if b0p (bitn (ins, 6))  then W
                      else L endif,
                      ins,
                      update-pc (add (L, mc-pc (s), WSZ), s))
else halt (PC-SIGNAL, s) endif
```

TST instruction. MC68020 and MC68000 differ about addressing modes.

DEFINITION:

```
tst-addr-modep (oplen, ins)
  =
if oplen = B  then data-addr-modep (s_mode (ins), s_rn (ins))
else addr-modep (s_mode (ins), s_rn (ins)) endif
```

DEFINITION:

```
tst-ins (oplen, ins, s)
  =
if tst-addr-modep (oplen, ins)
then let s&addr  be  mc-instate (oplen, ins, s)
    in
    if mc-haltp (car (s&addr))  then car (s&addr)
    else update-ccr (move-cvznx (oplen,
                                 operand (oplen, cdr (s&addr), s),
                                 mc-ccr (s)),
                     car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

TAS instruction. It is usually used as a multiprocessor operation.

DEFINITION:

```
tas-addr-modep (ins)
  =
(data-addr-modep (s_mode (ins), s_rn (ins))
    ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

DEFINITION:
tas-effect (*opd*, *ccr*) = cons (setn (*opd*, `7`, B1), move-cvznx (B, *opd*, *ccr*))

The opsize of the TAS instruction is byte.

DEFINITION:

```
tas-ins (ins, s)
  =
if tas-addr-modep (ins)
then let s&addr  be  mc-instate (B, ins, s)
     in
     if mc-haltp (car (s&addr))  then car (s&addr)
     else mapping (B,
                   tas-effect (operand (B, cdr (s&addr), s), mc-ccr (s)),
                   s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

The TST instruction subgroup includes TAS and TST. Detect ILLEGAL instruction.

DEFINITION:

```
tst-subgroup (ins, s)
  =
if op-len (ins) = Q
then if head (ins, 6) = 60  then halt ('illegal-unspecified, s)
     else tas-ins (ins, s) endif
else tst-ins (op-len (ins), ins, s) endif
```

DIVS_L instructions. D / S → D. 32/32 → 32q, 32/32 → 32r:32q. The order of write-dn: remainder first, and then quotient. The overflow happens only when the *dopd* is -2^{31} and *sopd* is -1.

DEFINITION:

```
divsl_l (sopd, dopd, dq, dr, s)
  =
if b0p (divs-v (L, sopd, L, dopd, L))
then let q  be  iquot (L, sopd, L, dopd, L),
         r  be  irem (L, sopd, L, dopd, L)
     in
     update-ccr (divs-cvznx (L, sopd, L, dopd, L, mc-ccr (s)),
                 write-dn (L, q, dq, write-dn (L, r, dr, s))) endlet
else halt ('divs-overflow, update-ccr (set-v (B1, mc-ccr (s)), s)) endif
```

64/32 → 32r:32q.

DEFINITION:
```
divs_l (sopd, dopd_low, dq, dr, s)
    =
let dopd  be  app (L, dopd_low, read-dn (L, dr, s))
in
if b0p (divs-v (L, sopd, L, dopd, Q))
then let q  be  iquot (L, sopd, L, dopd, Q),
         r  be  irem (L, sopd, L, dopd, Q)
     in
     update-ccr (divs-cvznx (L, sopd, L, dopd, Q, mc-ccr (s)),
                 write-dn (L, q, dq, write-dn (L, r, dr, s))) endlet
else halt ('divs-overflow, update-ccr (set-v (B1, mc-ccr (s)), s)) endif endlet
```

DIVU_L instructions. D / S → D. 32/32 → 32q, 32/32 → 32r:32q. In this case, overflow never happens! It is justified by the event quotient-nat-rangep.

DEFINITION:
```
divul_l (sopd, dopd, dq, dr, s)
    =
let q  be  quot (L, sopd, dopd),
    r  be  rem (L, sopd, dopd)
in
update-ccr (divu-cvznx (L, sopd, dopd, mc-ccr (s)),
            write-dn (L, q, dq, write-dn (L, r, dr, s))) endlet
```

64/32 → 32r:32q.

DEFINITION:
```
divu_l (sopd, dopd_low, dq, dr, s)
    =
let dopd  be  app (L, dopd_low, read-dn (L, dr, s))
in
if b0p (divu-v (L, sopd, dopd))
then let q  be  quot (L, sopd, dopd),
         r  be  rem (L, sopd, dopd)
     in
     update-ccr (divu-cvznx (L, sopd, dopd, mc-ccr (s)),
                 write-dn (L, q, dq, write-dn (L, r, dr, s))) endlet
else halt ('divu-overflow, update-ccr (set-v (B1, mc-ccr (s)), s)) endif endlet
```

DEFINITION: dq (*word*) = bits (*word*, 12, 14)

DEFINITION: dr (*word*) = bits (*word*, 0, 2)

DEFINITION:

```
div_l-ins (sopd, word, s)
    =
let dopd_low be read-dn (L, dq (word), s)
in
if b0p (bitn (word, 11))
then if nat-to-uint (sopd) = 0 then halt ('trap-exception, s)
       elseif b0p (bitn (word, 10))
       then divul_l (sopd, dopd_low, dq (word), dr (word), s)
       else divu_l (sopd, dopd_low, dq (word), dr (word), s) endif
elseif nat-to-int (sopd, L) = 0 then halt ('trap-exception, s)
elseif b0p (bitn (word, 10))
then divsl_l (sopd, dopd_low, dq (word), dr (word), s)
else divs_l (sopd, dopd_low, dq (word), dr (word), s) endif endlet
```

MULS/MULU-long instructions. S * D → D.

DEFINITION:

```
mulu_l_dl (sopd, dopd, dl, s)
    =
update-ccr (mulu-cvznx (L, sopd, dopd, L, mc-ccr (s)),
              write-dn (L, mulu (L, sopd, dopd, L), dl, s))
```

DEFINITION:

```
mulu_l_dldh (sopd, dopd, dl, dh, s)
    =
if dl = dh then halt ('mc-undefined, s)
else update-ccr (mulu-cvznx (Q, sopd, dopd, L, mc-ccr (s)),
                    write-dn (L,
                               tail (mulu (Q, sopd, dopd, L), L),
                               dh,
                               write-dn (L, head (mulu (Q, sopd, dopd, L), L), dl, s))) endif
```

DEFINITION:

```
muls_l_dl (sopd, dopd, dl, s)
    =
update-ccr (muls-cvznx (L, sopd, dopd, L, mc-ccr (s)),
              write-dn (L, muls (L, sopd, dopd, L), dl, s))
```

DEFINITION:

```
muls_l_dldh (sopd, dopd, dl, dh, s)
    =
if dl = dh then halt ('mc-undefined, s)
else update-ccr (muls-cvznx (Q, sopd, dopd, L, mc-ccr (s)),
                    write-dn (L,
                    tail (muls (Q, $sopd$, $dopd$, L), L),
                    $dh$,
                    write-dn (L, head (muls (Q, $sopd$, $dopd$, L), L), $dl$, $s$))) endif
```

DEFINITION: dl ($word$) = bits ($word$, 12, 14)

DEFINITION: dh ($word$) = bits ($word$, 0, 2)

DEFINITION:

```
mul_l-ins ($sopd$, $word$, $s$)
  =
let $dopd$ be read-dn (L, dl ($word$), $s$)
in
if b0p (bitn ($word$, 11))
then if b0p (bitn ($word$, 10)) then mulu_l_dl ($sopd$, $dopd$, dl ($word$), $s$)
     else mulu_l_dldh ($sopd$, $dopd$, dl ($word$), dh ($word$), $s$) endif
elseif b0p (bitn ($word$, 10)) then muls_l_dl ($sopd$, $dopd$, dl ($word$), $s$)
else muls_l_dldh ($sopd$, $dopd$, dl ($word$), dh ($word$), $s$) endif endlet
```

DEFINITION:

```
mul-div_l-ins ($word$, $ins$, $s$)
  =
if b0p (bitn ($word$, 15)) $\wedge$ (bits ($word$, 3, 9) = 0)
then if mul&div-addr-modep ($ins$)
     then let $s\&addr$ be mc-instate (L, $ins$, $s$)
          in
          if mc-haltp (car ($s\&addr$)) then car ($s\&addr$)
          else let $sopd$ be operand (L, cdr ($s\&addr$), car ($s\&addr$))
               in
               if b0p (bitn ($ins$, 6))
               then mul_l-ins ($sopd$, $word$, car ($s\&addr$))
               else div_l-ins ($sopd$,
                               $word$,
                               car ($s\&addr$)) endif endlet endif endlet
     else halt (MODE-SIGNAL, $s$) endif
else halt (RESERVED-SIGNAL, $s$) endif
```

MOVEM EA to RN instruction. The addressing modes are control plus postincrement. We deal with (An)+ separately.

DEFINITION:
movem-ea-rn-addr-modep (ins) = control-addr-modep (s_mode (ins), s_rn (ins))

DEFINITION:

```
readmp ($mask$, $oplen$, $addr$, $mem$)
  =
read-memp ($addr$, $mem$, op-sz ($oplen$) $*$ movem-len ($mask$))
```

In the mode of postincrement, if the address register is also loaded from the memory, the value of it upon completion of this instruction has no difference from the other modes.

DEFINITION:

```
movem-ea-rn-ins(mask, oplen, ins, s)
    =
if postinc-modep (s_mode (ins))
then let addr  be  read-an (L, s_rn (ins), s)
       in
       if readmp (mask, oplen, addr, mc-mem(s))
       then write-an (L,
                          add (L, addr, op-sz (oplen) * movem-len (mask)),
                          s_rn (ins),
                          update-rfile (writem-rn (oplen,
                                                    readm-mem (op-sz (oplen),
                                                               addr,
                                                               mc-mem (s),
                                                               movem-len (mask)),
                                                    movem-rnlst (mask, 0),
                                                    mc-rfile (s)),
                                        s))
       else halt (READ-SIGNAL, s) endif endlet
elseif movem-ea-rn-addr-modep (ins)
then let s&addr  be  effec-addr (oplen, s_mode (ins), s_rn (ins), s)
       in
       if mc-haltp (car (s&addr))  then car (s&addr)
       elseif readmp (mask, oplen, cddr (s&addr), mc-mem(s))
       then update-rfile (writem-rn (oplen,
                                      readm-mem (op-sz (oplen),
                                                 cddr (s&addr),
                                                 mc-mem (s),
                                                 movem-len (mask)),
                                      movem-rnlst (mask, 0),
                                      mc-rfile (s)),
                           car (s&addr))
       else halt (READ-SIGNAL, s) endif endlet
else halt (MODE-SIGNAL, s) endif
```

The MOVEM-EA-RN-SUBGROUP includes MOVEM, DIVS/U and MULS/U instructions.

DEFINITION:

```
movem-ea-rn-subgroup (ins, s)
    =
if pc-word-readp (mc-pc (s), mc-mem (s))
then let word be pc-word-read (mc-pc (s), mc-mem (s))
     in
     if b0p (bitn (ins, 7))
     then mul-div_l-ins (word, ins, update-pc (add (L, mc-pc (s), WSZ), s))
     else movem-ea-rn-ins (word,
                           if b0p (bitn (ins, 6)) then W
                           else L endif,
                           ins,
                           update-pc (add (L, mc-pc (s), WSZ), s)) endif endlet
else halt (PC-SIGNAL, s) endif
```

LINK-long instruction. LINK and UNLK are somewhat complicated. When sp is used as an, the execution order seems different from a simple instantiation.

DEFINITION:

```
link-mapping (an, disp, s)
  =
let sp be sub (L, LSZ, read-sp (s))
in
if write-memp (sp, mc-mem (s), LSZ)
then update-mem (write-mem (read-an (L, an, s), sp, mc-mem (s), LSZ),
                 write-sp (add (L, sp, disp), write-an (L, sp, an, s)))
else halt (WRITE-SIGNAL, s) endif endlet
```

DEFINITION:

```
link_l-ins (an, s)
  =
if pc-long-readp (mc-pc (s), mc-mem (s))
then link-mapping (an,
                   pc-long-read (mc-pc (s), mc-mem (s)),
                   update-pc (add (L, mc-pc (s), LSZ), s))
else halt (PC-SIGNAL, s) endif
```

LINK-word instruction.

DEFINITION:

```
link_w-ins (an, s)
  =
if pc-word-readp (mc-pc (s), mc-mem (s))
then link-mapping (an,
                   ext (W, pc-word-read (mc-pc (s), mc-mem (s)), L),
                   update-pc (add (L, mc-pc (s), WSZ), s))
else halt (PC-SIGNAL, s) endif
```

UNLK instruction.

DEFINITION:
```
unlk-ins ($an$, $s$)
  =
let $sp$ be read-an (L, $an$, $s$)
in
if long-readp ($sp$, mc-mem ($s$))
then write-an (L, long-read ($sp$, mc-mem ($s$)), $an$, write-sp (add (L, $sp$, LSZ), $s$))
else halt (READ-SIGNAL, $s$) endif endlet
```

The unlk instruction subgroup includes UNLK and LINK-word instructions. detect trap instruction.

DEFINITION:
```
unlk-subgroup ($ins$, $s$)
  =
if b0p (bitn ($ins$, 4)) then halt ('trap-unspecified, $s$)
elseif b0p (bitn ($ins$, 3)) then link_w-ins (s_rn ($ins$), $s$)
else unlk-ins (s_rn ($ins$), $s$) endif
```

NOP instruction. The machine state, except the program counter, is not affected. But we have already incremented pc when we read the first word of the current instruction. Therefore, we simply return s.

DEFINITION: nop-ins (s) = s

RTD instruction.

DEFINITION:
```
rtd-mapping ($sp$, $disp$, $s$)
  =
if long-readp ($sp$, mc-mem ($s$))
then let $new\text{-}sp$ be add (L, add (L, $sp$, LSZ), ext (W, $disp$, L))
     in
     update-pc (long-read ($sp$, mc-mem ($s$)), write-sp ($new\text{-}sp$, $s$)) endlet
else halt (READ-SIGNAL, $s$) endif
```

DEFINITION:
```
rtd-ins ($s$)
  =
if pc-word-readp (mc-pc ($s$), mc-mem ($s$))
then rtd-mapping (read-sp ($s$), pc-word-read (mc-pc ($s$), mc-mem ($s$)), $s$)
else halt (PC-SIGNAL, $s$) endif
```

RTS instruction. Notice that disp is 0.

DEFINITION: rts-ins (s) = rtd-mapping (read-sp (s), 0, s)

RTR instruction. Notice that disp is 0.

DEFINITION:

```
rtr-ins (s)
   =
let sp  be  read-sp (s)
in
if word-readp (sp, mc-mem (s))
then rtd-mapping (add (L, sp, WSZ),
                  0,
                  update-ccr (word-read (sp, mc-mem (s)), s))
else halt (READ-SIGNAL, s) endif endlet
```

TRAPV instruction. If the overflow is set, we simply halt the machine. Otherwise, nop. To handle this instruction in verifications, we intend to prove the overflow is not set, and hence the machine performs nop.

DEFINITION: bvs (v) = fix-bit (v)

DEFINITION:

```
trapv-ins (s)
   =
if b1p (bvs (ccr-v (mc-ccr (s))))  then halt ('trapv-exception, s)
else s endif
```

The NOP instruction subgroup includes NOP, RTD, RTS, and RTR instructions. Detect RESET, STOP, RTE, and TRAPV.

DEFINITION:

```
nop-subgroup (ins, s)
   =
if b0p (bitn (ins, 2))
then if b0p (bitn (ins, 1))
      then if b0p (bitn (ins, 0))  then halt ('reset-privileged, s)
            else nop-ins (s) endif
      elseif b0p (bitn (ins, 0))  then halt ('stop-privileged, s)
      else halt ('rte-privileged, s) endif
elseif b0p (bitn (ins, 1))
then if b0p (bitn (ins, 0))  then rtd-ins (s)
      else rts-ins (s) endif
elseif b0p (bitn (ins, 0))  then trapv-ins (s)
else rtr-ins (s) endif
```

JMP instruction. The JMP instruction is unsized. To calculate the effective address by effec-addr, one can arbitrarily supply the operand length. Note that the addr-predec, addr-postinc and immediate are not allowed.

DEFINITION:
jmp-addr-modep (*ins*) = control-addr-modep (s_mode (*ins*), s_rn (*ins*))

JMP does not affect CCR!

DEFINITION:
```
jmp-mapping($addr$, $s$)
    =
if mc-haltp ($s$)  then $s$
else update-pc ($addr$, $s$) endif
```

DEFINITION:
```
jmp-ins ($ins$, $s$)
    =
if jmp-addr-modep ($ins$)
then let $s\&addr$  be  effec-addr (L, s_mode ($ins$), s_rn ($ins$), $s$)
       in
       jmp-mapping(cddr ($s\&addr$), car ($s\&addr$)) endlet
else halt (MODE-SIGNAL, $s$) endif
```

JSR instruction. JSR does not affect CCR!

DEFINITION:
jsr-addr-modep (*ins*) = control-addr-modep (s_mode (*ins*), s_rn (*ins*))

DEFINITION:
```
jsr-ins ($ins$, $s$)
    =
if jsr-addr-modep ($ins$)
then let $s\&addr$  be  effec-addr (L, s_mode ($ins$), s_rn ($ins$), $s$)
       in
       if mc-haltp (car ($s\&addr$))  then car ($s\&addr$)
       else jmp-mapping(cddr ($s\&addr$),
                        push-sp (LSZ,
                                 mc-pc (car ($s\&addr$)),
                                 car ($s\&addr$))) endif endlet
else halt (MODE-SIGNAL, $s$) endif
```

NOT instruction.

DEFINITION:
```
not-addr-modep ($ins$)
    =
(data-addr-modep (s_mode ($ins$), s_rn ($ins$))
   $\wedge$
 alterable-addr-modep (s_mode ($ins$), s_rn ($ins$)))
```

DEFINITION:
```
not-z (opd)
    =
if opd = 0 then B0
else B1 endif
```

DEFINITION:
```
not-n (oplen, opd)
    =
if opd < exp (2, oplen − 1) then B1
else B0 endif
```

DEFINITION:
```
not-cvznx (oplen, opd, ccr)
    =
cvznx (B0, B0, not-z (opd), not-n (oplen, opd), ccr-x (ccr))
```

DEFINITION:
```
not-effect (oplen, opd, ccr)
    =
cons (lognot (oplen, opd), not-cvznx (oplen, opd, ccr))
```

DEFINITION:
```
not-ins (oplen, ins, s)
    =
if not-addr-modep (ins)
then let s&addr be mc-instate (oplen, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else mapping (oplen,
                         not-effect (oplen,
                                        operand (oplen, cdr (s&addr), s),
                                        mc-ccr (s)),
                         s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

DEFINITION:
```
not-subgroup (ins, s)
    =
if op-len (ins) = Q then halt ('move-to-sr-privileged, s)
else not-ins (op-len (ins), ins, s) endif
```

Opcode 0100. The miscellaneous instruction group includes LEA, CLR, MOVE from CCR, NEG, MOVE to CCR, NOT, SWAP, PEA, EXT-word, MOVEM to EA, TST, TAS, MOVEM to RN, LINK, UNLK, NOP, RTD, RTS, RTR, JSR, JMP.

DEFINITION:
```
misc-group (ins, s)
    =
if b0p (bitn (ins, 8))
then if b0p (bitn (ins, 11))
      then if b0p (bitn (ins, 10))
            then if b0p (bitn (ins, 9)) then negx-subgroup (ins, s)
                  else clr-subgroup (ins, s) endif
            elseif b0p (bitn (ins, 9)) then neg-subgroup (ins, s)
            else not-subgroup (ins, s) endif
      elseif b0p (bitn (ins, 10))
      then if b0p (bitn (ins, 9))
            then if b0p (bitn (ins, 7))
                  then if b0p (bitn (ins, 6))
                        then if b0p (bitn (ins, 5))
                                   ∧
                                b0p (bitn (ins, 4))
                                   ∧
                                b1p (bitn (ins, 3)) then link_l-ins (s_rn (ins), s)
                              else halt ('nbcd-unspecified, s) endif
                        else pea-subgroup (ins, s) endif
                  else ext-subgroup (ins, s) endif
            else tst-subgroup (ins, s) endif
      elseif b0p (bitn (ins, 9)) then movem-ea-rn-subgroup (ins, s)
      elseif b0p (bitn (ins, 7))
      then if b0p (bitn (ins, 6)) then halt (RESERVED-SIGNAL, s)
            elseif b0p (bitn (ins, 5)) then unlk-subgroup (ins, s)
            elseif b0p (bitn (ins, 4)) then halt ('move-usp-unspecified, s)
            elseif b0p (bitn (ins, 3)) then nop-subgroup (ins, s)
            else halt ('movec-unspecified, s) endif
      elseif b0p (bitn (ins, 6)) then jsr-ins (ins, s)
      else jmp-ins (ins, s) endif
elseif b1p (bitn (ins, 6)) ∧ b1p (bitn (ins, 7)) then lea-subgroup (ins, s)
else halt ('chk-unspecified, s) endif
```

Some useful definitions for Bcc and Scc instruction groups. Notice that bvs has been defined in TRAPV.

DEFINITION: bcs (c) = fix-bit (c)

DEFINITION: beq (z) = fix-bit (z)

DEFINITION: bmi (n) = fix-bit (n)

DEFINITION:
ble (v, z, n) = b-or $(z$, b-or (b-and $(n$, b-not $(v))$, b-and (b-not (n), $v)))$

DEFINITION:
bgt $(v,\ z,\ n)$ = b-and (b-or (b-and $(n,\ v)$, b-and (b-not (n), b-not (v))), b-not (z))

DEFINITION:
blt $(v,\ n)$ = b-or (b-and (n, b-not (v)), b-and (b-not (n), v))

DEFINITION:
bge $(v,\ n)$ = b-or (b-and $(n,\ v)$, b-and (b-not (n), b-not (v)))

DEFINITION: bls $(c,\ z)$ = b-or $(c,\ z)$

DEFINITION: bhi $(c,\ z)$ = b-and (b-not (c), b-not (z))

DEFINITION:
branch-cc $(cond,\ ccr)$

```
    =
if cond < 8
then if cond < 4
       then if cond < 2
              then if cond = 0 then B1
                     else B0 endif
              elseif cond = 2 then bhi (ccr-c (ccr), ccr-z (ccr))
              else bls (ccr-c (ccr), ccr-z (ccr)) endif
       elseif cond < 6
       then if cond = 4 then b-not (bcs (ccr-c (ccr)))
              else bcs (ccr-c (ccr)) endif
       elseif cond = 6 then b-not (beq (ccr-z (ccr)))
       else beq (ccr-z (ccr)) endif
elseif cond < 12
then if cond < 10
       then if cond = 8 then b-not (bvs (ccr-v (ccr)))
              else bvs (ccr-v (ccr)) endif
       elseif cond = 10 then b-not (bmi (ccr-n (ccr)))
       else bmi (ccr-n (ccr)) endif
elseif cond < 14
then if cond = 12 then bge (ccr-v (ccr), ccr-n (ccr))
       else blt (ccr-v (ccr), ccr-n (ccr)) endif
elseif cond = 14 then bgt (ccr-v (ccr), ccr-z (ccr), ccr-n (ccr))
else ble (ccr-v (ccr), ccr-z (ccr), ccr-n (ccr)) endif
```

BSR instruction.

DEFINITION:
bsr-ins $(pc,\ disp,\ s)$ = push-sp (LSZ, pc, update-pc (add (L, mc-pc (s), $disp$), s))

Bcc and BRA instructions are specified as follows. The BSR instruction needs some auxiliary functions to specify it. We define BRA and Bcc together. Since 0000 is always true.

DEFINITION:

```
bcc-ra-sr (pc, cond, disp, s)
    =
if cond = 0 then update-pc (add (L, mc-pc (s), disp), s)
elseif cond = 1 then bsr-ins (pc, disp, s)
elseif b0p (branch-cc (cond, mc-ccr (s))) then update-pc (pc, s)
else update-pc (add (L, mc-pc (s), disp), s) endif
```

Opcode 0110. The Bcc instruction group includes Bcc, BRA and BSR instructions.

DEFINITION:

```
bcc-group (disp, ins, s)
    =
if disp = 0
then if pc-word-readp (mc-pc (s), mc-mem(s))
       then bcc-ra-sr (add (L, mc-pc (s), WSZ),
                        cond-field (ins),
                        ext (W, pc-word-read (mc-pc (s), mc-mem(s)), L),
                        s)
       else halt (PC-SIGNAL, s) endif
elseif disp = 255
then if pc-long-readp (mc-pc (s), mc-mem (s))
       then bcc-ra-sr (add (L, mc-pc (s), LSZ),
                        cond-field (ins),
                        pc-long-read (mc-pc (s), mc-mem (s)),
                        s)
       else halt (PC-SIGNAL, s) endif
else bcc-ra-sr (mc-pc (s), cond-field (ins), ext (B, disp, L), s) endif
```

Scc instruction.

DEFINITION:

```
scc-addr-modep (ins)
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

CCR is not affected by Scc.

DEFINITION:

```
scc-effect (cond, ccr)
   =
cons (if b0p (branch-cc (cond, ccr)) then 0
      else 255 endif,
      ccr)
```

DEFINITION:

```
scc-ins (ins, s)
   =
if scc-addr-modep (ins)
then let s&addr be mc-instate (B, ins, s)
      in
      if mc-haltp (car (s&addr)) then car (s&addr)
      else mapping (B,
                    scc-effect (cond-field (ins), mc-ccr (s)),
                    s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

DBcc instruction.

DEFINITION:

```
dbcc-loop (rn, s)
   =
let cnt be sub (W, 1, read-dn (W, rn, s))
in
if nat-to-int (cnt, W) = -1
then update-pc (add (L, mc-pc (s), WSZ), write-dn (W, cnt, rn, s))
else update-pc (add (L,
                     mc-pc (s),
                     ext (W, pc-word-read (mc-pc (s), mc-mem (s)), L)),
                write-dn (W, cnt, rn, s)) endif endlet
```

DEFINITION:

```
dbcc-ins (ins, s)
   =
if pc-word-readp (mc-pc (s), mc-mem (s))
then if b0p (branch-cc (cond-field (ins), mc-ccr (s)))
      then dbcc-loop (s_rn (ins), s)
      else update-pc (add (L, mc-pc (s), WSZ), s) endif
else halt (PC-SIGNAL, s) endif
```

ADDQ instruction.

DEFINITION:

```
addq-addr-modep (oplen, ins)
```

```
    =
(alterable-addr-modep (s_mode (ins), d_mode (ins))
     ∧
 (¬ byte-an-direct-modep (oplen, s_mode (ins))))
```

It seems to us that there is no difference between word and long word operations for the ADDQ instruction in the address register direct mode.

DEFINITION:

```
addq-ins (oplen, ins, s)
    =
if addq-addr-modep (oplen, ins)
then if an-direct-modep (s_mode (ins))
       then write-an (L,
                      add (L, read-an (L, s_rn (ins), s), i-data (d_rn (ins))),
                      s_rn (ins),
                      s)
       else add-mapping (i-data (d_rn (ins)), oplen, ins, s) endif
else halt (MODE-SIGNAL, s) endif
```

SUBQ instruction. Same remark as for ADDQ.

DEFINITION:

```
subq-addr-modep (oplen, ins)
    =
(alterable-addr-modep (s_mode (ins), d_mode (ins))
     ∧
 (¬ byte-an-direct-modep (oplen, s_mode (ins))))
```

DEFINITION:

```
subq-ins (oplen, ins, s)
    =
if subq-addr-modep (oplen, ins)
then if an-direct-modep (s_mode (ins))
       then write-an (L,
                      sub (L, i-data (d_rn (ins)), read-an (L, s_rn (ins), s)),
                      s_rn (ins),
                      s)
       else sub-mapping (i-data (d_rn (ins)), oplen, ins, s) endif
else halt (MODE-SIGNAL, s) endif
```

Opcode 0101. The Scc instruction group includes Scc, DBcc, ADDQ, and SUBQ instructions.

DEFINITION:

```
scc-group (ins, s)
```

```
    =
if op-len (ins) = Q
then if s_mode (ins) = 1 then dbcc-ins (ins, s)
        elseif s_mode (ins) = 7 then halt ('trapcc-unspecified, s)
        else scc-ins (ins, s) endif
elseif b0p (bitn (ins, 8)) then addq-ins (op-len (ins), ins, s)
else subq-ins (op-len (ins), ins, s) endif
```

Opcode 0111. MOVEQ instruction.

DEFINITION:

```
moveq-ins (ins, s)
    =
if b0p (bitn (ins, 8))
then d-mapping (L, move-effect (L, ext (B, head (ins, B), L), mc-ccr (s)), d_rn (ins), s)
else halt (RESERVED-SIGNAL, s) endif
```

CMP instruction.

DEFINITION:

```
cmp-cvznx (oplen, sopd, dopd, ccr)
    =
cvznx (sub-c (oplen, sopd, dopd),
         sub-v (oplen, sopd, dopd),
         sub-z (oplen, sopd, dopd),
         sub-n (oplen, sopd, dopd),
         ccr-x (ccr))
```

DEFINITION:

```
cmp-addr-modep (oplen, ins)
    =
(addr-modep (s_mode (ins), s_rn (ins))
     ∧
 (¬ byte-an-direct-modep (oplen, s_mode (ins))))
```

The execution of the CMP instruction.

DEFINITION:

```
cmp-ins (oplen, ins, s)
    =
if cmp-addr-modep (oplen, ins)
then let s&addr be mc-instate (oplen, ins, s)
        in
        if mc-haltp (car (s&addr)) then car (s&addr)
        else update-ccr (cmp-cvznx (oplen,
                                            operand (oplen,
                                                  cdr ($s\&addr$),
                                                  car ($s\&addr$)),
                                            read-dn ($oplen$, d_rn ($ins$), $s$),
                                            mc-ccr ($s$)),
                                      car ($s\&addr$)) endif endlet
          else halt (MODE-SIGNAL, $s$) endif
```

CMPA instruction.

DEFINITION:
cmpa-addr-modep (ins) = addr-modep (s_mode (ins), s_rn (ins))

The cvznx-flag setting is the same as the CMP instruction. The only difference is that word operation is sign-extended to longword operation.

DEFINITION:

```
cmpa-ins ($oplen$, $ins$, $s$)
  =
if cmpa-addr-modep ($ins$)
then let $s\&addr$ be mc-instate ($oplen$, $ins$, $s$)
     in
     if mc-haltp (car ($s\&addr$)) then car ($s\&addr$)
     else update-ccr (cmp-cvznx (L,
                                  ext ($oplen$,
                                       operand ($oplen$, cdr ($s\&addr$), $s$),
                                       L),
                                  read-an (L, d_rn ($ins$), $s$),
                                  mc-ccr ($s$)),
                            car ($s\&addr$)) endif endlet
else halt (MODE-SIGNAL, $s$) endif
```

EOR instruction.

DEFINITION:

```
eor-z ($sopd$, $dopd$)
  =
if $sopd = dopd$ then B1
else B0 endif
```

DEFINITION:
eor-n ($oplen$, $sopd$, $dopd$) = b-eor (bitn ($sopd$, $oplen - 1$), bitn ($dopd$, $oplen - 1$))

DEFINITION:

```
eor-cvznx ($oplen$, $sopd$, $dopd$, $ccr$)
  =
cvznx (B0, B0, eor-z ($sopd$, $dopd$), eor-n ($oplen$, $sopd$, $dopd$), ccr-x ($ccr$))
```

```
DEFINITION:
eor-effect (oplen, sopd, dopd, ccr)
    =
cons (logeor (sopd, dopd), eor-cvznx (oplen, sopd, dopd, ccr))

DEFINITION:
eor&eori-addr-modep (ins)
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))

DEFINITION:
eor-mapping (sopd, oplen, ins, s)
    =
let s&addr  be  mc-instate (oplen, ins, s)
in
if mc-haltp (car (s&addr))  then car (s&addr)
else mapping (oplen,
                 eor-effect (oplen,
                             sopd,
                             operand (oplen, cdr (s&addr), s),
                             mc-ccr (s)),
                 s&addr) endif endlet

DEFINITION:
eor-ins (oplen, ins, s)
    =
if eor&eori-addr-modep (ins)
then eor-mapping (read-dn (oplen, d_rn (ins), s), oplen, ins, s)
else halt (MODE-SIGNAL, s) endif
```

CMPM instruction.

```
DEFINITION:
cmpm-mapping (addr, oplen, ins, s)
    =
let s&addr  be  addr-postinc (oplen, d_rn (ins), s)
in
if read-memp (cddr (s&addr), mc-mem (s), op-sz (oplen))
then update-ccr (cmp-cvznx (oplen,
                               operand (oplen, addr, s),
                               operand (oplen, cdr (s&addr), s),
                               mc-ccr (s)),
                    car (s&addr))
else halt (READ-SIGNAL, s) endif endlet
```

DEFINITION:
```
cmpm-ins($oplen$, $ins$, $s$)
    =
let $s\&addr$ be addr-postinc($oplen$, s_rn($ins$), $s$)
in
if read-memp(cddr($s\&addr$), mc-mem($s$), op-sz($oplen$))
then cmpm-mapping(cdr($s\&addr$), $oplen$, $ins$, car($s\&addr$))
else halt(READ-SIGNAL, $s$) endif endlet
```

Opcode 1011. The CMP instruction group includes instructions CMP, CMPA, EOR, and CMPM.

DEFINITION:
```
cmp-group($oplen$, $ins$, $s$)
    =
if b0p(bitn($ins$, 8))
then if $oplen$ = Q then cmpa-ins(W, $ins$, $s$)
       else cmp-ins($oplen$, $ins$, $s$) endif
elseif $oplen$ = Q then cmpa-ins(L, $ins$, $s$)
elseif s_mode($ins$) = 1 then cmpm-ins($oplen$, $ins$, $s$)
else eor-ins($oplen$, $ins$, $s$) endif
```

MOVEP instruction. MOVEP moves a data register into alternate bytes of memory.

DEFINITION:
```
movep-writep($addr$, $mem$, $n$)
    =
if $n \simeq 0$ then t
else byte-writep(add(L, $addr$, $2 * (n - 1)$), $mem$)
          $\wedge$
       movep-writep($addr$, $mem$, $n - 1$) endif
```

DEFINITION:
```
movep-write($value$, $addr$, $mem$, $n$)
    =
if $n \simeq 0$ then $mem$
else movep-write(tail($value$, B),
                 $addr$,
                 byte-write($value$, add(L, $addr$, $2 * (n - 1)$), $mem$),
                 $n - 1$) endif
```

DEFINITION:
```
movep-to-mem($addr$, $oplen$, $ins$, $s$)
    =
if movep-writep($addr$, mc-mem($s$), op-sz($oplen$))
```

```
then update-mem (movep-write (read-dn (oplen, d_rn (ins), s),
                              addr,
                              mc-mem (s),
                              op-sz (oplen)),
                 s)
else halt (WRITE-SIGNAL, s) endif
```

MOVEP moves alternate bytes in memory into a data register.

DEFINITION:
```
movep-readp (addr, mem, n)
   =
if n ≃ 0 then t
else byte-readp (addr, mem) ∧ movep-readp (add (L, addr, WSZ), mem, n − 1) endif
```

DEFINITION:
```
movep-read (addr, mem, n)
   =
if n ≃ 0 then 0
else app (B,
          byte-read (add (L, addr, 2 * (n − 1)), mem),
          movep-read (addr, mem, n − 1)) endif
```

DEFINITION:
```
movep-to-reg (addr, oplen, ins, s)
   =
if movep-readp (addr, mc-mem (s), op-sz (oplen))
then write-dn (oplen, movep-read (addr, mc-mem (s), op-sz (oplen)), d_rn (ins), s)
else halt (READ-SIGNAL, s) endif
```

DEFINITION: evenp (x) = b0p (bcar (x))

DEFINITION:
```
movep-addr (s&addr)
   =
if evenp (cddr (s&addr)) then cddr (s&addr)
else add (L, cddr (s&addr), BSZ) endif
```

DEFINITION:
```
movep-ins (opmode, ins, s)
   =
let s&addr be addr-disp (mc-pc (s), s_rn (ins), s)
in
if mc-haltp (car (s&addr)) then car (s&addr)
elseif opmode < 6
```

```
then if $opmode = 4$
        then movep-to-reg (movep-addr ($s\&addr$), W, $ins$, car ($s\&addr$))
        else movep-to-reg (movep-addr ($s\&addr$), L, $ins$, car ($s\&addr$)) endif
elseif $opmode = 6$
then movep-to-mem (movep-addr ($s\&addr$), W, $ins$, car ($s\&addr$))
else movep-to-mem (movep-addr ($s\&addr$), L, $ins$, car ($s\&addr$)) endif endlet
```

Some functions for bit operations.

DEFINITION:
```
bxxx-oplen ($smode$)
    =
if dn-direct-modep ($smode$)  then L
else B endif
```

DEFINITION:
```
bxxx-num ($smode$, $bnum$)
    =
if dn-direct-modep ($smode$)  then head ($bnum$, 5)
else head ($bnum$, 3) endif
```

DEFINITION:
```
bxxx-opd ($smode$, $s\&addr$)
    =
if dn-direct-modep ($smode$)  then read-dn (L, cddr ($s\&addr$), car ($s\&addr$))
else byte-read (cddr ($s\&addr$), mc-mem (car ($s\&addr$))) endif
```

BCHG instruction.

DEFINITION:
```
bchg-addr-modep ($ins$)
    =
(data-addr-modep (s_mode ($ins$), s_rn ($ins$))
     $\land$
 alterable-addr-modep (s_mode ($ins$), s_rn ($ins$)))
```

DEFINITION:
```
bchg-effect ($bnum$, $opd$, $ccr$)
    =
cons (setn ($opd$, $bnum$, b-not (bitn ($opd$, $bnum$))), set-z (b-not (bitn ($opd$, $bnum$)), $ccr$))
```

DEFINITION:
```
bchg-ins ($bnum$, $ins$, $s$)
    =
if bchg-addr-modep ($ins$)
then let $s\&addr$  be  mc-instate (B, $ins$, $s$)
```

```
        in
        if mc-haltp (car (s&addr))  then car (s&addr)
        else mapping (bxxx-oplen (s_mode (ins)),
                      bchg-effect (bxxx-num (s_mode (ins), bnum),
                                   bxxx-opd (s_mode (ins), s&addr),
                                   mc-ccr (s)),
                      s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

BCLR instruction.

DEFINITION:
```
bclr-addr-modep (ins)
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
    ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

DEFINITION:
```
bclr-effect (bnum, opd, ccr)
    =
cons (setn (opd, bnum, B0), set-z (b-not (bitn (opd, bnum)), ccr))
```

DEFINITION:
```
bclr-ins (bnum, ins, s)
    =
if bclr-addr-modep (ins)
then let s&addr  be  mc-instate (B, ins, s)
        in
        if mc-haltp (car (s&addr))  then car (s&addr)
        else mapping (bxxx-oplen (s_mode (ins)),
                      bclr-effect (bxxx-num (s_mode (ins), bnum),
                                   bxxx-opd (s_mode (ins), s&addr),
                                   mc-ccr (s)),
                      s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

BSET instruction.

DEFINITION:
```
bset-addr-modep (ins)
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
    ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

DEFINITION:

```
bset-effect (bnum, opd, ccr)
    =
cons (setn (opd, bnum, B1), set-z (b-not (bitn (opd, bnum)), ccr))
```

DEFINITION:

```
bset-ins (bnum, ins, s)
    =
if bset-addr-modep (ins)
then let s&addr be mc-instate (B, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else mapping (bxxx-oplen (s_mode (ins)),
                     bset-effect (bxxx-num (s_mode (ins), bnum),
                                  bxxx-opd (s_mode (ins), s&addr),
                                  mc-ccr (s)),
                     s&addr) endif endlet
else halt (MODE-SIGNAL, s) endif
```

BTST instruction.

DEFINITION:

```
btst-addr-modep (ins) = data-addr-modep (s_mode (ins), s_rn (ins))
```

DEFINITION:

```
btst-ins (bnum, ins, s)
    =
if btst-addr-modep (ins)
then let s&addr be mc-instate (B, ins, s)
       in
       if mc-haltp (car (s&addr)) then car (s&addr)
       else update-ccr (set-z (b-not (bitn (bxxx-opd (s_mode (ins),
                                                       s&addr),
                                             bxxx-num (s_mode (ins), bnum))),
                              mc-ccr (s)),
                        car (s&addr)) endif endlet
else halt (MODE-SIGNAL, s) endif
```

'bit-ins' includes the BTST, BCLR, BCHG, and BSET instructions.

DEFINITION:

```
bit-ins (bnum, ins, s)
    =
let type be bits (ins, 6, 7)
in
```

```
if $type < 2$
then if $type = 0$ then btst-ins ($bnum$, $ins$, $s$)
     else bchg-ins ($bnum$, $ins$, $s$) endif
elseif $type = 2$ then bclr-ins ($bnum$, $ins$, $s$)
else bset-ins ($bnum$, $ins$, $s$) endif endlet
```

Dynamic bit operation. BTST, BCLR, BCHG, and BSET instructions.

DEFINITION:
```
d-bit-subgroup ($ins$, $s$)
  =
if s_mode ($ins$) = 1 then movep-ins (opmode-field ($ins$), $ins$, $s$)
else bit-ins (read-dn (L, d_rn ($ins$), $s$), $ins$, $s$) endif
```

Static bit operation. BTST, BCLR, BCHG, and BSET instructions.

DEFINITION:
```
s-bit-subgroup ($ins$, $s$)
  =
if pc-word-readp (mc-pc ($s$), mc-mem ($s$))
then if pc-byte-read (mc-pc ($s$), mc-mem ($s$)) = 0
     then bit-ins (pc-byte-read (add (L, mc-pc ($s$), BSZ), mc-mem ($s$)),
                   $ins$,
                   update-pc (add (L, mc-pc ($s$), WSZ), $s$))
     else halt (RESERVED-SIGNAL, $s$) endif
else halt (PC-SIGNAL, $s$) endif
```

ORI instruction.

DEFINITION:
```
ori-addr-modep ($ins$)
  =
(data-addr-modep (s_mode ($ins$), s_rn ($ins$))
   $\land$
 alterable-addr-modep (s_mode ($ins$), s_rn ($ins$)))
```

DEFINITION:
```
ori-ins ($oplen$, $ins$, $s$)
  =
let $s\&idata$ be immediate ($oplen$, mc-pc ($s$), $s$)
in
if mc-haltp (car ($s\&idata$)) then car ($s\&idata$)
elseif ori-addr-modep ($ins$)
then or-mapping (cddr ($s\&idata$), $oplen$, $ins$, car ($s\&idata$))
else halt (MODE-SIGNAL, $s$) endif endlet
```

ORI to CCR instruction.

DEFINITION:
```
ori-to-ccr-ins (pc, s)
    =
if pc-word-readp (pc, mc-mem (s))
then if pc-byte-read (pc, mc-mem (s)) = 0
       then update-ccr (logor (pc-byte-read (add (L, pc, BSZ), mc-mem (s)), mc-ccr (s)),
                        update-pc (add (L, pc, WSZ), s))
       else halt (RESERVED-SIGNAL, s) endif
else halt (PC-SIGNAL, s) endif
```

ORI and ORI to CCR instructions. Detect ORI to SR, CMP2, and CHK2.

DEFINITION:
```
ori-subgroup (oplen, ins, s)
    =
if oplen = Q then halt ('cmp2-chk2-unspecified, s)
elseif head (ins, 6) = 60
then if oplen = B then ori-to-ccr-ins (mc-pc (s), s)
       elseif oplen = W then halt ('ori-to-sr-privileged, s)
       else halt (RESERVED-SIGNAL, s) endif
else ori-ins (oplen, ins, s) endif
```

ANDI instruction.

DEFINITION:
```
andi-addr-modep (ins)
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

DEFINITION:
```
andi-ins (oplen, ins, s)
    =
let s&idata be immediate (oplen, mc-pc (s), s)
in
if mc-haltp (s) then car (s&idata)
elseif andi-addr-modep (ins)
then and-mapping (cddr (s&idata), oplen, ins, car (s&idata))
else halt (MODE-SIGNAL, s) endif endlet
```

ANDI to CCR instruction.

DEFINITION:
```
andi-to-ccr-ins (pc, s)
    =
if pc-word-readp (pc, mc-mem (s))
then if pc-byte-read (pc, mc-mem(s)) = 0
        then update-ccr (logand (pc-byte-read (add (L, pc, BSZ), mc-mem (s)), mc-ccr (s)),
                         update-pc (add (L, pc, WSZ), s))
        else halt (RESERVED-SIGNAL, s) endif
else halt (PC-SIGNAL, s) endif
```

ANDI and ANDI to CCR instructions. Detect ANDI to SR, CMP2 and CHK2.

DEFINITION:

```
andi-subgroup (oplen, ins, s)
    =
if oplen = Q then halt ('cmp2-chk2-unspecified, s)
elseif head (ins, 6) = 60
then if oplen = B then andi-to-ccr-ins (mc-pc (s), s)
        elseif oplen = W then halt ('andi-to-sr-unspecified, s)
        else halt (RESERVED-SIGNAL, s) endif
else andi-ins (oplen, ins, s) endif
```

SUBI instruction. Detect CMP2 and CHK2.

DEFINITION:

```
subi-addr-modep (ins)
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

DEFINITION:

```
subi-ins (oplen, ins, s)
    =
let s&idata be immediate (oplen, mc-pc (s), s)
in
if mc-haltp (car (s&idata)) then car (s&idata)
elseif subi-addr-modep (ins)
then sub-mapping (cddr (s&idata), oplen, ins, car (s&idata))
else halt (MODE-SIGNAL, s) endif endlet
```

DEFINITION:

```
subi-subgroup (oplen, ins, s)
    =
if oplen = Q then halt ('cmp2-chk2-unspecified, s)
else subi-ins (oplen, ins, s) endif
```

ADDI instruction. Detect RTM and CALLM.

DEFINITION:
```
addi-addr-modep (ins)
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 alterable-addr-modep (s_mode (ins), s_rn (ins)))
```

DEFINITION:
```
addi-ins (oplen, ins, s)
    =
let s&idata be immediate (oplen, mc-pc (s), s)
in
if mc-haltp (car (s&idata)) then car (s&idata)
elseif addi-addr-modep (ins)
then add-mapping (cddr (s&idata), oplen, ins, car (s&idata))
else halt (MODE-SIGNAL, s) endif endlet
```

DEFINITION:
```
addi-subgroup (oplen, ins, s)
    =
if oplen = Q then halt ('rtm-callm-unspecified, s)
else addi-ins (oplen, ins, s) endif
```

EORI instruction.

DEFINITION:
```
eori-ins (oplen, ins, s)
    =
let s&idata be immediate (oplen, mc-pc (s), s)
in
if mc-haltp (car (s&idata)) then car (s&idata)
elseif eor&eori-addr-modep (ins)
then eor-mapping (cddr (s&idata), oplen, ins, car (s&idata))
else halt (MODE-SIGNAL, s) endif endlet
```

EORI to CCR instruction.

DEFINITION:
```
eori-to-ccr-ins (pc, s)
    =
if pc-word-readp (pc, mc-mem (s))
then if pc-byte-read (pc, mc-mem(s)) = 0
       then update-ccr (logeor (pc-byte-read (add (L, pc, BSZ), mc-mem(s)), mc-ccr (s)),
                        update-pc (add (L, pc, WSZ), s))
       else halt (RESERVED-SIGNAL, s) endif
else halt (PC-SIGNAL, s) endif
```

EORI and EORI to CCR instructions. Detect EORI to SR, CAS and CAS2 instructions!

DEFINITION:

```
eori-subgroup (oplen, ins, s)
    =
if oplen = Q then halt ('cas-cas2-unspecified, s)
elseif head (ins, 6) = 60
then if oplen = B then eori-to-ccr-ins (mc-pc (s), s)
      elseif oplen = W then halt ('eori-to-sr-unspecified, s)
      else halt (RESERVED-SIGNAL, s) endif
else eori-ins (oplen, ins, s) endif
```

CMPI instruction.

DEFINITION:

```
cmpi-addr-modep (ins)
    =
(data-addr-modep (s_mode (ins), s_rn (ins))
     ∧
 (¬ idata-modep (s_mode (ins), s_rn (ins))))
```

DEFINITION:

```
cmpi-mapping(idata, oplen, ins, s)
    =
let s&addr be mc-instate(oplen, ins, s)
in
if mc-haltp (car (s&addr)) then car (s&addr)
else update-ccr (cmp-cvznx (oplen,
                            idata,
                            operand (oplen, cdr (s&addr), s),
                            mc-ccr (s)),
                 car (s&addr)) endif endlet
```

DEFINITION:

```
cmpi-ins (oplen, ins, s)
    =
let s&idata be immediate(oplen, mc-pc (s), s)
in
if mc-haltp (car (s&idata)) then car (s&idata)
elseif cmpi-addr-modep (ins)
then cmpi-mapping(cddr (s&idata), oplen, ins, car (s&idata))
else halt (MODE-SIGNAL, s) endif endlet
```

The CMPI subgroup includes only the CMPI instruction. Detect CAS and CAS2 instructions!

Definition:
```
cmpi-subgroup (oplen, ins, s)
  =
if oplen = Q then halt ('cas-cas2-unspecified, s)
else cmpi-ins (oplen, ins, s) endif
```

Opcode 0000. This instruction group includes instructions ORI, ORI to CCR, BTST, BCLR, BCHG, BSET, MOVEP, ANDI, ANDI to CCR, SUBI, ADDI, EORI, EORI to CCR, CMPI.

Definition:
```
bit-group (ins, s)
  =
if b0p (bitn (ins, 8))
then if b0p (bitn (ins, 11))
     then if b0p (bitn (ins, 10))
          then if b0p (bitn (ins, 9)) then ori-subgroup (op-len (ins), ins, s)
               else andi-subgroup (op-len (ins), ins, s) endif
          elseif b0p (bitn (ins, 9)) then subi-ins (op-len (ins), ins, s)
          else addi-subgroup (op-len (ins), ins, s) endif
     elseif b0p (bitn (ins, 10))
     then if b0p (bitn (ins, 9)) then s-bit-subgroup (ins, s)
          else eori-subgroup (op-len (ins), ins, s) endif
     elseif b0p (bitn (ins, 9)) then cmpi-subgroup (op-len (ins), ins, s)
     else halt ('moves-cas-cas2-unspecified, s) endif
else d-bit-subgroup (ins, s) endif
```

The opcode field.

Definition: opcode-field (*ins*) = bits (*ins*, 12, 15)

Execute the current instruction. See Table 3-14 of [4] for this classification.

Definition:
```
execute-ins (ins, s)
  =
let opcode be opcode-field (ins)
in
if opcode < 8
then if opcode < 4
     then if opcode < 2
          then if opcode = 0 then bit-group (ins, s)
               else move-ins (B, ins, s) endif
          elseif opcode = 2 then move-group (L, ins, s)
          else move-group (W, ins, s) endif
     elseif opcode < 6
```

```
          then if $opcode$ = 4 then misc-group ($ins$, $s$)
                   else scc-group ($ins$, $s$) endif
          elseif $opcode$ = 6 then bcc-group (head ($ins$, B), $ins$, $s$)
          else moveq-ins ($ins$, $s$) endif
elseif $opcode$ < 12
then if $opcode$ < 10
          then if $opcode$ = 8 then or-group (op-len ($ins$), $ins$, $s$)
                   else sub-group (opmode-field ($ins$), $ins$, $s$) endif
          elseif $opcode$ = 10 then halt (RESERVED-SIGNAL, $s$)
          else cmp-group (op-len ($ins$), $ins$, $s$) endif
elseif $opcode$ < 14
then if $opcode$ = 12 then and-group (op-len ($ins$), $ins$, $s$)
          else add-group (opmode-field ($ins$), $ins$, $s$) endif
elseif $opcode$ = 14 then s&r-group ($ins$, $s$)
else halt ('coprocessor-unspecified, $s$) endif endlet
```

'current-ins' is a function of two arguments, pc and s. pc is the current value of the program counter, and s is the current state. 'current-ins' returns the current instruction (a word, not including any possible extension words), that is, the word pointed to by pc. To determine what instruction we are to execute, this word may only provide partial information. Many instructions require that we examine subsequent words to determine what to do. But to figure out how many words we need, we must start with the first word.

DEFINITION: current-ins (pc, s) = pc-word-read (pc, mc-mem(s))

13 Stepi and Stepn

'stepi' maps a machine state to the next machine state by executing the current instruction.

DEFINITION:
```
stepi ($s$)
    =
if evenp (mc-pc ($s$))
then if pc-word-readp (mc-pc ($s$), mc-mem($s$))
        then execute-ins (current-ins (mc-pc ($s$), $s$),
                              update-pc (add (L, mc-pc ($s$), WSZ), $s$))
        else halt (PC-SIGNAL, $s$) endif
else halt (PC-ODD-SIGNAL, $s$) endif
```

'stepn' is a function of two arguments: s is the current state of the machine, and n is the number of instructions to execute.

DEFINITION:
```
stepn ($s$, $n$)
    =
if mc-haltp ($s$) $\vee$ ($n \simeq 0$) then $s$
else stepn (stepi ($s$), $n - 1$) endif
```

14 Auxiliary Functions

This section contains some auxiliary functions which are not needed to define 'stepn' but are used only in the example of the next section. 'map-update' updates the map in the memory. The *map* is a binary tree with a list of keys in the key field. By updating the map we assign new properties to the memory.

DEFINITION:
```
cons-key-lst ($key$, $lst$)
    =
if $key \in lst$ then $lst$
else cons ($key$, $lst$) endif
```

DEFINITION:
```
key-field ($map$)
    =
if listp ($map$) then car ($map$)
else nil endif
```

DEFINITION:
```
make-map($key$, $map$)
    =
make-bt (cons-key-lst ($key$, key-field ($map$)), branch0 ($map$), branch1 ($map$))
```

Definition:

```
map-update(key, x, n, map)
  =
if $n \simeq 0$ then make-map(key, map)
elseif b0p(bitn(x, $n - 1$))
then make-bt(key-field(map),
             map-update(key, x, $n - 1$, branch0(map)),
             branch1(map))
else make-bt(key-field(map),
             branch0(map),
             map-update(key, x, $n - 1$, branch1(map))) endif
```

Load the values in the list into the memory starting from location *addr.*

Definition:

```
load-lst-mem(opsz, lst, addr, mem)
  =
if listp(lst)
then load-lst-mem(opsz,
                  cdr(lst),
                  add(32, addr, opsz),
                  write-mem(car(lst), addr, mem, opsz))
else mem endif
```

Event: For efficiency, compile those definitions not yet compiled.

Event: Make the library "mc20-1".

15 An Example of Simulation

Here is an utterly concrete theorem about ‘stepn.’ Roughly speaking, the theorem states that if ‘stepn’ executes 37 instructions starting in a state that contains machine code instructions for Euclid’s GCD algorithm in ROM and the integers 54 and 42 on the stack, then the correct answer, 6, is the value of data register d0 in the resulting state. This theorem has, of course, an utterly trivial proof: we just run ‘stepn’. We present this trivial theorem here only to illustrate setting up ‘stepn’ to run.

THEOREM: gcd-example

$((\textit{stack-pointer} = \texttt{EFFFE40}_{16})$
$\quad\wedge$
$(\textit{rfile} = \texttt{'(0 0 0 0 0 0 0 0 0 0 0 0 0 0 EFFFE4C}_{16}$
$\qquad\qquad \texttt{EFFFE40}_{16}\texttt{)})$
$\quad\wedge$
$(\textit{pc} = \texttt{22B6}_{16})$
$\quad\wedge$
$(\textit{ccr} = \texttt{0})$
$\quad\wedge$
$(\textit{gcd-code} = \texttt{'(78 86 0 0 72 231 48 0 36 46 0 8 38 46 0 12 74}$
$\qquad\qquad \texttt{130 103 28 74 131 102 4 32 2 96 22 182 130 108}$
$\qquad\qquad \texttt{8 76 67 40 0 36 0 96 232 76 66 56 0 38 0 96}$
$\qquad\qquad \texttt{224 32 3 76 238 0 12 255 248 78 94 78 117)})$
$\quad\wedge$
$(\textit{empty-memory} = \texttt{'((nil (nil (nil (nil (nil ((rom) nil)))))))})$
$\quad\wedge$
$(\textit{mem} = \text{load-lst-mem}(\texttt{4},$
$\qquad\qquad \texttt{'(22B2}_{16}\texttt{ 54 42)},$
$\qquad\qquad \textit{stack-pointer},$
$\qquad\qquad \text{load-lst-mem}(\texttt{1}, \textit{gcd-code}, \textit{pc}, \textit{empty-memory})))$
$\quad\wedge$
$(\textit{initial-state} = \text{mc-state}(\texttt{'running}, \textit{rfile}, \textit{pc}, \textit{ccr}, \textit{mem}))$
$\quad\wedge$
$(\textit{final-state} = \text{stepn}(\textit{initial-state}, \texttt{37})))$
$\quad\rightarrow$
$((\text{mc-status}(\textit{final-state}) = \texttt{'running})$
$\quad\wedge$
$(\text{mc-rfile}(\textit{final-state})$
$\qquad =$
$\texttt{'(6 0 0 0 0 0 0 0 0 0 0 0 0 0 EFFFE4C}_{16}\texttt{ EFFFE44}_{16}\texttt{)})$
$\quad\wedge$
$(\text{mc-pc}(\textit{final-state}) = \texttt{22B2}_{16}))$

Here is a paraphrase of the foregoing theorem. The specific numbers in the theorem are derived from the compilation of a C program for GCD and from the result of loading that program on a Sun-3.

- If
  1. *stack-pointer* = $\texttt{EFFFE40}_{16}$,
  2. the register file *rfile* is all `0`'s excepting for A6 and SP, which are $\texttt{EFFFE4C}_{16}$ and *stack-pointer*, respectively,
  3. the program counter *pc* = $\texttt{22B6}_{16}$ and the condition code register *ccr* = `0`,
  4. *gcd-code* is the long list of integers above beginning with `78` and ending with `117`,
  5. *empty-memory* is a pair representing a 32-bit wide memory which has a `0` byte at every address, which is of type ROM from address $\texttt{0}_{16}$ to address $\texttt{7FFFFFF}_{16}$, and which is of type RAM at all other addresses,
  6. *mem* is the result of first loading *gcd-code* into an empty memory at *pc* and then further loading the two natural numbers `54` and `42` and the return address of the caller ($\texttt{22B2}_{16}$) at the location pointed to by *stack-pointer*,
  7. *initial-state* is an mc-state whose five fields are `'running`, *rfile*, *pc*, *ccr*, and *mem*, respectively, and finally
  8. *final-state* is the result of running 'stepn' for `37` instructions starting with *initial-state*,
- then, if we examine *final-state*, we find:
  1. the machine is still `'running`,
  2. the register file is `'(6 0 0 0 0 0 0 0 0 0 0 0 0 0` $\texttt{EFFFE4C}_{16}$ $\texttt{EFFFE44}_{16}$`)`, observing that d0 is equal to `6`, the GCD of `54` and `42`, and
  3. the program counter is set to $\texttt{22B2}_{16}$, the return address to the caller.

This theorem should not be confused with the much more general theorem stating the correctness of the same GCD program on *all* input, a theorem whose mechanical proof is described in [3].

16 Acknowledgements

We would like to thank Bill Bevier, Don Good, Warren Hunt, Matt Kaufmann, J Moore, and Bill Schelter for their many constructive suggestions and discussions. Special thanks to Fay Goytowski for her meticulous reading of the MC68020 specification, which revealed a dozen or so errors. The general style of Nqthm formalization used in this MC68020 specification is the product of over a decade of study by the authors of Nqthm and their students. Especially influential was the FM8502 and Piton work [1]. The development of Nqthm was primarily supported by NSF, ONR, and DARPA.

17 Syntax Summary

Here is a summary of the conventional syntax used in this report in terms of the official syntax of the Nqthm logic described in [2]. ('cond' and 'let' are recent extensions not described in [2].)

1. Variables. x, y, z, etc. are printed in italics.

2. Function application. For any function symbol for which special syntax is not given below, an application of the symbol is printed with the usual notation; e.g., the term `(fn x y z)` is printed as $\mathrm{fn}(x,\ y,\ z)$. Note that the function symbol is printed in Roman. In the special case that 'c' is a function symbol of no arguments, i.e., it is a constant, the term `(c)` is printed merely as C, in small caps, with no trailing parentheses. Because variables are printed in italics, there is no confusion between the printing of variables and constants.

3. Other constants. **t**, **f**, and **nil** are printed in bold. Quoted constants are printed in the ordinary fashion of the Nqthm logic, e.g., `'(a b c)` is still printed just that way. `#b001` is printed as $\mathtt{001}_2$, `#o765` is printed as $\mathtt{765}_8$, and `#xA9` is printed as $\mathtt{A9}_{16}$.

4. `(if x y z)` is printed as

   if *x* **then** *y* **else** *z* **endif**.

5. `(cond (test1 value1) (test2 value2) (t value3))` is printed as

   if *test1* **then** *value1* **elseif** *test2* **then** *value2* **else** *value3* **endif**.

6. `(let ((var1 val1) (var2 val2)) form)` is printed as

   let *var1* **be** *val1*, *var2* **be** *val2* **in** *form* **endlet**.

7. The remaining function symbols that are printed specially are described in the following table.

Nqthm Syntax	Conventional Syntax
`(or x y)`	$x \vee y$
`(and x y)`	$x \wedge y$
`(times x y)`	$x * y$
`(plus x y)`	$x + y$
`(remainder x y)`	$x\ \mathbf{mod}\ y$
`(quotient x y)`	$x \div y$
`(difference x y)`	$x - y$
`(implies x y)`	$x \rightarrow y$
`(member x y)`	$x \in y$
`(geq x y)`	$x \geq y$
`(greaterp x y)`	$x > y$
`(leq x y)`	$x \leq y$
`(lessp x y)`	$x < y$
`(equal x y)`	$x = y$
`(not (member x y))`	$x \notin y$
`(not (geq x y))`	$x \not\geq y$
`(not (greaterp x y))`	$x \not> y$
`(not (leq x y))`	$x \not\leq y$
`(not (lessp x y))`	$x \not< y$
`(not (equal x y))`	$x \neq y$
`(minus x)`	$- x$
`(add1 x)`	$1 + x$
`(nlistp x)`	$x \simeq \mathbf{nil}$
`(zerop x)`	$x \simeq 0$
`(numberp x)`	$x \in \mathbf{N}$
`(sub1 x)`	$x - 1$
`(not (nlistp x))`	$x \not\simeq \mathbf{nil}$
`(not (zerop x))`	$x \not\simeq 0$
`(not (numberp x))`	$x \notin \mathbf{N}$

References

[1] William Bevier, J Strother Moore, Warren Hunt, and William Young. Special issue on system verification. *Journal of Automated Reasoning*, 5(4), 1989.

[2] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[3] Robert S. Boyer and Yuan Yu. Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor, Technical Report TR-91-33, University of Texas at Austin, to appear in the proceedings of the 11th International Conference on Automated Deduction, Lecture Notes in Computer Science, Springer-Verlag, 1992.

[4] Motorola, Inc. *MC68020 32-bit Microprocessor User's Manual*. Prentice Hall, New Jersey, 1989.

Index