# Micro-Operation Description Coding Guide

Version 0.5

2002/05/30

## 1. Basic of coding micro-operation description

When coding micro-operation description, you can use resource specified in the [Resource Declaration] to write the below operations for pipeline stages.

(1) Data transfer

(2) Process with resources (operation, reading, writing)

The data transfer is represented with symbol "=". This is a typical programming concept to assign the right side value to the left. When coding the process with resource, the code must be written in the order of <resource name> ", " <function name> " (" <input list> ")". For <function name>, you can use the function written with "Function Set". Use the Resource Declaration window to view the "Function Set". Following code shows the extraction from the "Function Set" of model alu.

```
model alu32{
        port {
        in a[31:0], b[31:0], cin, mode[4:0];
        out result[31:0], flag[3:0];
}
        ..........................................
/** add : signed add, flag(3):C, flag(2):Z, flag(1):S, flag(0):V */
function add{
input{
unsigned a, b;
}
output{
unsigned result = add(a,b);
bit_vector flag = alu_flag(mode, a, b, cin);
}
control{
in mode;
in cin;
}
protocol{
[mode == ”01001” && cin == ’0’]{
valid result;
valid flag;
}
}
}
            ..................................
}
```

For example, if you want to use function add (add is a function of resource ALU which is instantiated from model alu), use defined variables, result, flag, a, b and write as follows.

```
<result, flag> = ALU.add (a,b);
```

Variables a and b are inputs to ALU and result and flag are outputs from ALU. Same description like the function call in the programming language applies to the inputs to the resources. The order of input to resource should follow the order of the “port name” in input code of the “Function Set”.  When resource has multiple outputs, use brackets “<>” to place the variables between them. The order of outputs from resource should follow the order of the “port name” in output code of “Function Set”.

## 2. Elements for Micro-operation Description Statement

### 2.1 Variable

When you write micro-operation description, variables are when transferring the data. In order to use the variable, it must be declared before its use. Refer to Appendix for BNF to see the detailed information for variable declaration. The variables used for micro-operation description may be categorized into two groups.

(1) Variable declared for each instruction

This variable is used for multiple pipeline stages. It functions as pipeline register.

(2) Variable declared in pipeline stages

This variable is used only within the pipeline stage. It functions as signal line to transfer the data between the resources.

### 2.2 Binary constant

All the constants used for micro-operation description shall be binary constants.

The binary constants may be grouped in two groups. First group is “bit literal”, which is represented by ‘0’ (zero bracketed with single quote) or ‘1’(one bracketed with single quote). Second is “vector literal”.　It is represented by sequence of 0 and 1 bracketed with double quote (ex. “010”).　Generally, one bit constant is represented by bit literal, and constants longer than two bits are represented by vector literal.

## 2.3 Concatenation

When you concatenate bits, you are required to use brackets “<>” to show the concatenation. Variables are the only available element that you can concatenate. For example, if you want to concatenate three variables a(3 bit), b(2 bit), and c(4 bit) to handle as one variable d(9 bit), write the following statement.

```
d=<a,b,c>;
```

Here the most significant bit of variable d is that of the variable a, and the least significant bit of variable d is that of variable c.

## 2.4 Bitwise operator

There are three bitwise operators which are AND, OR and one’s complement operator.
All values for bitwise operators shall be variables.

(1) Bitwise AND operation

Use “&” operator.

```
a=”010”;
b=”110”;
c=a&b;
```

From this example, the variable c results to “010”.

(2) Bitwise OR operation

Use “|” operator.

```
a=”010”;
b=”110”;
c=a|b;
```

From this example, the variable c results to “110”.

(3) One’s complement operator

Use “~” for operator.

```
a=”010”;
```

```
b=~a;
```

From this example, the variable b results as “101”.

## 2.5 Relational Operator

Use relational operator to compare the variable and provided constants. There are two relational operators as follows.

(1) Equality

```
b=a==”01”;
```

From this example, b becomes ‘1’ when a is equal to “01”. If a is not equal to “01”, b becomes as ‘0’.

(2) InEquality

```
b=a !=”01”;
```

From this example, b becomes ‘0’ when a is equal to “01”. If a is not equal to “01”, b becomes as ‘1’.

## 2.6 Bit Range Specification

Use this operator to extract certain bit range from variable.

(1) To specify one bit is written as;

```
b = a[3];
```

From this example, the third bit of variable a is assigned to b. Therefore, b must be declared as one bit variable.

(2) To specify more than two bits write as;

```
b = a[3:1];
```

From this example, the first to third bit of variable a is assigned to b.  Therefore, b must be declared as three bits variable.

## 2.7 Conditional Operation

Conditional operations are written as follows.

(1) Conditional data transfer

```
result = (condition) ? a:b;
```

The same concept of ternary operator of C programming language applies to this statement. For this example when conditional variable condition is 1, assign a to result, when 0 assign b to result.

(2) Conditional resource execution

The following example shows write function of resource REG is executed, when conditional variable condition is set to 1.

```
null = [condition] REG.write(data);
```

### 3. Directions for writing micro-operations description.

Follow the below directions to write the micro-operation description.

The stage operation before the decode stage, which is specified by “Design Goal& Arch.Design”step of ASIP Meister, must be same in each instruction.

An Error occurs when you have a pair of instruction that causes resource confliction.

When the operand name is defined in the instruction type definition, the name is processed as a declared variable for the instruction deriving from the instruction type.

When you write the code, you must space one empty line between the variable declarations and micro-operation description.

## A BNF for Micro-operation Description

```
< Variable declaration part > ::= { < Variable declaration > }
< Variable declaration > ::= < Wire declaration >
< Wire declaration > ::= ’wire’ { <Bit range specification >} < Wire name > ’;’
< Wire name > ::= < Identifier >

< Micro-operation description part > ::= {< Micro-operation description > }
< Micro-operation description > ::= < Stage variable declaration part >
< Statements >
< Stage variable declaration part > ::= < Variable declaration part >
< Statements > ::= { < Statement > }

< Statement > ::= < Simple assignment statement > |
        < Conditional assignment statement > |
        < Conditional functional execution statement > |
        < Simple assignment statement > ::= < Left side > ’=’ < Right side > ’;’
        < Left side > ::= < Variable name > |< Variable name set> |’null’
        < Right side > ::= < Bitwise AND operation expression > |< Bitwise OR operation expression > |
                < One's complement operation expression > |< Comparison expression > |
                < Sets > |< Resource reference > |< Partial select expression > |
                < Binary constant > |< Variable reference >
        < Bitwise AND operation expression > ::= < Variable reference > ’&’  < Variable reference >
        < Bitwise OR operation expression > ::= < Variable reference > ’|’ < Variable reference >
        < One's complement operation expression > ::= ’~’ < Variable reference >
        < Comparison expression > ::= < Variable reference > < Relational operator > < Binary constant >
        < Relational operator > ::= ’==’ |’!=’

        < Variable reference > ::= < Variable name >
        < Variable name > ::= < Wire name >
        < Variable name set > ::= ’< ’ < Variable name > { ’,’ < Variable name > } ’ >’
        < Sets > ::= ’ <' < Variable reference > { ’,’ < Variable reference > } ’ >’

        < Resource reference > ::= < Resource name> ’.’ < Function name > ’(’  [<Argument list >] ’)’
        < Function name > ::= < Identifier >
        < Argument list > ::= < Argument > { ’,’ < Argument > }
        < Argument > ::= < Variable reference >

        < Partial select expression > ::= < Variable reference > < Partial selection operator >
        < Partial selection operator > ::= ’[’ < Index > [’:’ < Index >] ’]’
        < Index > ::= < Nonnegative integer >

        < Conditional assignment statement > ::= < Left side > ’=’ ’(’ < Conditional variable reference > ’)’ ’?’
                                                 < Variable reference > ’:’ < Variable reference > ’;’
        < Conditional variable reference > ::= < Variable reference >
        < Conditional functional execution statement > ::= < Left side > ’=’
                            ’[’ < Conditional variable reference > ’]’ < Resource function specification > ’;’
                            < Resource function specification > ::= < Resource name> ’.’ < Function name > ’(’
                            [<Argument list >] ’)’

        < Identifier > ::= < Alphabet > { < Alpha-numeral > |’ ’ }
        < Alphabet > ::= < Uppercase letter > |< Lowercase letter >
        < Uppercase letter > ::= ’A’ |’B’ |’C’ |’D’ |’E’ |
                            ’F’ |’G’ |’H’ |’I’ |’J’ |
                            ’K’ |’L’ |’M’ |’N’ |’O’ |
                            ’P’ |’Q’ |’R’ |’S’ |’T’ |
                            ’U’ |’V’ |’W’ |’X’ |’Y’ |’Z’
        < Lowercase letter > ::= ’a’ |’b’ |’c’ |’d’ |’e’ |
                            ’f’ |’g’ |’h’ |’i’ |’j’ |
                            ’k’ |’l’ |’m’ |’n’ |’o’ |
                            ’p’ |’q’ |’r’ |’s’ |’t’ |
                            ’u’ |’v’ |’w’ |’x’ |’y’ |’z’
        < Alpha-numeral > ::= < Alphabet > |< Number >
        < Number > ::= ’0’ |< Non-zero numbers >
        < Non-zero numbers > ::= ’1’ |’2’ |’3’ |’4’ |’5’ |’6’ |’7’ |’8’ |’9’

        < Natural number > ::= < Non-zero numbers > { < Number > }
        < Nonnegative integer > ::= ’0’ |< Natural number >

        < Binary constant > ::= < Bit literal > |< Vector literal >
        < Bit literal > ::= ’’’ < Binary character > ’’’
        < Vector literal > ::= ’”’ < Binary character > {< Binary character >} ’”’
```

< Binary character > ::= ’0’ | ’1’

< String > ::= ’”’ { < Character other than LF(Line Feed) character > } ’”’

< Comments > ::= < Same as comment rule in C/C++ language >