Fusion Language v1

Fusion file consists of set of macro definitions or include statements.

Include statement allows reuse of previously defined macros and comes in form:

include "relativePath\file.ext"

Macro definition is the main compilation element of the language and comes in form:

macro <MacroName> <ParameterList> { <MacroBody> }

So it is always starts with "macro" keyword followed by the name of the macro. Name of macro should follow rules of identifiers (like in C, C#, VB etc). Parameter list is optional and is comma separated list of identifiers. Macro body always starts with open curly brace and ended with closing one. It consists of set of numbers, macro expansions or "if" or "error" statements. There should be one macro called main. It will be the macro compiler starts expansion. Main macro should not have any parameters.

Let see for some examples.

macro main { 0 }

This will define macro main which is fully constitute a fusion program so it will be compiled without errors and produce file that contain one byte – 0.

To try this in your favorite text editor create a file that contains the above line, and save it with any extension you like (Test1.asm for example) and then from a command window run the following command (This assume you either have a copy of the Fusion.exe in current folder or added a path environment variable to a folder that contains it):

Fusion.exe Test1.asm Test1.bin

This will produce the following output on the screen:

C:\FusionTest>Fusion.exe Test1.asm Test1.bin

0

>>>>>00000000: 00

To see the content of the Test1.bin file open it in a binary viewer or editor of your choice. If you do not have one you can open it in LogicCircuit by creating a ROM circuit and set it data and address bit width to 8 bits both. Then in the ROM dialog click Load… button , select the bin file and click OK. You’ll see the content of the file in the editor grid.

The following fusion file will produce 4 byte file:

macro main { 3 4 5 6 }

Each number can come in one of the following form:

* Integer number (like 1, 3, 100, 1024)
* Hexadecimal number (like 0x1, 0xFF, 0x3c)
* Binary number (like 0b1, 0b0, 0b1010101)

Instead of number you can put a string in form:

"Some text you like to have here"

In the output it will be replaced with ASCII representation of all the characters between the quotas. At the end it will be a 0 byte to indicate end of the string.

Now let see some more complex example:

macro ThreeAndFive { 3 5 }

macro main { 4 ThreeAndFive 6 }

This two line file will be compiled to 4 byte file with content: 4 3 5 6. This is because the first macro defines two byte output and main is calling it among its own values.

So as you can see instead of any set of numbers it can be a macro call (or macro expansion). Actually it also can be an expression so for example:

macro main { 3 + 4 }

This will produce one byte file with content 7.

Macros can have parameters. For example:

macro sum a, b { a + b }

macro main { 1 sum 2, 3 4 }

This will produce 3 byte output: 1, 5, 4

Let see how it happened: main start expanding its body and the first number is 1, so it gets to the result. Then it comes a call to sum macro. This macro defined with two parameters: a and b, so main will expect a list of two comma separated expressions. In our example the expressions are trivial – just 2 and 3. So now compiler will expand sum macro which is actually produce one number output which is sum of its’ parameters. This gives us 5 in the output. Then main continue with its own body and that’s where 4 gets to output.

So far we were using only + expression. Actually Fusion is following C language style of expressions so you can use:

|  |  |
| --- | --- |
| Symbol | Expression |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| % | Reminder |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise NOT |
| << | Left shift |
| >> | Right shift |
| < | Less than |
| <= | Less than or equal |
| == | Equal |
| != | Not equal |
| >= | Greater than or equal |
| > | Greater than |
| && | Logical AND |
| || | Logical OR |

All comparisons and logical operators produce either 0 or 1 value.

You can group expression in parentheses ( and ).

The language cannot be full without conditional operators. In Fusion it is "if" statement which comes in form:

if(condition) { true clause } [else { false clause}]

The else part is optional. The condition is an expression which is evaluated to some number. If this number not equals to zero then the true clause get expanded and it is equals to zero then false clause get expanded if it is present.

Let’s look at the examples:

macro max a, b { if(a > b) { a } else { b } }

macro main { max 1, 3 }

This will produce one byte output with value 3.

The last operator of the language is error statement it comes in form:

error "Error message"

when executed it will produce a compilation error and can be used to validation of macro calls.

You can use comments in your file. To start comment put a semicolon at any pleace where you can have a white space. From the semicolon to the end of the line will be ignored:

; Here is my first comment.

macro main

{ ; this will be ignored

0

}

You can refer to position in the output file by defining a label in form:

LabelName:

Let look at this example:

macro Quote argument {

argument

}

macro main {

Quote 3

Quote 4

MyLabel:

Quote 5

Quote MyLabel

Quote 6

}

This will produce binary file with content: 3, 4, 5, 2, 6. Number 2 was the result of getting value of MyLabel which is second byte in the output.

# Formal definition of the Fusion language.

<FusionFile> ::= <FusionDefinitionList>

<FusionDefinitionList> ::= <FusionDefinition> | <FusionDefinition> <FusionDefinitionList>

<FusionDefinition> ::= <MacroDefinition> | <IncludeStatement>

<IncludeStatement> ::= include <String>

<String> ::= "<StringChars>"

<MacroDefinition> ::= <MacroKeyword> <MacroName> <FormalParameterList> { <MacroBody> }

<MacroKeyword> ::= macro

<MacroName> ::= <Identifier>

<FormalParameterList> ::= <EmptyParameterList> | <ParameterList>

<EmptyParameterList> ::=

<ParameterList> ::= <ParameterName> | <ParameterName>, <ParameterList>

<ParameterName> ::= <Identifier>

<MacroBody> ::= <EmptyMacroBody> | <StatementList>

<EmptyMacroBody> ::=

<StatementList> ::= <Statement> | <Statement> <StatementList>

<Statement> ::= <String> | <Label> | <Expression>

<Label> ::= <LabelName>:

<LabelName> ::= <Identifier>

<Expression> ::= <Number> | <LabelName> | <MacroCall> | <ArithmeticExpression> | <IfStatement> | <ErrorStatement> | (<Expression>)

<Number> ::= <DecimalNumber> | <HexadecimalNumber> | <BinaryNumber>

<MacroCall> ::= <MacroName> <ActualParameterList>

<ArithmeticExpression> ::= <Expression> <BinaryOperation> <Expression> | <UnaryOperation> <Expression>

<BinaryOperation> ::= +| - | \* | / | % | & | | | << | >> | > | >= | == | != |<= |< | && | ||

<UnaryOperation> ::= - | + | ~ | !

<ActualParameterList> ::= <EmptyParameterList> | <ExpressionList>

<ExpressionList> ::= <Expression> | <Expression>, <ExpressionList>

<IfStatement> ::= <SimpleIfStatement> | <IfElseStatement>

<SimpleIfStatement> ::= if(<Expression>) { <MacroBody> }

<IfElseStatement> ::= <SimpleIfStatement> else { <MacroBody> }

<ErrorStatement> ::= error <String>

<DecimalNumber> ::= <DecimalDigit> | <DecimalDigit><DecimalNumber>

<HexadecimalNumber> :: 0x<HexadecimalDigits>

<BinaryNumber> ::= 0b<BinaryDigits>

<HexadecimalDigits> ::= <HexDigit> | <HexDigit><HexadecimalDigits>

<BinaryDigits> ::= <BinDigit> | <BinDigit><BinaryDigits>

<DecimalDigit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<HexDigit> ::= <DecimalDigit> | A | B | C | D | E | F | a | b | c | d | e | f

<BinDigit> ::= 0 | 1

<Identifier> ::= <Letter> | <Letter><LetterOrDigitChain>

<LetterOrDigitChain> ::= <LetterOrDigit> | <LetterOrDigit><LetterOrDigitChain>

<LetterOrDigit> ::= <Letter> | <DecimalDigit>

<Letter> ::= Like in C: [a-zA-Z\_]

<StringChars> ::= Like in C with escapes

# Some practical examples:

## Defining and using addition assembly instruction for some hypothetical CPU

;Define registers

macro A { 0 }

macro B { 1 }

macro C { 2 }

macro D { 3 }

;Checks if register is valid value

macro ValidateRegister register {

if(!(0 <= register && register <= 3)) {

error "Invalid register " + register

}

}

;Defines Addition code of operation. This command adds register A with the provided register as a parameter.

;Code of operation is 8 bit number with high 6 bits code of addition and 2 low bits register number to add to A.

macro ADD register {

; Make sure the register is a valid number

ValidateRegister register

;Code of ADD operation is 0b011001 concatenate it with register number to produce actual code of operation

0b01100100 | register

}

;Defines halt CPU command

macro HALT {

;code of operation for halt command is 0

0

}

macro main {

;sample program that adds value in register B to value in register A and store the result in register A, then it halts the execution

ADD B

HALT

}

## Defining and using labels

macro Quote argument {

argument

}

macro main {

Quote 3

Quote 4

MyLabel:

Quote 5

Quote MyLabel

Quote 6

}

## Filling up memory block with 0

macro Fill value, size {

if(size > 0) {

value

Fill value, size - 1

}

}

macro main {

Fill 0, 10

}