Miny18

Programmers Manual

By Mike Christle

Revision History

|  |  |  |
| --- | --- | --- |
| Version | Date | Description |
| 1.0 | 09/01/2020 | Initial release |

Table of Contents

[Introduction 1](#_Toc49850163)

[TERMS OF USE: MIT License 1](#_Toc49850164)

[Build Process 2](#_Toc49850165)

[Program Structure 2](#_Toc49850166)

[Language Syntax 3](#_Toc49850167)

[Comments 3](#_Toc49850168)

[Data Types 3](#_Toc49850169)

[Literals 3](#_Toc49850170)

[Variable Declaration 4](#_Toc49850171)

[Integer Arrays and Variables 4](#_Toc49850172)

[boolean Variables 5](#_Toc49850173)

[If Statements 5](#_Toc49850174)

[Loops 5](#_Toc49850175)

[Functions 6](#_Toc49850176)

[Interrupt Service Routines 7](#_Toc49850177)

[Pause Timer 8](#_Toc49850178)

[System Control 8](#_Toc49850179)

[Strings 9](#_Toc49850180)

[Appendix A - BNF Grammar 10](#_Toc49850181)

# Introduction

Does the world need another compiler? Probably not. However, I took a course in compiler design a couple of years ago and found the subject absolutely fascinating. Since then I have had a lot of fun working on this little project. If anyone else finds it useful, well that’s good to.

Miny18 is a command line compiler that outputs Proc18 assembly code. Proc18 is a companion project that I have been working on. Proc18 is a micro-controller implemented in Verilog for use in FPGA designs. See the Proc18 User’s Manuel for details.

The language is called Miny18 because my original design goal was to develop a useful language with minimal syntax. After many iterations of both Miny18 and Proc18 both have become very useful tools for FPGA based embedded applications. Miny18 consist of two programs, a compiler and an assembler, both written in Java.

# TERMS OF USE: MIT License

Copyright Mike Christle 2020

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# Build Process



The .m file is your Miny18 source. The .asm file contains the Proc18 assembly source. The .asm file is passed to the Proc18 assembler. The code.hex file contains the object code, and the const.hex contains constant data, as hexadecimal constants. The .hex files are included in the applicable Verilog files for code and constant data.

I developed this project on a Windows PC, so I have included several .bat files with the source. You Linux people should have no trouble converting these to script files.

init.bat Initialize environment variables

b.bat Build a single Java file

ball.bat Build all Java files

miny.bat Compile and assemble a miny18 file

Miny18 will stop on the first error and output a (hopefully useful) error message and the source line number.

# Program Structure

Each .m file contains a module. The file name must match the module name. Lines starting with the ‘#’ character are comments. This listing shows the code to light a LED.

|  |
| --- |
| # File: LedOn1.m  module LedOn1  {  bool LED 63  func main()  {  LED = true  halt  }  } |

The top level module in a project must have a main function that has no parameters and returns no value. This program ends with a halt statement because the main function must never return. A typical program will either setup interrupts then halt, or use an endless loop. The bool statement defines the boolean port that is connected to the LED in Verilog, see the Proc18 User’s Manual for details. Ports can be boolean or integer. In a real project I put all of the port declarations in a separate file named IO.m, and then reference these ports in the other files by appending IO. to the port name. So this program should look like the following:

|  |
| --- |
| # File: IO.m  Module IO  {  bool LED 63  port UART 23  }  # File: LedOn2.m  module LedOn2  {  func main()  {  IO.LED = true  IO.UART = ‘A’  halt  }  } |

Variables declared in a function are only visible to that function. All functions are global, as are all port, constants and variables declared at the module level.

# Language Syntax

Miny18 is case sensitive, ABC != Abc != abc. Appendix A is the BNF for the precise syntax of the language.

## Comments

Miny18 supports line comments with the # character. Anything on the line after the # is ignored.

## Data Types

int 18 bit signed integer

bool 1 bit boolean

## Literals

Literal values are specified similar to C, they start with a digit and the default is decimal. The 0x, 0o, and 0b prefixes specify hexadecimal, octal, and binary formats respectively. Character constants are enclosed in single quotes, for example ‘A’, and are interpreted as integers. Literal values can contain the underscore character to make them more readable.

Named literals can be declared to the module level using the const keyword. The expression for the value can use any of the regular operators, plus division and modulo.

## Variable Declaration

Variables can be declared at the module level or anywhere in a function. Variables declared to the module level are visible to other modules with the syntax ModuleName ‘.’ VariableName. Variables declared in a function must be given a value when declared, and must be declared before being used.

|  |
| --- |
| #File: Mod1.m  module Mod1  {  bool b1, b2  int i1, i2  const int WIDTH = 0x100  const int HEIGHT = 0x200  const int SIZE = WIDTH \* HEIGHT  const bool ENABLED = true  }  # File: Mod2.m  module Mod2  {  func some\_func()  {  int i3 = Mod1.i1  bool b3 = Mod1.b2  int i4 = Mod1.SIZE  }  } |

## Integer Arrays and Variables

Integer variables can be declared in functions or at the module level. Integer arrays can be declared only at the module level. Variables are stored in one of 64 registers. Arrays are stored in external RAM, see the Proc18 User’s Manual for size limitations.

Integer expressions can be constructed using these operators. They are listed in order of precedence.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Assignment** | **Description** |
| () |  | Parentheses. |
| + - ~ |  | Unary positive, negative, bitwise not. |
| \* | \*= | Multiply. |
| + - | += -= | Addition, subtraction. |
| << >> | <<= >>= | Shift left, shift right. |
| & | &= | Bitwise and. |
| ^ | ^= | Bitwise exclusive or. |
| | | |= | Bitwise or. |

Note: The Proc18 core does not have division or modulus operators. These can be added with a utility module, see the Proc18 User’s Manual.

## boolean Variables

Proc18 supports 64 boolean variables which are shared with boolean IO ports, see the Proc18 User’s Manual for details. An integer constant follows the variable name specifies the boolean bit number. If no number is given, the compiler will assign one starting at 0. Hardware connections should start at the high end.

|  |
| --- |
| **bool LED 63 # Hardware connection**  **bool b1, b2 # Variables** |

Logical expressions can be constructed using these operators. They are listed in order of precedence.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| not | Logical not. |
| == != < > <= >= | Compare two expressions. |
| and | Logical and. |
| or | Logical or. |

## If Statements

If statements are constructed similar to Python, but with less syntax.

|  |
| --- |
| **if i == 2 j = 5**  **elif i >= 5 and b b = false**  **else**  **{**  **b = true**  **j = -2**  **}** |

## Loops

Miny18 has a single loop statement loop, which loops forever. Use the break statement to exit a loop, and continue statements to restart the loop.

|  |
| --- |
| **int i = 0**  **bool b = false**  **loop**  **{**  **if i > 10 break**  **...**  **if b continue**  **i += 1**  **}** |

## Functions

Functions are declared with the func keyword, followed by the function name, followed by the parameter list in parentheses, and finally the optional return type or an ISR number. Interrupt Service Routines are explained in the next section. Functions can access parameters, local variables, and module variables. To call a function in another module, append the module name and a period.

|  |
| --- |
| **module Mod1**  **{**  **int i1**  **func f1(int i) int**  **{**  **int j = 5**  **return i + j**  **}**  **func f2(bool b) bool**  **{**  **return not b**  **}**  **func f3()**  **{**  **i1 += f1(3)**  **}**  **}**  **module Mod2**  **{**  **func main()**  **{**  **Mod1.f3()**  **int i = Mod1.f1(5)**  **}**  **}** |

## Interrupt Service Routines

The Proc18 processor provides 15 levels of interrupts. To link a function to an interrupt level, add the isr keyword and an integer from 1 to 15 on the function declaration. Obviously, an ISR can’t accept or return parameters.

The level keyword sets the interrupt level. Interrupts above the set level are allowed. The parameter must evaluate to an integer constant in the range 0 to 15, where 0 allows all interrupts, and 15 blocks all interrupts.

The halt keyword stops the processor until an interrupt occurs. After the ISR returns, processing continues after the halt statement. Since this application has no more work to do, I put the halt inside a loop.

The INT\_SET and INT\_TIMER ports access Verilog functions that are detailed in the Proc18 User’s Manual.

|  |
| --- |
| **# Toggle an LED once a second**  **module ToggleLED**  **{**  **bool LED 63 # Port number of the LED**  **port INT\_SET 0o01 # Enable interrupt source**  **port INT\_TIMER 0o10 # Periodic interrupt timer**  **func main()**  **{**  **INT\_TIMER = 100000 # 10 uSec \* 100000 = 1 Sec**  **INT\_SET = 0x10 # Enable interrupt 5**  **level 2 # Allow all interrupts above level 2**  **loop halt # Halt all processing**  **}**  **func timer\_isr() isr 5**  **{**  **int a = INT\_TIMER # Clear timer interrupt flag**  **LED = not LED # Toggle LED**  **}**  **}** |

## Pause Timer

The timer and pause keywords work together to provide short and precise time delays. The timer keyword is followed by an integer constant in the range 1 to 4095. This loads a counter that will count down to zero once per processor clock cycle, and then stop. The pause keyword will pause processing until the timer decrements to zero. You can do any other work between timer and pause. If the other work takes longer than the timer period the pause will continue immediately, and the timing will be extended accordingly. There is a 3 clock overhead so the counts are 3 less than the desired delay.

|  |
| --- |
| **# Output a 1 uSec pulse every 10 uSec**  **# Clock frequency 50 MHz**  **# 1 uSec = 50 clocks**  **# 10 uSec = 500 clocks**  **module Pulse**  **{**  **bool PIN 62**  **func main()**  **{**  **loop**  **{**  **pause**  **timer 47 # 50 - 3**  **PIN = true**  **# Do other work**  **pause**  **timer 447 # 500 – 50 - 3**  **PIN = false**  **# Do other work**  **}**  **}**  **}** |

## System Control

The reset command will pulse the RESET signal to reset the registers in the Bit Memory, the timer, and the utility modules. The restart command will restart the processor by setting the program counter and the stack pointer to zero.

## Strings

The string keyword is used to declare an ASCII string. Since an ASCII character uses only 7 bits and the constants ROM a 18 bits, ASCI characters are packed two per ROM word. The first character in the string is in the low half of the word and the next character is in the high half. A terminating zero is also added to the end.

|  |
| --- |
| **module StrTest**  **{**  **port BAUD\_RATE 0o30**  **port UART\_DATA 0o40**  **bool TX\_EMPTY 0o74**  **string str1 "Hello World\n"**  **func main()**  **{**  **BAUD\_RATE = 326**  **int i = 0**  **loop**  **{**  **int c = str1[i]**  **if c == 0 break**  **putc(c)**  **c >>= 9**  **if c == 0 break**  **putc(c)**  **i += 1**  **}**  **halt**  **}**  **func putc(int c)**  **{**  **loop if TX\_EMPTY break**  **UART\_DATA = c**  **}**  **}** |

# Appendix A - BNF Grammar

//-------------------------------------------------------------------------

// Miny18 Language Grammar

// Mike Christle Aug 2019

//-------------------------------------------------------------------------

// Abc Rule

// -> Is defined by

// abc Keyword

// | Or

// [] Range

// + Repeat one or more times

// \* Repeat zero or more times

// ? Repeat zero or one times

// # Comma separated list X# -> X ( ',' X )\*

// () Group

// 'x' Symbol

// ; Rule terminator

//-------------------------------------------------------------------------

Start -> ModuleDecl\* ;

ModuleDecl -> module Label '{' ( ItemDecl | FuncDecl )\* '}' ;

ItemDecl -> IntDecl | BoolDecl | PortDecl | ConstDecl ;

FuncDecl -> func Label ParmList ( bool | int | IsrDecl )? CodeBlock ;

ParmList -> '(' ( ParmType Label )#? ')' ;

ParmType -> bool | int | ram | rom ;

IsrDecl -> isr DecConst ;

IntDecl -> int ( Label ( '[' IntConst ']' )? )# ;

BoolDecl -> bool ( Label ( IntConst )? )# ;

PortDecl -> port Label IntConst ;

ConstDecl -> const Label ( LogOrExpr# | StringConst ) ;

CodeBlock -> '{' Statement\* '}' ;

Statement -> DataDecl | AssignStmt | IfStmt

| LoopStmt | BreakStmt | ContinueStmt

| ResetStmt | RestartStmt | LevelStmt

| HaltStmt | TimerStmt | PauseStmt

| ReturnStmt | CodeBlock ;

DataDecl -> ( bool | int ) Label '=' LogOrExpr ;

AssignStmt -> LabelDest AssignOp LogOrExpr ;

LabelDest -> Label

| Label '[' OrExpr ']'

| Label '.' Label

| Label '.' Label '[' OrExpr ']' ;

AssignOp -> '='

| '+=' | '-=' | '\*='

| '&=' | '|=' | '^='

| '>>=' | '<<=' ;

IfStmt -> IfClause ElIfClause\* ElseClause? ;

IfClause -> if LogOrExpr Statement ;

ElIfClause -> elif LogOrExpr Statement ;

ElseClause -> else Statement ;

LoopStmt -> loop Statement ;

BreakStmt -> break ;

ContinueStmt -> continue ;

ResetStmt -> reset ;

RestartStmt -> restart ;

LevelStmt -> level IntConst ;

HaltStmt -> halt ;

TimerStmt -> timer OrExpr ;

PauseStmt -> pause ;

ReturnStmt -> return ( LogAndExpr )? ;

LogOrExpr -> LogAndExpr ( or LogAndExpr )\* ;

LogAndExpr -> RelExpr ( and RelExpr )\* ;

RelExpr -> OrExpr ( CompareOp OrExpr )? ;

CompareOp -> '==' | '!=' | '>=' | '<=' | '>' | '<' ;

OrExpr -> XorExpr ( '|' XorExpr )\* ;

XorExpr -> AndExpr ( '^' AndExpr )\* ;

AndExpr -> ShiftExpr ( '&' ShiftExpr )\* ;

ShiftExpr -> AddExpr ( ( '<<' | '>>' ) AddExpr )? ;

AddExpr -> MultExpr ( ( '+' | '-' ) MultExpr )\* ;

MultExpr -> UnaryExpr ( ( '\*' | '/' | '%' ) UnaryExpr )\* ;

UnaryExpr -> ( '+' | '-' | '~' | not )? Atom ;

Atom -> LabelExpr

| BoolConst

| IntConst

| '(' LogOrExpr ')' ;

LabelExpr -> Label

| Label '[' OrExpr ']'

| Label '(' LogOrExpr# ')'

| Label '.' Label

| Label '.' Label '[' OrExpr ']'

| Label '.' Label '(' LogOrExpr# ')' ;

Label -> [a-zA-Z][a-zA-Z0-9\_]\* ;

BoolConst -> true | false ;

IntConst -> BinConst | OctConst | DecConst | HexConst | CharConst ;

DecConst -> [0-9][0-9\_]\* ;

BinConst -> 0b [01\_]+ ;

OctConst -> 0o [0-7\_]+ ;

HexConst -> 0x [0-9a-fA-F\_]+ ;

StringConst -> '"' ascii\_text '"' ;

CharConst -> ''' ascii\_char ''' ;