Asm

COLLABORATORS			
	TITLE : Asm		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 31, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Asm

1.1 main

```
   Short instruction for PtnAsm v1.0.

   Introduction
   Command line parameters
   Labels and expressions
   Directives
   Extended expressions
   Predefined symbols
   Thanks
   Registration
```

1.2 Introduction

```
PtnAsm is a PowerPC assembler for Amiga. It requies a ppc processor
and a WarpOS installed. Most important features:

- creates objects in EHF format

- 603 and 604 instructions and all 32-bit extended mnemonics are
  supported

- complete floating-point support

- uses the same syntax as PowerASM

- support for most of the popular directives

Note: PtnAsm still uses a 68k names of data types:
   byte       -  8 bit integer
   word       -  16 bit integer (halfword in ppc architecture)
   longword   -  32 bit integer (word in ppc architecture)
```

1.3 Command line parameters

```
You can call the assembler with following parameters:

-o filename
              Specifies the name of the output file. If this parameter is
              left no output will be generated.

-binary
              Forces assembler to write a binary file. This file consist pure
              code.
-link
              In this mode assembler will produce a linkable file. Then
              it can be processed by linker. This option is a default.

-errors=<max errors>
              Specifies maximan number of errors.

-D<symbol>[=value]
              Defines a new symbol. If [value] is missed it defaults to 1.
              Several symbols may be defined by separating them with commas.

-I<path>
              Defines a directory, where assembler should look for includes
              and incbins. Several paths may be defined by separating them
              with commas.
```

1.4 Labels and expressions

```
Labels must always start in the first column of a line. The colon after a
label is optional. Valid characters for labels are: 'a'-'z', 'A'-'Z', '0'-'9',
'_', '@'. Labels can't start start with a digit. Local labels are preceded
by a dot '.', and they're valid between two global labels.

Expressions consist of symbols and and constans. Supported types of constans
are:

-binary       preceded by a '%', consist of '0' and '1'
-decimal      consist of '0'-'9'
-hexadecimal  preceded by a '$' consist of '0'-'9' and 'a'-'f'
-float        has the form [+/-][integer][.fraction]
-string       embedded by ' or "
              "" or '' sequenct will be replaced by " or '

Assembler supports following integer operators (from highest to lowest
priority):

1. -(x)    negation               ~(x)       not
   high(x) extracts upper 16 bits  low(x)     extracts lower 16 bits

2. << shift left       >> shift right
3. &  and              |  or             !   or        ^  exclusive or
4. *  multiplication   /  division       //  modulo
5. +  addition         -  subtraction
```

```
6. Comprasions:
   =   or  ==  equal
   !=  or  <>  not equal
   <           greater then
   >           lower then
   <=  or  =<  lower/equal then
   >=  or  =>  greater/equal then

   Comprasion operators return -1 if comprasion is true or 0 if comprasion is
   false.

Example:
label1     =     ~(low($1fffe)-2+3<<1)

Following operations are allowed for relative or external symbols:

   reloc + abs
   reloc - abs
   reloc - reloc

   extern + abs
   extern - abs

You can use also floating-point expressions. The following
operators are valid in this mode:

1. -(x)    negation
   sin(x)  sine
   cos(x)  cosine
   sqr(x)  square root
   exp(x)  e^x
   log(x)  natural logarithm
   tan(x)  tangent

   Arguments of trigonometric functions are expressed in radians, unles
   you change it by a deg directive.

2. * multiplication          / division
3. + addition                - subrtaction


Comments start with ';' or '*'. If operand is given the comment must start
with ';' character.
```

1.5 Directives

```
1. Directives to assign constans

   =
   ==
   equ
   equate
   define
   set
```

```
   fequ
   fset

2. Directives to assign registers to symbols

   equr
   setr
   fequr
   fsetr

3. Directives to conditional assembly

   ifxx
   else
   elseif
   endc
   endif

4. Directives to define macros

   macro
   endm
   mexit
   rexit

5. Directives to define repetitions

   rept
   repeat
   endr
   endrepeat

6. Directives to control output

   print
   echo
   printx
   printexp
   printv
   printvf
   printfloat
   fail

7. Directives to define structures

   so
   clrso
   soreset
   setso
   soval
   rs
   clrrs
   rsreset
   setrs
   rsval
   fo
   clrfo
```

```
   foreset
   setfo
   foval

8. Directives to align structures

   alignso.x
   alignrs.x
   alignfo.x

9. Directives to define sections

   section
   code
   code_f
   code_c
   data
   data_f
   data_c
   bss
   bss_f
   bss_c
   save
   restore

10. Directives to align code

   align
   alignndx
   cnop
   align.x

11. Directives to import or export symbols

   xdef
   global
   xref
   extern

12. Directives to include files

   incdir
   include
   incbin

13. Directives to define unit name

   ttl
   unit

14. Directives to define constants

   dc
   cstring
   cstr

15. Directives to define memory blocks
```

```
   ds
   blk
   dcb

17. Directives to toggle escape characters

   escapestr
   permit_esc
   forbid_esc

16. Miscellaneous

   deg
   rad
```

1.6 EQU

```
   <symbol> =       <expression>
   <symbol> ==      <expression>
   <symbol> equ     <expression>
   <symbol> equate  <expression>
   <symbol> define  <expression>
```

Assigns <expression> to a <symbol>.

1.7 SET

```
   <symbol> set     <expression>
```

Assigns <expression> to a <symbol>. A value of symbol defined by a SET can be changed by another usage of SET directive. No relocatables or exterals are allowed.

1.8 FEQU

```
   <symbol> fequ    <expression>
```

Resoult of a floating point expression is assigned to a <symbol>.

1.9 FSET

```
<symbol> fset    <expression>
```

Resoult of a floating point expression is assigned to a <symbol>. A value of symbol can be changed by another usage of FSET directive.

1.10 IFxx

```
ifxx
```

These directives mark beginning of conditional assembly. The code following ifxx directive is assembled if a condtion is true.
Supported conditions are:

```
ifc     <string1>,<string2>
   Compares two strings. Code is assembled if strings are identical.

ifnc    <string1>,<string2>
   Code is assembled if strings aren't identical.

ifd     <symbol>
ifdef   <symbol>
   Tests if <symbol> was defined before if directive.

ifnd    <symbol>
ifndef  <symbol>
   Tests if <symbol> wasn't defined.

if      <exp>
ifne    <exp>
   Tests if <exp> != 0

ifeq    <exp>
   Tests if <exp> = 0

ifgt    <exp>
   Tests if <exp> > 0

ifge    <exp>
   Tests if <exp> >= 0

iflt    <exp>
   Tests if <exp> < 0

ifle    <exp>
   Tests if <exp> <= 0


It's also possible to use some of ifxx directives with two arguments:

ifne    <exp1>,<exp2>
  Tests if <exp2> != <exp1>
```

```
ifeq    <exp1>,<exp2>
  Tests if <exp2> = <exp1>

ifgt    <exp1>,<exp2>
  Tests if <exp2> > <exp1>

ifge    <exp1>,<exp2>
  Tests if <exp2> >= <exp1>

iflt    <exp1>,<exp2>
  Tests if <exp2> < <exp1>

ifle    <exp1>,<exp2>
  Tests if <exp2> <= <exp1>
```

1.11 ELSE

```
else
elseif
```

Negates the result of last if directive.

1.12 ENDC

```
endc
endif
```

Ends an if block.

1.13 MACRO

```
<symbol>  macro
```

Begins definition of a macro. The code between macro and endm directives
will be inserted into source when assembler finds <symbol>. You
can pass 34 arguments to a macro separated by a comas. First 9 arguments
are referenced as \1 through \9 and last 25 as \a through \z.
Argument \0 is reserved for branch prediction or rc update:

```
macro: macro
       ble\0  \1
       endm
```

Complex text sequences can be passed between '<' and '>' characters.
Number of arguments passed to macro is stored in NARG symbol.

Macro functions:

\@
   Useful for createing unique labels. @ will be replaced by '_xxxx'
   sequence. xxxx is a number which is incremented by 1 each time
   the macro is called.

\
   Creates a single backslash character.

\(exp)
   First the exp expression is evaluated. Sequence \(exp) will be replaced
   by a macro's argument whose number is resoult of expression.

\.
   This sequence will be replaced by a macro's argument whose number
   is taken from the symbol CARG. CARG is initialized to 1 when macro
   is called.

\+
   This function is similar to \. but CARG will be incremented by 1
   after function usage.

\-
   This function is similar to \. but CARG will be decremented by 1
   after function usage.

1.14 ENDM

   endm

Terminates a macro definition.

1.15 MEXIT

   mexit

Assembler leaves a macro when it encounters this directive.

1.16 REXIT

   rexit   <exp>

Assembler leaves a macro if nesting depth equal to <exp> has been reached.

1.17 REPT

```
rept    <count>
repeat  <count>
```

All instructions embedded by REPT/ENDR, will be assembled <count> times.

1.18 ENDR

```
endr
```

Ends a loop.

1.19 PRINT

```
print   <text>
printx  <text>
echo    <text>
```

Prints a <text> to the standart output.

1.20 PRINTV

```
printv     <exp>
printexp   <exp>
```

Prints a value of expression <exp> to the standart output.

1.21 PRINTVF

```
printvf    <exp>
printfloat <exp>
```

Prints a value of floating point expression <exp> to the standart output.

1.22 FAIL

```
fail
```

This directive terminates assembly with an error.

1.23 SO

```
<symbol> so.x <count>
<symbol> rs.x <count>

RS (or SO) is internal counter. It can be used to create structures with
increasing offsets. First value of RS/SO counter is assigned to a <symbol>,
then it's increased by <cout> multiplied by size extension. The actual
value of RS/SO counter is stored in __RS and __SO symbols.
The size can be one of:
   .b   1 byte
   .w   2 bytes
   .l   4 bytes
   .q   8 bytes
   .s   4 bytes
   .d   8 bytes
```

1.24 CLRSO

```
   clrso
   soreset
   clrrs
   rsreset


Sets value of SO/RS counter to zero.
```

1.25 SETSO

```
   setso   <exp>
   setrs   <exp>

The SO/RS counter is set to the value of <exp>.
```

1.26 SOVAL

```
<symbol> soval
<symbol> rsval


The value of SO/RS counter is assigned to a <symbol>.
```

1.27 FO

```
<symbol> fo.x <count>
```

FO is internal frame offset counter. This directive decrements fo counter by size extension multiplied by <count>, then assigns this value to a <symbol>. Value of FO counter is stored in __FO symbol.

1.28 CLRFO

```
   clrfo
```

This directive sets value of FO counter to zero.

1.29 SETFO

```
   setfo   <exp>
```

FO counter is set to the value of <exp>.

1.30 FOVAL

```
<symbol> foval
```

The value of FO counter is assigned to a <symbol>.

1.31 ALIGNSO

```
     alignso.x
     alignrs.x
```

Aligns up the SO/RS counter to value divisible by:

```
   .b   1 byte
   .w   2 bytes
   .l   4 bytes
   .q   8 bytes
   .s   4 bytes
   .d   8 bytes
```

1.32 ALIGNFO

```
alignfo.x
```

This directive aligns down the FO counter to value divisible by:

```
.b   1 byte
.w   2 bytes
.l   4 bytes
.q   8 bytes
.s   4 bytes
.d   8 bytes
```

1.33 SECTION

```
section <name[,type[,memory type]]>
```

Creares a new section. <name> defines a name of a section. [type] can be one of: CODE,CODE_C,CODE_F,DATA,DATA_C,DATA_F,BSS,BSS_C,BSS_F. Legal memory types are FAST,CHIP,PUBLIC. If type is one of _C or _F types memory type must not be defined.

1.34 CODE

```
code
```

This directive is an alias for section "CODE",code.

1.35 CODE_F

```
code_f
```

This directive is an alias for section "CODE_F",code,fast.

1.36 CODE_C

```
code_c
```

This directive is an alias for section "CODE_C",code,chip.

1.37 DATA

```
data
```

This directive is an alias for section "DATA",data.

1.38 DATA_F

```
data_f
```

This directive is an alias for section "DATA_F",data,fast.

1.39 DATA_C

```
data_c
```

This directive is an alias for section "DATA_C",data,chip.

1.40 BSS

```
bss
```

This directive is an alias for section "BSS",bss.

1.41 BSS_F

```
bss_f
```

This directive is an alias for section "BSS_F",bss,fast.

1.42 BSS_C

```
bss_c
```

This directive is an alias for section "BSS_C",bss,chip.

1.43 SAVE

```
save
```

Saves the status of current section. It can be later be restored by restore directive.

1.44 RESTORE

```
restore
```

The status of saved section is restored.

1.45 ALIGN

```
align   <offset>,<align>
cnop    <offset>,<align>
```

The following code is aligned to the address divisible by <align>. Then the <offset> is added.

1.46 ALIGN.X

```
align.x
```

The following code is aligned according to the size extension.

1.47 XDEF

```
xdef    <symbol[,symbol[,symbol...]]>
global  <symbol[,symbol[,symbol...]]>
```

Makes a symbol visible to other modules.

1.48 XREF

```
xref    <symbol[symbol[,symbol...]]>
```

This directive tells assembler that <symbol> was defined outside this module.

1.49 EQUR

```
<symbol> equr <register>
```

This directive assigns a <register> to the <symbol>. Floating-point registers must be assigned by fequr direcitve.

1.50 SETR

```
<symbol> setr <floating-point register>
```

This directive assigns a <register> to the <symbol>. Registers defined by setr directive can be redefined in the source code. Floating-point registers must be assigned by fsetr direcitve.

1.51 FEQUR

```
<symbol> fequr <floating-point register>
```

This directive assigns a <floating-point register> to the <symbol>.

1.52 FSETR

```
<symbol> fsetr <floating-point register>
```

This directive assigns a <floating-point register> the <symbol>. Registers defined by fsetr directive can be redefined in the source code.

1.53 TTL

```
ttl  <name>
unit <name>
```

Sets the name of the object wich assembler will generate. It's stored in HUNK_UNIT, and it's used by a linker.

1.54 INCDIR

```
   incdir  <path[,path[,path..]]>
```

With this directive you can set a directories where assembler should
look for includes and incbins.

1.55 INCLUDE

```
   include <file>
```

Includes a file into a curent source. All paths defined by incdir are searched.

1.56 INCBIN

```
   incbin <file[,size[,offset]]
```

The file <file> is included into current section as a binary file. [size]
specifies the amount of bytes to be loaded. [offset] specifies an offset
form the beginning (or the end if offset is negative) of the file.

1.57 DC.X

```
   dc.x <exp1[,exp2[,exp3...]]>
```

The dc directive allocates and initializes one or more data elements. Each
of them has got a size specified by size extension. Valid extensions are:

```
   .b   1 byte
   .w   2 bytes
   .l   4 bytes
   .f   4 bytes single precision
   .d   8 bytes double precision
```

1.58 CSTRING

```
   cstring <string>
   cstr    <string>
```

Creates the null terminated string. This directive is an alias for:

```
   dc.b string,0
```

1.59 ESCAPESTR

```
   escapestr <mode>
```

Escapestr directive is use to forbid or permit usage of escape characters.
If mode is equal to zero usage of escape characters is forbiden.

Escape characters:

```
   \   -  '\' character
   \'   -  ',' character
   \"   -  '"' character
   \t   -  tabulator
   \0   -  NULL byte
   \f   -  from feed
   \n   -  line feed
   \e   -  escape code
   \b   -  backspace
   \r   -  carriage return
```

1.60 PERMIT_ESC

```
   permit_esc
```

This directive is an alias for "escapestr 1".

1.61 FORBID_ESC

```
   forbit_esc
```

This directive is an alias for "escapestr 0".

1.62 DS

```
   ds.x    <amount>[,fill]
   blk.x   <amount>[,fill]
   dcb.x   <amount>[,fill]
```

These direcitves allocate memory block having <amount> entries. The block
will be initialized with [fill]. If [fill] is missed the block will be
clreared.

1.63 DEG

```
   deg
```

Form now on, arguments of trigonometric functions are expressed in degrees.
See Labels and expressions

1.64 RAD

```
   rad
```

Form now on, arguments of trigonometric functions are expressed in radians.
See Labels and expressions

1.65 equations

With PtnAsm it's posible to write a code as a sequence of equations. What does it mean? Instead of writting addi r3,r4,3 you can write r3 = r4 + 3. This gives the same resoult. It's possible to acces almost all instructions in this way. The general format is:

```
   destination [.][o][s] = source1[operator source2 ...]
```

A dot '.' before equator tells assembler to set the rc bit, and 'o' to set the oe bit. 's' is used in floating-point expressions. If 's' is not ommitet assembler will try to use single precision instruction if possible.

Examples:

```
     r3 .o = r3 + r4         ->    addo.  r3,r3,r4
     f3 .s = f1 * f3         ->    fmuls. f3,f1,f3
     r5    = ashiftr(r4,4)   ->    srawi  r5,r4,4
```

The next sections describe how to use this feature of PtnAsm. Each section is divided into several parts. Each of them consist of operation's name and all posible combinations of rc,oe,s fields.

```
   Integer operations
   Logical operations
   Floatning point operations
   Memory acceses
   Comprasions
   Building structures
```

This feature also forces some restrictions. You can use a dot '.' only as first character of label. Otherwise it would cause conflicts with rn.size etc. If you mark comments with an asterix '*' please start them at the beginning of a line.

1.66 Integer Operations

```
1. Moving data from register to register

   rd   = ra           ->    mr       rd,ra
   rd  .= ra           ->    mr.      rd,ra
   rd   = ra.b         ->    mb       rd,ra
   rd  .= ra.b         ->    mb.      rd,ra
   rd   = ra.w         ->    mh       rd,ra
   rd  .= ra.w         ->    mh.      rd,ra
   rd   = ra.l         ->    mr       rd,ra
   rd  .= ra.l         ->    mr.      rd,ra

2. Loading immediate values

   rd   = simm         ->    li       rd,simm
   rn   = &_label      ->    la       rd,_label

3. Negation

   rd   = - ra         ->    neg      rd,ra

4. Addition

   rd   = ra + rb      ->    add      rd,ra,rb
   rd  .= ra + rb      ->    add.     rd,ra,rb
   rd  o= ra + ra      ->    addo     rd,ra,rb
   rd .o= ra + rb      ->    addo.    rd,ra,rb
   rd   = ra + simm    ->    addi     rd,ra,simm

4. Subtraction

   rd   = ra - rb      ->    sub      rd,ra,rb
   rd  .= ra - rb      ->    sub.     rd,ra,rb
   rd  o= ra - rb      ->    subo     rd,ra,rb
   rd .o= ra - rb      ->    subo.    rd,ra,rb
   rd   = ra - simm    ->    subi     rd,ra,simm

5. Multiplication

   rd   = ra * rb      ->    mullw    rd,ra,rb
   rd  .= ra * rb      ->    mullw.   rd,ra,rb
   rd  o= ra * rb      ->    mullwo   rd,ra,rb
   rd .o= ra * rb      ->    mullwo.  rd,ra,rb
   rd   = ra * simm    ->    mulli    rd,ra,simm

6. Division

   rd   = ra / rb      ->    divw     rd,ra,rb
   rd  .= ra / rb      ->    divw.    rd,ra,rb
   rd  o= ra / rb      ->    divwo    rd,ra,rb
   rd .o= ra / rb      ->    divwo.   rd,ra,rb
```

1.67 Logical Operations

```
1. And

   rd   = ra & rb    -> and      rd,ra,rb
   rd  .= ra & rb    -> and.     rd,ra,rb
   rd  .= ra & uimm  -> andi.    rd,ra,uimm

2. Or

   rd   = ra | rb    -> or       rd,ra,rb
   rd  .= ra | rb    -> or.      rd,ra,rb
   rd   = ra | uimm  -> ori      rd,ra,uimm

3. Exclusive or

   rd   = ra ^ rb    -> xor      rd,ra,rb
   rd  .= ra ^ rb    -> xor.     rd,ra,rb
   rd   = ra ^ uimm  -> xori     rd,ra,uimm

4. Shifts

   rd   = shiftl(ra,rb)     -> slw    rd,ra,rb
   rd  .= shiftl(ra,rb)     -> slw.   rd,ra,rb
   rd   = shiftl(ra,uimm)   -> slwi   rd,ra,uimm
   rd  .= shiftl(ra,uimm)   -> slwi.  rd,ra,uimm

   rd   = lshiftr(ra,rb)    -> srw    rd,ra,rb
   rd  .= lshiftr(ra,rb)    -> srw.   rd,ra,rb
   rd   = lshiftr(ra,uimm)  -> srwi   rd,ra,uimm
   rd  .= lshiftr(ra,uimm)  -> srwi.  rd,ra,uimm

   rd   = ashiftr(ra,rb)    -> sraw   rd,ra,rb
   rd  .= ashiftr(ra,rb)    -> sraw.  rd,ra,rb
   rd   = ashiftr(ra,uimm)  -> srawi  rd,ra,uimm
   rd  .= ashiftr(ra,uimm)  -> srawi. rd,ra,uimm
```

1.68 Floating-point operations

```
1. Moving data from register to register

   fd   =  fa             -> fmr      fd,fa
   fd  .=  fa             -> fmr.     fd,fa

2. Negation

   fd   =  -fa            -> fneg     fd,fa
   fd  .=  -fa            -> fneg.    fd,fa

3. Addition

   fd   =  fa + fb        -> fadd     fd,fa,fb
   fd  .=  fa + fb        -> fadd.    fd,fa,fb
   fd  s=  fa + fb        -> fadds    fd,fa,fb
   fd .s=  fa + fb        -> fadds.   fd,fa,fb
```

4. Subtraction

```
fd   =  fa - fb        -> fsub     fd,fa,fb
fd  .=  fa - fb        -> fsub.    fd,fa,fb
fd  s=  fa - fb        -> fsubs    fd,fa,fb
fd .s=  fa - fb        -> fsubs.   fd,fa,fb
```

5. Multiplication

```
fd   =  fa * fb        -> fmul     fd,fa,fb
fd  .=  fa * fb        -> fmul.    fd,fa,fb
fd  s=  fa * fb        -> fmuls    fd,fa,fb
fd .s=  fa * fb        -> fmuls.   fd,fa,fb
```

6. Division

```
fd   =  fa / fb        -> fdiv     fd,fa,fb
fd  .=  fa / fb        -> fdiv.    fd,fa,fb
fd  s=  fa / fb        -> fdivs    fd,fa,fb
fd .s=  fa / fb        -> fdivs.   fd,fa,fb
```

7. Multiplication and addition

```
fd   =  fa * fb + fc   -> fmadd    fd,fa,fb,fc
fd  .=  fa * fb + fc   -> fmadd.   fd,fa,fb,fc
fd  s=  fa * fb + fc   -> fmadds   fd,fa,fb,fc
fd .s=  fa * fb + fc   -> fmadds.  fd,fa,fb,fc
```

8. Multiplication and subtraction

```
fd   =  fa * fb - fc   -> fmsub    fd,fa,fb,fc
fd  .=  fa * fb - fc   -> fmsub.   fd,fa,fb,fc
fd  s=  fa * fb - fc   -> fmsubs   fd,fa,fb,fc
fd .s=  fa * fb - fc   -> fmsubs.  fd,fa,fb,fc
```

9. Multiplication and addition with negation

```
fd   =  -(fa * fb + fc) -> fnmadd   fd,fa,fb,fc
fd  .=  -(fa * fb + fc) -> fnmadd.  fd,fa,fb,fc
fd  s=  -(fa * fb + fc) -> fnmadds  fd,fa,fb,fc
fd .s=  -(fa * fb + fc) -> fnmadds. fd,fa,fb,fc
```

10. Multiplication and subtraction with negation

```
fd   =  -(fa * fb - fc) -> fnmsub   fd,fa,fb,fc
fd  .=  -(fa * fb - fc) -> fnmsub.  fd,fa,fb,fc
fd  s=  -(fa * fb - fc) -> fnmsubs  fd,fa,fb,fc
fd .s=  -(fa * fb - fc) -> fnmsubs. fd,fa,fb,fc
```

11. Rounding

```
fd   =  frsp(fa)       -> frsp     fd,fa
fd  .=  frsp(fa)       -> frsp.    fd,fa
fd   =  fctiw(fa)      -> fctiw    fd,fa
fd  .=  fctiw(fa)      -> fctiw.   fd,fa
fd   =  fctiwz(fa)     -> fctiwz   fd,fa
```

```
   fd  .=  fctiwz(fa)     -> fctiwz.  fd,fa

12. Estaminate

   fd   =  fres(fa)       -> fres     fd,fa
   fd  .=  fres(fa)       -> fres.    fd,fa

13. Etaminate of a square root

   fd   =  frsqrte(fa)    -> frsqrte  fd,fa
   fd  .=  frsqrte(fa)    -> frsqrte. fd,fa

14. Absoulute value

   fd   =  fabs(fa)       -> fabs     fd,fa
   fd  .=  fabs(fa)       -> fabs.    fd,fa

15. Negated absoulute value

   fd   =  fnabs(fa)      -> fnabs    fd,fa
   fd  .=  fnabs(fa)      -> fnabs.   fd,fa

16. Square root

   fd   =  fsqrt(fa)      -> fsqrt    fd,fa
   fd  .=  fsqrt(fa)      -> fsqrt.   fd,fa
   fd  s=  fsqrt(fa)      -> fsqrts   fd,fa
   fd .s=  fsqrt(fa)      -> fsqrts.  fd,fa

17. Floatig-point select

   fd   =  fsel(fa,fb,fc) -> fsel     fd,fa,fb,fc
   fd  .=  fsel(fa,fb,fc) -> fsel.    fd,fa,fb,fc
```

1.69 Memory acceses

```
1. Loading integers

   rd   =  d(ra.b)   -> lbz      rd,d(ra)
   rd   =  d(ra.w)   -> lhz      rd,d(ra)
   rd   =  d(ra.wa)  -> lha      rd,d(ra)
   rd   =  d(ra.l)   -> lwz      rd,d(ra)
   rd   =  d(ra)     -> lwz      rd,d(ra)

2. Loading integers with update

   rd   =  d[ra.b]   -> lbzu     rd,d(ra)
   rd   =  d[ra.w]   -> lhzu     rd,d(ra)
   rd   =  d[ra.wa]  -> lhau     rd,d(ra)
   rd   =  d[ra.l]   -> lwzu     rd,d(ra)
   rd   =  d[ra]     -> lwzu     rd,d(ra)

3. Loading integers indexed

   rd   =  (ra.b,rb) -> lbzx     rd,ra,rb
```

```
   rd   =  (ra.w,rb) -> lhzx     rd,ra,rb
   rd   =  (ra.wa,rb)-> lhax     rd,ra,rb
   rd   =  (ra.l,rb) -> lwzx     rd,ra,rb
   rd   =  (ra,rb)   -> lwzx     rd,ra,rb

4. Loading integers indexed with update

   rd   =  [ra.b,rb] -> lbzux    rd,ra,rb
   rd   =  [ra.w,rb] -> lhzux    rd,ra,rb
   rd   =  [ra.wa,rb]-> lhaux    rd,ra,rb
   rd   =  [ra.l,rb] -> lwzux    rd,ra,rb
   rd   =  [ra,rb]   -> lwzux    rd,ra,rb

5. Storing integers

   d(ra.b) =  rd     -> stb      rd,d(ra)
   d(ra.w) =  rd     -> sth      rd,d(ra)
   d(ra.l) =  rd     -> stw      rd,d(ra)
   d(ra)   =  rd     -> stw      rd,d(ra)

6. Storing integers with update

   d[ra.b] =  rd     -> stbu     rd,d(ra)
   d[ra.w] =  rd     -> sthu     rd,d(ra)
   d[ra.l] =  rd     -> stwu     rd,d(ra)
   d[ra]   =  rd     -> stwu     rd,d(ra)

7. Storing integers indexed

   (ra,rb.b) = rd    -> stbx     rd,ra,rb
   (ra,rb.w) = rd    -> sthx     rd,ra,rb
   (ra,rb.l) = rd    -> stwx     rd,ra,rb
   (ra,rb)   = rd    -> stwx     rd,ra,rb

8. Storing integers indexed with update

   [ra,rb.b] = rd    -> stbux    rd,ra,rb
   [ra,rb.w] = rd    -> sthux    rd,ra,rb
   [ra,rb.l] = rd    -> stwux    rd,ra,rb
   [ra,rb]   = rd    -> stwux    rd,ra,rb

9. Loading floats

   fd   =  d(ra.s)   -> lfs      fd,d(ra)
   fd   =  d(ra.d)   -> lfd      fd,d(ra)
   fd   =  d(ra)     -> lfd      fd,d(ra)

10. Loading floats with update

   fd   =  d[ra.s]   -> lfsu     fd,d(ra)
   fd   =  d[ra.d]   -> lfdu     fd,d(ra)
   fd   =  d[ra]     -> lfdu     fd,d(ra)

11. Loading floats indexed

   fd   =  (ra.s,rb) -> lfsx     fd,ra,rb
   fd   =  (ra.d,rb) -> lfdx     fd,ra,rb
```

```
   fd   =  (ra,rb)   -> lfdx     fd,ra,rb

12. Loading floats indexed with update

   fd   =  [ra.s,rb] -> lfsux    fd,ra,rb
   fd   =  [ra.d,rb] -> lfdux    fd,ra,rb
   fd   =  [ra,rb]   -> lfdux    fd,ra,rb

13. Storing floats

   d(ra.s)   = fn    -> stfs     fn,d(ra)
   d(ra.d)   = fn    -> stfd     fn,d(ra)
   d(ra)     = fn    -> stfd     fn,d(ra)

14. Storing floats with update

   d[ra.s]   = fn    -> stfsu    fn,d(ra)
   d[ra.d]   = fn    -> stfdu    fn,d(ra)
   d[ra]     = fn    -> stfdu    fn,d(ra)

15. Storing floats indexed

   (ra.s,rb) = fn    -> stfsx    fn,ra,rb
   (ra.d,rb) = fn    -> stfdx    fn,ra,rb
   (ra,rb) = fn      -> stfdx    fn,ra,rb

16. Storing floats indexed with update

   [ra.s,rb] = fn    -> stfsu    fn,ra,rb
   [ra.d,rb] = fn    -> stfdux   fn,ra,rb
   [ra,rb] = fn      -> stfdux   fn,ra,rb

17. If you want to acces variables from data secton, you can use following
    shortcuts

   *label.b  = rn    -> stb      rn,label(r2)
   *labal.w  = rn    -> sth      rn,label(r2)
   *label.l  = rn    -> stw      rn,label(r2)
   *label    = rn    -> stw      rn,label(r2)

   *label.s  = fn    -> stfs     fn,label(r2)
   *label.d  = fn    -> stfd     fn,label(r2)
   *label    = fn    -> stfd     fn,label(r2)

   rn = *label.b     -> lbz      rn,label(r2)
   rn = *labal.w     -> lhz      rn,label(r2)
   rn = *label.wa    -> lha      rn,label(r2)
   rn = *label.l     -> lwz      rn,label(r2)
   rn = *label       -> lwz      rn,label(r2)

   fn = *label.s     -> lfs      fn,label(r2)
   fn = *label.d     -> lfd      fn,label(r2)
   fn = *label       -> lfd      fn,label(r2)
```

1.70 Comprasions

```
1. Floating-point comprasions

   crfd = fcmpo(fa,fb)     -> fcmpo crfd,fa,fb
   crfd = fcmpu(fa,fb)     -> fcmpu crfd,fa,fb

2. Integer comprasions

   crfd = cmpw(ra,rb)      -> cmp   crfd,0,ra,rb
   crfd = cmpw(ra,simm)    -> cmpi  crfd,0,ra,simm
   crfd = cmplw(ra,rb)     -> cmpl  crfd,0,ra,rb
   crfd = cmplw(ra,uimm)   -> cmpli crfd,0,ra,uimm
   crfd = cmpd(ra,rb)      -> cmp   crfd,1,ra,rb
   crfd = cmpd(ra,simm)    -> cmpi  crfd,1,ra,simm
   crfd = cmpld(ra,rb)     -> cmpl  crfd,1,ra,rb
   crfd = cmpld(ra,uimm)   -> cmpli crfd,1,ra,uimm

crfd field may be left, in that case cr0 is used as default.
```

1.71 Building structures

```
In PtnAsm abilities of SO and FO has increased. You don't have to remember
types of structure's fields. They are stored together with their offset.
Assembler will detect them and use coresponding instruction.

Example:

   clrso
sphere_r             so.d  1
sphere_coordinates   so.s  3
sphere_texture_ptr   so.l  1
sphere_id            so.w  1
sphere_size          soval

   r10   =  sphere_id(r3)             lhz  r10,sphere_id(r3)
   f1    =  sphere_coordinates(r3)    lfs  f1,sphere_coordinates(r3)
   f2    =  sphere_coordinates[1](r3) lfs  f2,sphere_coordinates+4(r3)
   f3    =  sphere_r(r3)              lfd  f3,sphere_r(r3)

You can also use notation known from C language:

   r10   =  r3->sphere_id             lhz  r10,sphere_id(r3)
```

1.72 Predefined symbos

```
   NARG       -  Number of macro parameters. See Macro
   CARG       -  Curent argument. See Macro
   __FO       -  Current value of FO counter. See FO
   __SO       -  Current value of RS/SO counter. See SO
   __RS       -  Current value of RS/SO counter. See SO
   __MODE
```

```
   __NEAR     -  These two symbols are present to keep compatibility
                 with powerpc/ppcmacros.i file. In orginal they were used
                 to determinate data model. PtnAsm always works in small
                 data model so they have the same value.
   _POWERMODE -  Similar as above.
```

1.73 Thanks

```
I would like to thank:

- Jaroslaw 'Mavey^PTN' Wojczakowski
- Frank Wille
```

1.74 registration

```
If you have any ideas, sugestions or bug reports please contact me.
Unregistrated version of PtnAsm can't produce code larger than 4096 kb.
If you'd like to register please send $15 (or 15 zl for poles)
and your e-mail address at:

Rafal Grembowski
Slowianska 17/40
85-163 Bydgoszcz
Poland

diamond@go2.pl
```

1.75 addres