

**ppctut**

**COLLABORATORS**

	<i>TITLE :</i> ppctut		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		April 18, 2022	

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>ppctut</b>	<b>1</b>
1.1	MagicSNs PowerPC ASM Coding Tutorial	1
1.2	MagicSNs PowerPC ASM Coding Tutorial	2
1.3	MagicSNs PowerPC ASM Coding Tutorial	3
1.4	MagicSNs PowerPC ASM Coding Tutorial	4
1.5	MagicSNs PowerPC ASM Coding Tutorial	4
1.6	MagicSNs PowerPC ASM Coding Tutorial	5
1.7	MagicSNs PowerPC ASM Coding Tutorial	9
1.8	MagicSNs PowerPC ASM Coding Tutorial	9
1.9	MagicSNs PowerPC ASM Coding Tutorial	10
1.10	MagicSNs PowerPC ASM Coding Tutorial	11
1.11	MagicSNs PowerPC ASM Coding Tutorial	12
1.12	MagicSNs PowerPC ASM Coding Tutorial	12
1.13	MagicSNs PowerPC ASM Coding Tutorial	13
1.14	MagicSNs PowerPC ASM Coding Tutorial	14
1.15	MagicSNs PowerPC ASM Coding Tutorial	14
1.16	MagicSNs PowerPC ASM Coding Tutorial	14
1.17	MagicSNs PowerPC ASM Coding Tutorial	16
1.18	MagicSNs PowerPC ASM Coding Tutorial	16
1.19	MagicSNs PowerPC ASM Coding Tutorial	20
1.20	MagicSNs PowerPC ASM Coding Tutorial	21
1.21	MagicSNs PowerPC ASM Coding Tutorial	24
1.22	MagicSNs PowerPC ASM Coding Tutorial	27
1.23	MagicSNs PowerPC ASM Coding Tutorial	27
1.24	MagicSNs PowerPC ASM Coding Tutorial	29
1.25	MagicSNs PowerPC ASM Coding Tutorial	30
1.26	MagicSNs PowerPC ASM Coding Tutorial	32
1.27	MagicSNs PowerPC ASM Coding Tutorial	33
1.28	MagicSNs PowerPC ASM Coding Tutorial	33
1.29	MagicSNs PowerPC ASM Coding Tutorial	34

---

---

1.30 MagicSNs PowerPC ASM Coding Tutorial . . . . .	35
1.31 MagicSNs PowerPC ASM Coding Tutorial . . . . .	35
1.32 MagicSNs PowerPC ASM Coding Tutorial . . . . .	35
1.33 MagicSNs PowerPC ASM Coding Tutorial . . . . .	36
1.34 MagicSNs PowerPC ASM Coding Tutorial . . . . .	37
1.35 MagicSNs PowerPC ASM Coding Tutorial . . . . .	38
1.36 MagicSNs PowerPC ASM Coding Tutorial . . . . .	39
1.37 MagicSNs PowerPC ASM Coding Tutorial . . . . .	39
1.38 MagicSNs PowerPC ASM Coding Tutorial . . . . .	40
1.39 MagicSNs PowerPC ASM Coding Tutorial . . . . .	42
1.40 MagicSNs PowerPC ASM Coding Tutorial . . . . .	43
1.41 MagicSNs PowerPC ASM Coding Tutorial . . . . .	44
1.42 MagicSNs PowerPC ASM Coding Tutorial . . . . .	44
1.43 MagicSNs PowerPC ASM Coding Tutorial . . . . .	45
1.44 MagicSNs PowerPC ASM Coding Tutorial . . . . .	46
1.45 MagicSNs PowerPC ASM Coding Tutorial . . . . .	46
1.46 MagicSNs PowerPC ASM Coding Tutorial . . . . .	46
1.47 MagicSNs PowerPC ASM Coding Tutorial . . . . .	47
1.48 MagicSNs PowerPC ASM Coding Tutorial . . . . .	49
1.49 MagicSNs PowerPC ASM Coding Tutorial . . . . .	49
1.50 MagicSNs PowerPC ASM Coding Tutorial . . . . .	51
1.51 MagicSNs PowerPC ASM Coding Tutorial . . . . .	51
1.52 MagicSNs PowerPC ASM Coding Tutorial . . . . .	52
1.53 MagicSNs PowerPC ASM Coding Tutorial . . . . .	52
1.54 MagicSNs PowerPC ASM Coding Tutorial . . . . .	53
1.55 MagicSNs PowerPC ASM Coding Tutorial . . . . .	53
1.56 MagicSNs PowerPC ASM Coding Tutorial . . . . .	53
1.57 MagicSNs PowerPC ASM Coding Tutorial . . . . .	54
1.58 MagicSNs PowerPC ASM Coding Tutorial . . . . .	54
1.59 MagicSNs PowerPC ASM Coding Tutorial . . . . .	54
1.60 MagicSNs PowerPC ASM Coding Tutorial . . . . .	55
1.61 MagicSNs PowerPC ASM Coding Tutorial . . . . .	55
1.62 MagicSNs PowerPC ASM Coding Tutorial . . . . .	55
1.63 MagicSNs PowerPC ASM Coding Tutorial . . . . .	56
1.64 MagicSNs PowerPC ASM Coding Tutorial . . . . .	56
1.65 MagicSNs PowerPC ASM Coding Tutorial . . . . .	56
1.66 MagicSNs PowerPC ASM Coding Tutorial . . . . .	57
1.67 MagicSNs PowerPC ASM Coding Tutorial . . . . .	57
1.68 MagicSNs PowerPC ASM Coding Tutorial . . . . .	57

---

---

1.69 MagicSNs PowerPC ASM Coding Tutorial . . . . .	58
1.70 MagicSNs PowerPC ASM Coding Tutorial . . . . .	58
1.71 MagicSNs PowerPC ASM Coding Tutorial . . . . .	58
1.72 MagicSNs PowerPC ASM Coding Tutorial . . . . .	59
1.73 MagicSNs PowerPC ASM Coding Tutorial . . . . .	59
1.74 MagicSNs PowerPC ASM Coding Tutorial . . . . .	59
1.75 MagicSNs PowerPC ASM Coding Tutorial . . . . .	60
1.76 MagicSNs PowerPC ASM Coding Tutorial . . . . .	61
1.77 MagicSNs PowerPC ASM Coding Tutorial . . . . .	62
1.78 MagicSNs PowerPC ASM Coding Tutorial . . . . .	64
1.79 MagicSNs PowerPC ASM Coding Tutorial . . . . .	65
1.80 MagicSNs PowerPC ASM Coding Tutorial . . . . .	66
1.81 MagicSNs PowerPC ASM Coding Tutorial . . . . .	68
1.82 MagicSNs PowerPC ASM Coding Tutorial . . . . .	69
1.83 MagicSNs PowerPC ASM Coding Tutorial . . . . .	70
1.84 MagicSNs PowerPC ASM Coding Tutorial . . . . .	72

---

# Chapter 1

## ppctut

### 1.1 MagicSNs PowerPC ASM Coding Tutorial

MagicSNs PowerPC ASM Coding Tutorial

Table of Contents

Part A : [A Quick Start](#)

Part B : [A First Program](#)

Part C : [Important Things](#)

Part D : [Load/Store Commands](#)

Part E : [Branch and Compare Commands](#)

Part F : [Logic Commands](#)

Part G : [Extension Commands](#)

Part H : [Integer Arithmetic Commands](#)

Part I : [Rotate and Shift Commands](#)

Part J : [Multiple Load/Store Operations](#)

Part K : [Condition-Logic Commands](#)

Part L : [Accessing Special Purpose Registers](#)

Part M : [The FPU of the PowerPC](#)

Part N : [Additional StormPowerASM Macros](#)

Part O : [Contextswitches and Mixed Binary](#)

Part P : [Some words about Optimizing](#)

Part Q : [Some words about 68k->PPC ASM Porting](#)

Part R : [History](#)

Part S : [Literature and Software](#)

Note: This tutorial only handles the stuff which will be USEFUL. Commands that are mainly of interest for programmers of Operating Systems (like the Reserved Commands and the trap commands) are not described. Also i did not describe the Interrupt-System of the PPC, as with both PPC Kernels existing

---

on Amiga you should not do this interrupt-stuff yourselves, else problems could appear. I also did not cover the MMU stuff - use the `powerpc.library` commands to use the MMU.

Another Note: This Tutorial is Copyright by MagicSN@Birdland.es.bawue.de. Spread it wherever you like, but don't change it (conversion to HTML or other formats is allowed, though). And have a look at the unofficial PowerUP Site for an update soon :)

Yet another note: Some of the Examples are taken from the StormPowerASM Docs, with the permission of Sam Jordan (the author).

## 1.2 MagicSNs PowerPC ASM Coding Tutorial

### Literature and Software

I recommend the following literature:

- PowerPC Microprocessor family : The Programming Environment

This book is available as book and as PDF File. The PDF File is found on the Motorola Site <http://www.mot.com>, and it is quite big (several MB). You can display PDF Files on your Amiga with `xpdf 0.7`, this program can also convert them to Postscript. You can print Postscript files with Ghostscript. Alternatively you can convert PDF Files to ASCII with `pdftotxt` (but you lose the correct display of all tables then...). In PDF, the document is >800 pages. I wrote this tutorial mainly to have a quick reference as replacement for this PDF File. `xpdf` is soooooo slow.

- PowerPC 603/603E/604 User Manuals

Might be interesting for some special knowledge also. Also available as books and as PDF File on <http://www.mot.com>. Also quite big.

There is some stuff found about Optimizing in these manuals.

- Optimizing PowerPC Code (Addison Wesley)

This is mainly about Optimizing PowerPC ASM Code.

- The documentation of WarpOS and StormPowerASM, from which i also took some of the examples in this document.

I Recommend the following software:

- WarpOS

This is a PowerPC Native Kernel compatible to the AmigaDOS, done by Haage&Partner. It exists in two versions: One of the versions (V7.0) is compatible to Phase 5's Kernel (but it has also the disadvantages of Phase 5's Kernel), the other version (V12+, currently V13.11 is the most recent version) provides several advantages. Well, version 7 is not really a kernel,

---

to tell the truth, it is just a wrap-around around Phase 5s kernel, only Version 8+ is a real kernel. Quite some features only exist for version 8+.

WarpOS is downloadable from <ftp://ftp.haage-partner.com/WarpUP>. Actually there are two archives found there, the base V12 Archive (with also the V7 executables) and an Update to V12.2 which fixes a bug in the initial release that caused a slowdown in memory access (a register was set incorrect).

#### - StormPowerASM

The current only serious PowerPC ASM on Amiga. It is very good. If you know the 68k ASM PhxAss, you will get used to StormPowerASM soon. It is very similar, in certain ways. It needs WarpOS to be installed.

Theoretically there is also GAS for the Phase 5 Software, but GAS was never meant to be a user-level Assembler. It was only meant to translate the output of GNU C. Also i have to note: All the things about Contextswitch, function design, Mixed/Fat Binary... in this document only apply to using StormPowerASM and WarpOS. Many of the commands explained above are not Standard PPC ASM, but StormPowerASM Macros.

#### - rtgmaster.library PPC

In case you want to code demos/games with the PPC, this will be interesting for you. It is a system to produce Video-Hardware-Independent code. The especially interesting about it is, that you get all Video Refresh functions PPC Native, so that you do not need to do a Contextswitch for Video Refresh. There is not yet a public release, but in case you need it, you can get the current executable by asking me for it ([MagicSN@Birdland.es.bawue.de](mailto:MagicSN@Birdland.es.bawue.de)). I hope to release the first version of rtgmaster.library PPC soon. rtgmaster.library PPC of course will be API-compatible to the 68k version.

## 1.3 MagicSNs PowerPC ASM Coding Tutorial

The FPU of the PowerPC

Part 1 : [The FPU: An Overview](#)

Part 2 : [FPU Arithmetic Commands](#)

Part 3 : [FPU Compare Commands](#)

Part 4 : [FPU Rounding+Conversion](#)

Part 5 : [FPU Multiply-Add Commands](#)

Part 6 : [FPU Move Commands](#)

Part 7 : [FPU Load/Store Commands](#)

Part 8 : [FPU StormPowerASM Macros](#)

Part 9 : [FPU Status and Control](#)

---



Part A : [FPU: \(De-\)Normalization](#)

Part B : [FPU: Exceptions](#)

Part C : [FPU: Conversion Examples](#)

Note: As with the Integer-Part, i only handle the things i consider useful. For example i only describe the Exception-Mechanism as far as i think the usual programmer wil need it.

Especially have a look at the Multiply-Add Commands. They are really VERY fast. For example the Cybermand Program (the one that zooms into the Mandelbrot set in realtime) uses them quite extensive. These commands only need very few CPU Cycles.

At this place i again remark, that the . Notation works in a different way for FPU commands than for Integer commands. For FPU Commands it only sets certain Exception Bits in cr1. If you want to compare FPU Numbers, use FCMP\* !!!

## 1.4 MagicSNs PowerPC ASM Coding Tutorial

FPU: (De-)Normalization

The PPC FPU also can handle denormalized numbers. A denormalized number is a number, which has an exponent of 0, and a significand  $\neq 0$ . This can happen after a very small result appeared.

With the Precision of

Doubles this normally should not happen, especially as it normally does not make sense to add very big and very small numbers together, in this way. But well, in case it happens the PPC is able to handle these numbers too (the 060 was not able to handle them, BTW, on the 060 the 68060.library had to provide emulation code for this).

## 1.5 MagicSNs PowerPC ASM Coding Tutorial

FPU: Exceptions

FPU Exceptions are a difficult matter on the PPC, and the normal coder does not have to know all about it. So i only will discuss the basics here, and won't discuss the stuff only interesting for OS designers. If you need further information, have a look at the Motorola documentation.

You should first read the chapter about [FPSCR and XER](#) .

Important Bits of the FPSCR:

FX: If an \*exception\* bit of the FPSCR was set from 0 to 1, this one is set.

If this Bit is set, the Exception handler of WarpOS is invoked.

FEX: Some Exceptions can be enabled/disabled. This bit shows you which of them this would be. If such an exception is disabled, a default result is given.

VX/VE: Control the Invalid Operation Exception

OX/OE: Control the Overflow Exception

UX/UE: Control the Underflow Exception

ZX/ZE: Control the Division by Zero Exception

XX/XE: Controls the FPU Inexact Exception. If XE is set and XX happens, and FI is changed from 0 to 1, the exception happens

VXSNAN: Operation used a Signalling "Not-A-Number" Result (if you do an unordered compare, this would only be a Quiet "Not a Number", which is valid).

VXISI/VXIDI/VXIMI: Exceptions dealing with arithmetics with Infinity

VXVC: Invalid Compare Exception

FR: Indicates that rounding happened (does not cause an exception)

FI: Indicates that the result is inexact (only causes an exception, if a XX exception happens, and if XE is set)

VXSQRT: Exception concerning SQRT

VXCVI: Exception concerning Conversion

The other Bits should not bother you.

You normally should not change these bits. If you for some reason want to do this, use the provided functions of powerpc.library.

In your place i would not bother too much with the FPSCR. If you don't change the FPSCR Bits (might be useful for Debugging) no Exception will happen. The FPU Exceptions are turned off in WarpOS as default.

## 1.6 MagicSNs PowerPC ASM Coding Tutorial

FPU: Conversion Examples

Because of the importance i am giving some FPU conversion examples out of the Motorola Docs here (Remember, that PowerUP currently only supports 32 Bit PPCs !!!)

First some basics:

For the standard FPU->Int Conversion you use fctiw, store the result from the FPU-register into the memory, and load it then from memory with a General purpose register. The standard Int->FPU Conversion is not that easy on a 32 Bit machine, below a Macro is given...

Only 64 Bit:

Floating Point Number -> Floating Integer

The number to be converted is in f1, the result will be in f3.

```
mtfsb0 23 ; clear VXCVI
```

```
fctid f3,f1 ; convert to int
```

```
fcfid f3,f3 ; convert back to double
```

```
mcrfs 7,5 ; VXCVI to CR
```

bf 31,\$+8 ; Jump, if VXCVI was 0

fmr f3,f1 ; input was fp int

Only 64 Bit:

Floating Point Number -> Signed Fixed-Point Integer Double-Word

A Doubleword at Offset disp of r1 is used as scratch-space,

the number to be converted is in f1, the result will be in r3.

fctid f2,f1 ; convert to dword int

stfd f2,disp(r1) ; store float

ld r3,disp(r1) ; load dword

Only 64 Bit:

Floating Point Number -> Unsigned Fixed-Point Integer Double-Word

The value to be converted is in f1, the value 0 in f0, the value

$2^{64}-2048$  in f3, the value  $2^{63}$  in f4 and r4, the result is given

in g3, and a double word at Offset disp of r1 is used as scratch-space.

fsel f2,f1,f1,f0 ; use 0, if <0

fsub f5,f3,f1 ; use max, if >max

fsel f2,f5,f2,f3

fsub f5,f2,f4 ; subtract  $2^{63}$

fcmpu cr2,f2,f4 ; use diff, if  $\geq 2^{63}$

fsel f2,f5,f5,f2

fctid f2,f2 ; convert to dword int

stfd f2,disp(r1) ; store float

ld r3,disp(r1) ; load dword

blt cr2,\$+8 ; add  $2^{63}$  if input was  $\geq 2^{63}$

add r3,r3,r4

32+64 Bit:

Floating Point Number -> Signed Fixed-Point Integer Word

A Double Word at Offset disp of r1 is used as scratch-space, the

number to be converted is in f1, the result will be in r3.

fctiw f2,f1 ; convert to int

stfd f2,disp(r1) ; store float

lwa r3,disp+4(r1) ; load algebraic word

32+64 Bit:

Floating Point Number -> Unsigned Fixed-Point Integer Word

This conversion works in a different way in 32 Bit than it does in 64 Bit.

64 Bit:

The value to be converted is assumed in f1, the value 0 in f0, the value

$2^{32}-1$  in f3, the result is returned in r3, and a doubleword at Offset disp

is used as Scratch-Space.

fsel f2,f1,f1,f0 ; use 0, if <0  
 fsub f4,f3,f1 ; use max, if >max  
 fsel f2,f4,f2,f3  
 fctid f2,f2 ; convert to dword int  
 stfd f2,disp(r1) ; store float  
 lwz r3,disp+4(r1) ; load word and zero

32 Bit:

Additionally to the 64 Bit version, for the 32 Bit version, the value  $2^{31}$  is needed in f4.

fsel f2,f1,f1,f0 ; use 0, if <0  
 fsub f5,f3,f1 ; use max, if >max  
 fsel f2,f5,f2,f3  
 fsub f5,f2,f4 ; subtract  $2^{31}$   
 fcmpu cr2,f2,f4 ; use diff, if  $>2^{31}$   
 fsel f2,f5,f5,f2  
 fctiw f2,f2 ; convert to int  
 stfd f2,disp(r1) ; store float  
 lwz r3,disp+4(r1) ; load word  
 blt cr2,\$+8 ; add  $2^{31}$ , if input was  $\geq 2^{31}$   
 xoris r3,r3,0x8000

Only 64 Bit:

Signed Fixed-Point Integer Double-Word -> Floating Point Number

The value to be converted is assumed in r3, the result will be put in f1, and a double word at Offset disp of r1 is used as scratch-space.

std r3,disp(r1) ; store dword  
 lfd f1,disp(r1) ; load float  
 fcfid f1,f1 ; convert to fpu int

Only 64 Bit:

Unsigned Fixed-Point Integer Double-Word -> Floating Point Number

The value to be converted is assumed in r3, the result will be put in f1, the value  $2^{32}$  is assumed in f4, and two double words at Offset Disp of r1 are used as scratch-space.

rldicl r2,r3,32,32 ; Isolate High Half  
 rldicl r0,r3,0,32 ; Isolate Low Half  
 std r2,disp(r1) ; store dword both  
 std r0,disp+8(r1)  
 lfd f2,disp(r1) ; load float both  
 lfd f1,disp+8(r1)  
 fcfid f2,f2 ; convert each half to fpu int

```
fcfid f1,f1
```

```
fmadd f1,f4,f2,f1 ; 2^32*high+low
```

If Rounding is defined towards +infinity or towards -infinity or if it is acceptable, that the converted number is either of the two representable FP-numbers nearest to the given fixed-point number (and only then !!!)

the following shorter version can be used (the registers are defined as usual, only that the value in f4 is not needed, f2 is assumed to be  $2^{64}$ ):

```
std r3,disp(r1) ; store dword
```

```
lfd f1,disp(r1) ; load float
```

```
fcfid f1,f1 ; convert to fpu int
```

```
fadd f4,f1,f2 ; add  $2^{64}$ 
```

```
fsel f1,f1,f1,f4 ; if r3<0
```

Only 64 Bit:

Signed Fixed-Point Integer Word -> Floating Point Number

It is assumed, that the value to be converted is found in r3, the result is returned in f1, and a double word at Offset disp of r1 is used as scratch-space.

```
extsw r3,r3 ; sign-extension
```

```
std r3,disp(r1) ; store dword
```

```
lfd f1,disp(r1) ; load float
```

```
fcfid f1,f1 ; convert to fpu int
```

Only 64 Bit:

Unsigned Fixed-Point Integer-Word -> Floating Point Number

It is assumed, that the value to be converted is found in r3, the result is returned in f1, and a double word at Offset disp of r1 is used as scratch-space.

```
rldicl r0,r3,0,32 ; 0-extend
```

```
std r0,disp(r1) ; store dword
```

```
lfd f1,disp(r1) ; load float
```

```
fcfid f1,f1 ; convert to fpu int
```

And now: The Macro for Int->FPU Conversion for 32 Bit Machines (will also work on 64 Bit machines). Note: This macro only works in Small-Data.

```
citf macro
```

```
xoris trash,\3,$8000
```

```
sw trash,_CITF_TEMP+4
```

```
If \1,_CITF_TEMP
```

```
fsub \1,\1,\2
```

```
endm
```

Parameter 1 is the destination FP-register, parameter 2 the intermediate FP-register (which MUST hold the value \$4330000080000000, parameter 3 the source GP-register).

## 1.7 MagicSNs PowerPC ASM Coding Tutorial

### FPU Status and Control

This section provides some commands to modify the FPSCR register (the FPU Control register), or to check its data. With the exception of `mcrfs`, all these commands support the `.` Notation like defined in

**Important things** . The `o` Notation is not supported, though.

Also have a look at **FPSCR and XER** .

#### `mffs Fd`

This command copies the contents of the FPSCR to the low-order of `Fd`.

The high-order is undefined.

#### `mcrfs crfd,crfs`

This command copies the contents of the FPSCR field `crfd` **Condition field Notation** to the CR field `crfD`. All exception bits copied, with the exception of `FEX` and `VX`, are deleted.

#### `mtfsfi crfd,IMM`

Copies the absolute value `IMM` to the Condition field `crfd` of FPSCR.

The contents of `FPSCR[FX]` are altered only, if `crfd = 0`.

#### `mtfsf FM,Fb`

The low-order of `Fb` is placed into the FPSCR, with the control of the mask `FM`. The mask identifies the fields (numbered from 0-7) to be affected, so it is a value between 0-127. The contents of `FPSCR[FX]` are only altered, if `FM[0]=1`.

#### `mtfsb0 crb`

The FPSCR Bit Location specified is cleared. Does not work with `FEX` and `VX`.

#### `mtfsb1 crb`

The FPSCR Bit Location specified is set. Does not work with `FEX` and `VX`.

## 1.8 MagicSNs PowerPC ASM Coding Tutorial

### FPU Rounding+Conversion

#### `frsp Fd,Fb`

This command rounds `Fb` to single-precision using the rounding mode specified in `FPSCR[RN]` and puts the result into `Rd`.

#### `fcfd Fd,Fb`

The 64 Bit Signed Integer in `Fb` is converted to a Double using rounding mode `FPSCR[RN]`. The result is put into `Fd`. This command only exists on 64 Bit Implementations.

#### `ftid Fd,Fb`

The Double in Fb is converted to a 64 Bit Signed Integer using rounding mode FPSCR[RN]. The result is put into Fd. This command only exists on 64 Bit Implementations.

fctidz Fd,Fb

The same like fctid, only that always "Round to 0" is used. Only exists on 64 Bit Implementations.

fctiw Fd,Fb

The Floating-Point operand in Fb is converted to a 32 Bit Signed Integer, using rounding mode FPSCR[RN]. It is placed in the low-order 32 Bits of Fd. Bits 0-31 of Fd are undefined.

fctiwz Fd,Fb

The same like fctiw, only that always rounding mode "Round to 0" is used. All Rounding+Conversion functions support the . Notation like described in **Important things** , but not the o Notation.

## 1.9 MagicSNs PowerPC ASM Coding Tutorial

Load/Store Commands

lfs Fd,d16(Ra)

lfsx Fd,Ra,Rb

lfsu Fd,d16(Ra)

lfsux Fd,Ra,Rb

lfd Fd,d16(Ra)

lfdx Fd,Ra,Rb

lfd� Fd,d16(Ra)

lfdû Fd,Ra,Rb

These are the commands to load data from memory to a Floating Point Register. The Data is interpreted as in Floating Point Format. In case of lfs\* it is interpreted as Single Precision, in case of lfd\* as Double Precision.

lfs/lfd are the "normal" Load-Commands, which load the EA=Ra+d16 (or d16+0, if a=0).

lfsx/lfdx use (Ra0)+Rb as EA instead. ("Indexed").

The versions with u added are "with Update", which means that Ra+d16 or Ra+Rb is written back to Ra at the end. r0 is not allowed as Ra for the versions "with Update".

stfs Fs,d16(Ra)

stfsx Fs,Ra,Rb

stfsu Fs,d16(Ra)

stfsux Fs,Ra,Rb

stfd Fs,d16(Ra)

stfdx Fs,Ra,Rb

stfdu Fs,d16(Ra)

stfdux Fs,Ra,Rb

These are the commands to store data from a Floating Point Register to memory.

The Data is stored in Floating Point Format. In case of stfs\* it is written

in Single Precision Format to the Memory, in case of stfd\* in Double Precision Format.

stfs/stfd are the "normal" Store-Commands, which store to the EA=Ra+d16 (or d16+0, if a=0).

stfsx/stfdx use (Ra0)+Rb as EA instead. ("Indexed").

The versions with u added are "with Update", which means that Ra+d16 or Ra+Rb is

written back to Ra at the end. r0 is not allowed as Ra for the versions "with Update".

Doubles take 64 Bit space in memory, Singles take 32 Bit.

stfiwx Fs,Ra,Rb

This command uses an EA of (Ra0)+Rb. The Contents of the low-order 32-Bits of Fs

are stored into the word given by the EA, without any conversion.

## 1.10 MagicSNs PowerPC ASM Coding Tutorial

FPU Multiply-Add Commands

The Multiply-Add Commands do something like  $a*b+c$  in one command. They are

very fast, and this way were useful for Matrix Multiplication. All these commands

support the . Notation like described in **Important things** , but not

the o Notation. The "normal" version is always the Double Precision version, the

one with added s the Single Precision Version.

fmadd Fd,Fa,Fb,Fc

fmadds Fd,Fa,Fb,Fc

fmsub Fd,Fa,Fb,Fc

fmsubs Fd,Fa,Fb,Fc

fnmadd Fd,Fa,Fb,Fc

fnmadds Fd,Fa,Fb,Fc

fnmsub Fd,Fa,Fb,Fc

fnmsubs Fd,Fa,Fb,Fc

fmadd/fmadds do  $Fd=Fa*Fb+Fc$

fmsub/fmsubs do  $Fd=Fa*Fb-Fc$

fnmadd/fnmadds do  $Fd=-(Fa*Fb+Fc)$

fnmsub/fnmsubs do  $Fd=-(Fa*Fb-Fc)$



## 1.11 MagicSNs PowerPC ASM Coding Tutorial

FPU Move Commands

fmr Fd,Fa

fneg Fd,Fa

fabs Fd,Fa

fnabs Fd,Fa

fmr copies Fa to Fd, fneg copies -Fa to Fd, fabs copies abs(Fa) to Fd, and fnabs copies -abs(Fa) to Fd.

All these commands support the . Notation, like described in

**Important things** , but not the o Notation.

## 1.12 MagicSNs PowerPC ASM Coding Tutorial

The FPU: An Overview

The FPU of the PowerPC knows two sorts of numbers: Single Precision 32 Bit Numbers and Double Precision 64 Bit Numbers.

Single Precision:  $+/-1.2 \cdot 10^{-38}$  to  $+/-3.4 \cdot 10^{38}$

Double Precision:  $+/-2.2 \cdot 10^{-308}$  to  $+/-1.8 \cdot 10^{308}$

The FPU conforms in both Precisions to the IEEE 754 Standard.

Single Precision Values can be converted automatically to Double Precision Values, but Double Precision to Single Precision Conversion has to be done manually.

Double Precision Calculations take Singles and Doubles as Parameter, the result is Double. Single Precision Calculations only take Singles as Parameter, the Result is Single.

The FPU has 32 64 Bit registers, known as f0-f31.

Floating Point Parameters of Subfunctions can be given in f1-f13, return values in f1-f4. f14-f31 have to be restored, if the subfunction changes them.

A Special case concerning FPU Operations are the "Not-A-Numbers". The "Not-A-Numbers" cause problems with a lot of calculation-commands, so it is advised, that you make security-checks, if you do not want CPU-Exceptions to happen. As default, the FPU Exceptions are switched off in WarpOS, though, so Exceptions cannot happen.

For Compare commands, there are two versions: One that causes an Exception, and one that sets a special "unordered" Bit in the Condition Field used.

## 1.13 MagicSNs PowerPC ASM Coding Tutorial

FPU Arithmetic Commands

fadd Fd,Fa,Fb

fadds Fd,Fa,Fb

fsub Fd,Fa,Fb

fsubs Fd,Fa,Fb

fmul Fd,Fa,Fb

fmuls Fd,Fa,Fb

fdiv Fd,Fa,Fb

fdivs Fd,Fa,Fb

fsqrt Fd,Fa

fsqrts Fd,Fa

These commands add, subtract, multiply, divide and calculate the square root of FPU numbers. The "normal" one is always the Double Precision Command, the one with the s at the end the Single Precision Command. The Substraction Commands do  $Fd=Fa-Fb$ , the Division Commands do  $Fd=Fa/Fb$ . No remainder is calculated.

fres Fd,Fa

This command calculates a single precision estimate of the reciprocal of the floating point operand in Fa. The estimate is correct to a precision of one part in 256 of the reciprocal of Fa. According to Sam Jordan this command produces an exact result, not an estimation, on the PPC 604e.

frsqrts Fd,Fa

This command calculates a double precision estimate of the reciprocal of the square root of the floating point operand in Fa. The estimate is correct to a precision of one part in 32 of the reciprocal of the square root of Fa. This command usually needs only 1 (One !!!) CPU Cycle.

fsel Fd,Fa,Fc,Fb

Fa is compared to 0. If  $Fa \geq 0$ , Fd is set to Fc. If it is  $< 0$  or not a number, Fd is set to Fb. The comparision ignores the sign of 0. This commands is the same like in C:  $Fd=Fa?Fc:Fb$ ; Combined with an fsub, fsel can for example be used for simple if-then-else, or to calculate the Minimum/Maximum.

Examples:

fsub f3,f2,f1

fsel f4,f3,f1,f2 ;  $f4=\min(f1,f2)$

fsub f3,f2,f1

fsel f4,f3,f2,f1 ;  $f4=\max(f1,f2)$

Note: These Examples give a wrong result, if one of f1,f2 is NaN (not a number).

With the exception of fsel all FPU Arithmetic Commands support the . Notation like described in **Important things** . They don't support the o Notation.

## 1.14 MagicSNs PowerPC ASM Coding Tutorial

FPU Compare Commands

`fcmpu crf,Fa,Fb`

`fcmpo crf,Fa,Fb`

These are the compare commands for FPU Numbers. The `crf` should always be specified.

`fcmpu` and `fcmpo` only are different in the handling of Not-A-Number's. I

recommend not using Not-a-Numbers with compares to keep this stuff easy.

Branch Commands are the usual `beq,ble,bgt,...`, with an additional `bun` (Branch, if unordered) and `bnu` (Branch, if not unordered), which uses Bit 3 of the `crf`.

A Compare is Unordered, if one of the Numbers was a Not-A-Number.

Principally you get an exception, when you do a `fcmpo`, and at least one of the numbers is not a number (at least if the not-a-number was created because of an invalid arithmetic operation). This does not happen with `fcmpu`, where you can check this with `bun` and `bnu` as described above. Note, that the exception won't happen, if you don't change the FPSCR Bits. WarpOS switches the FPU Exceptions off, as default.

## 1.15 MagicSNs PowerPC ASM Coding Tutorial

FPU StormPowerASM Macros

`If Fd,variable`

`ls Fd,variable`

`sf Fd,variable`

`ss Fd,variable`

These macros are used to access variables with the FPU. You should NOT access variables in any other way, because of the way Large and Small Data is handled.

Floating Point variables are declared with `dc.s` and `dc.d`. `ls` and `ss` are the

Load/Store Macros for Single Precision, `lf` and `sf` for Double Precision Numbers.

## 1.16 MagicSNs PowerPC ASM Coding Tutorial

Accessing Special Purpose Registers

Principally there are two sort of commands to access the special purpose registers of the PPC. `mtxxx` ("move to") moves data FROM a general purpose register TO a special purpose register, `mfxxx` ("move from") moves data FROM a special purpose register TO a general purpose register.

The Syntax is:

`mtXXX Rs`

---

mfXXX Rd

where d,s=0..31

xxx can mean (don't bother, if most of these don't mean anything to you, most of these registers should NOT be modified by a program that is NOT the OS itself). You definitely should NOT modify the BAT registers yourselves, for example. Use the Memory-Allocation functions of powerpc.library to access the BAT registers...

xer: The XER register

lr: The Link register

ctr : The Count register (One of the most useful commands here)

dsisr: The DSISR register

dar: The DAR register

dec: The Decrementer register

sdr1: The SDR1 register

srr0: Save and restore Register 0

srr1: Save and restore Register 1

asr: Address Space Register

ear: External Access Register

tbl: Time Base Lower

tbu: Time Base Upper

ibatu: IBAT Register, Upper

ibatl: IBAT Register, Lower

dbatu: DBAT Register, Upper

dbatl: DBAT Register, Lower

mtspr n,Rs

mfspr rD,n

Modify/Save SPRG0-SPRG3, n is the number of the SPRG to modify/to save.

mfpvr rD

Get the value of the processor version register (which tells you what PPC is inside your system). It is NOT possible to write TO the pvr.

It is also possible to use the "general form" mtspr/mfspr instead of these extended commands, but as the "general form" looks very abstract (mfspr Rd,8 means mflr Rd, for example), it is recommended to use the extended commands.

There are some not-extended commands also (asides from the general form):

mtrcf CRM,Rs

CRM is a mask specifying for each crn, if it will be modified by this command or not. If CRM(i) = 1, CR field i (CR bits 4\*i through 4\*i+3) will be modified. The command replaces the not-masked-out bits by the

low-order 32 Bits of Rs.

mcrxr crf

The contents of XER Bits 0-3 are copied into the Condition Field specified (for example cr2).

mfcrr Rd

The contents of the Condition Register are placed into the low-order 32 Bits of Rd. The Contents of the high-order 32 Bits of Rd are cleared in 64 Bit Implementations.

None of the Special Purpose Register Commands support the . or o Notation (which is described in [Important things](#) .

## 1.17 MagicSNs PowerPC ASM Coding Tutorial

Contextswitches and Mixed Binaries

[Calling PPC Shared Libraries from PPC Code](#)

[Calling 68k Code from PPC Code \(Contextswitch\)](#)

[Calling PPC Code from 68k Code \(Contextswitch\)](#)

[Asynchrone Contextswitches](#)

[Mixed/Fat Binary](#)

[Sushi](#)

Just to note it once more: You cannot use StormPowerASM together with ppc.library. You HAVE to use powerpc.library !!! StormPowerASM NEEDS the Amiga Executable Format, it CANNOT produce ELF Format Executables !!! (It needs the PowerOpen/EHF Executable format, to be exact...)

## 1.18 MagicSNs PowerPC ASM Coding Tutorial

Mixed/Fat Binary

Mixed/Fat Binary Creation is a bit more complicated. You CANNOT do it from the CLI Interface of the Assembler. You HAVE to use the GUI. I will describe how to create a Mixed/Fat Binary now in several steps:

1. Create a Project file of the Storm Environment.

The Settings you use (PPC/Mixed Binary/...) do not matter at all, as you will change them completely, anyways.

2. Remove all entries from the project, so that it is an empty project now.

3. In the Assembler-Settings do:

\* Append Header File: stormc:asm-includes/powerpc/ppcmacros.i

\* PowerPC: Remove "Codegenerator: Create executable Program" Checkmark  
(if not already Removed)

4. In the Linker-Settings do:

\* Linker1: Set to "Custom Startup-Code" Progdir:startups/asm\_startup.o

\* Link Program

5. Save these Settings somewhere. You will always use them,  
if you want to do a Mixed Binary WITHOUT C parts (with  
C Parts you don't need this, you only need this for a  
100% ASM Project).

6. Select "Choose Program Name" and set the Name of your  
Program

7. Include your 68k sources and your PPC sources (see below)

8. ASM the thing by pressing the button in the Storm Environment

As to how the sources have to look:

A. PPC-Part

=====

1. XREF the \_PowerPCBase
2. XDEF your PPC-Functions
3. XREF all used 68k Functions called by Context-Switches
4. Define your functions as usual in the code-section,  
don't forget to create the Stackframes

Example (out of the StormPowerASM Docs) :

```
incdir "stormc:asm-includes"
```

```
include powerpc/powerpc.i
```

```
XREF Func68K
```

```
XREF _PowerPCBase
```

```
XDEF PPCMain
```

```
section code
```

```
PPCMain
```

```
prolog
```

```
mr r4,r3
```

```
li r3,8
```

```
RUN68K Func68K
```

```
epilog
```

B. 68k-Part

=====

1. XREF all PPC-functions to be used
2. XREF \_LinkerDB, if you use Global Variables inside the PPC-Functions

In this case you should put it to a4 using

---

```
lea _LinkerDB,a4
```

before calls of RUNPOWERPC/RUNPOWERPC\_XL. \_LinkerDB points to the SmallDataBase.

3. Define all 68k-Functions the PPC-Source calls by using Context-Switches

4. Define \_PowerPCBase

5. Define \_main as the main function of your 68k Code.

A Mixed Binary always starts on 68k side !!!

6. Open powerpc.library like described in the section about Context-Switches (OPENPOWERPC,CLOSEPOWERPC,POWERDATA)

7. Do the interfacing between 68k and PPC Parts by using Context-Switches

The difference between Mixed and Fat Binary is mainly:

- Mixed Binaries do some things in 68k, some things in PPC

- Fat Binary have all PPC functions also available in

68k versions. If powerpc.library fails to open, they

run the 68k versions, else the PPC versions

BTW: PPC-Library-Functions get their Library-Bases in r3 after the call !!!

Example out of the StormPowerASM Docs:

68k Part (remember to name your sources differently !!!)

```
XREF PPCMain
```

```
XREF _LinkerDB
```

```
XDEF Func68K
```

```
XDEF _PowerPCBase
```

```
XDEF _main
```

```
incdir "stormc:Asm-includes"
```

```
include powerpc/powerpc.i
```

```
include lvos/exec_lib.i
```

```
include lvos/dos_lib.i
```

```
section "",code
```

```
_main
```

```
movem.l d1-a6,-(sp)
```

```
move.l $4.w,_SysBase
```

```
lea _LinkerDB,a4
```

```
OPENPOWERPC
```

```
tst.l _PowerPCBase
```

```
beq.b .exit
```

```
lea dos_name,a1
```

```
moveq #0,d0
```

```
CALLEXEC OpenLibrary
```

```
tst.l d0
```

```
beq,b .exit
move.l d0,_DOSBase
move.l #12,d0
RUNPOWERPC PPCMain
move.l d0,Args
move.l #result_text,d1
move.l #Args,d2
CALLDOS VPrintf
move.l _DOSBase,a1
CALLEXEC CloseLibrary
.exit
CLOSEPOWERPC
movem.l (sp)+,d1-a6
moveq #0,d0
rts
Func68K
movem.l d1-a6,-(sp)
muls d1,d0
movem.l (sp)+,d1-a6
rts
section "",data
POWERDATA
dos_name dc.b "dos.library",0
result_text dc.b "The PPCMain function returned %ld\n",0
section "",bss
_SysBase ds.l 1
_DOSBase ds.l 1
Args ds.l 1
PPC Part:
incdir "stormc:asm-includes"
include powerpc/powerpc.i
XREF Func68K
XREF _PowerPCBase
XDEF PPCMain
section code
PPCMain
prolog
mr r4,r3
li r3,8
RUN68K Func68K
epilog
```

---



## 1.19 MagicSNs PowerPC ASM Coding Tutorial

Asynchrone Contextswitches

You should first read about [Calling 68k Code from PPC Code](#) and about [Calling PPC Code from 68k Code](#) and you should

ALWAYS remember:

There is no way

to do PPC<->68k communication WITHOUT Contextswitches in a efficient way, even if some people assumed so on the comp.sys.amiga.misc discussions.

Often in these discussions "Messaging" like the announced Messaging-Concept of Phase 5 was described as being superior to Context-Switches. It should

be once more clarified, that Context-Switching is nothing more than a

"special case" of Messaging. It was also discussed, that running PPC and

68k parallel might help. In most cases this is not true. As soon as the

two use common data there are serious problems coming out of the hardware

design of the Boards, which will slow down the code below 68060 speed. And

even in the best cases a speedup of more than 20% is not possible, even

with a 060. If you really KNOW what you are doing, you COULD still try

running the two CPUs parallel, like described in

[Asynchrone Contextswitches](#) . Don't try it, when you don't

know EXACTLY what you are doing !!! To clarify it once more: This has nothing

to do with problems of certain kernel implementations, it is just a problem

of the existing hardware. It behaves like this on both implementations

for the current hardware. So only use the asynchrone version, if you

know that the advantages will be better than the disadvantages in your

specific coding problem.

A Contextswitch consumes about 0.5 milliseconds, so you should be VERY careful, when to use it.

Also note, that other tasks can continue while the context-switch waits

for the second CPU to complete the Cache-stuff EVEN WITH SYNCHRONOUS

CONTEXT-SWITCHES.

The principal way to do asynchrone Contextswitches is:

1. Call the Contextswitch
2. Now do other stuff (the Contextswitch is asynchrone, so you do not have to wait)
3. Wait for the Contextswitch to be finished later

There is one limitation: You are not allowed to do a synchrone contextswitch, before you waited for the LAST ASYNCHRONOUS contextswitch.

A Contextswitch is declared asynchrone by setting the ASYNC Bit in the

flag parameter of the Contextswitch Macro (first Bit, defined in powerpc.i).

The Wait-Functions are:

WAITFORPPC [FPU]

WAITFORPPC\_XL [FPU]

Their only (optional) parameter is FPU, in case the FPU-registers should be converted, when the Context-Switch returns. The not-XL-Version only returns d0 (fp0/fp1 with FPU-option) and d1/a0/a1 are trashed for the not-XL-version.

WAITFOR68K [FPU]

WAITFOR68K\_XL [FPU]

Their only (optional) parameter is FPU, in case the FPU-registers should be converted, when the Context-Switch returns. The XL-Version trashes r7-r10, if the RUN68K/RUN68K\_XL called a library function, then r31 contains the library base. The Not-XL-Version trashes r4-r10, and only d0 (fp0/fp1, if FPU was specified) are converted.

RunPPC, WaitForPPC, Run68K and WaitFor68K (and their \_XL-versions) also support error messages. They return:

PPERR\_SUCCESS = 0 ;success

PPERR\_ASYNCERR = 1 ;synchron call after asynchron call

PPERR\_WAITERR = 2 ;WaitFor[PPC/68K] after synchron call

And ALWAYS remember: NEVER USE ASYNCHRONE CONTEXTSWITCHES unless you know exactly what you are doing. You will have to be VERY careful with the caches, and even if you do everything 100% correct, it is still very probably, that your result will be slower, than if you did not use asynchrone contextswitches at all !!!

## 1.20 MagicSNs PowerPC ASM Coding Tutorial

Calling 68k Code from PPC Code (Contextswitch)

A Contextswitch can call 68k Code out of a PPC program, for example an AmigaDOS function.

ALL AMIGADOS FUNCTIONS CALLED BY THE PPC WILL CAUSE A CONTEXTSWITCH.

There is no way

to do PPC<->68k communication WITHOUT Contextswitches in a efficient way,

even if some people assumed so on the comp.sys.amiga.misc discussions.

Often in these discussions "Messaging" like the announced Messaging-Concept

of Phase 5 was described as being superior to Context-Switches. It should

be once more clarified, that Context-Switching is nothing more than a

"special case" of Messaging. It was also discussed, that running PPC and

68k parallel might help. In most cases this is not true. As soon as the

two use common data there are serious problems coming out of the hardware design of the Boards, which will slow down the code below 68060 speed. And even in the best cases a speedup of more than 20% is not possible, even with a 060. If you really KNOW what you are doing, you COULD still try running the two CPUs parallel, like described in

**Asynchrone Contextswitches** . Don't try it, when you don't know EXACTLY what you are doing !!! To clarify it once more: This has nothing to do with problems of certain kernel implementations, it is just a problem of the existing hardware. It behaves like this on both implementations for the current hardware. So only use the asynchrone version, if you know that the advantages will be better than the disadvantages in your specific coding problem.

A Contextswitch consumes about 0.5 milliseconds, so you should be VERY careful, when to use it.

Also note, that other tasks can continue while the context-switch waits for the second CPU to complete the Cache-stuff EVEN WITH SYNCHRONOUS CONTEXT-SWITCHES.

When you use the contextswitch, the registers are mapped the following way:

```
d0 <-> r3 fp0 <-> f1
d1 <-> r4 fp1 <-> f2
d2 <-> r22 fp2 <-> f3
d3 <-> r23 fp3 <-> f4
d4 <-> r24 fp4 <-> f5
d5 <-> r25 fp5 <-> f6
d6 <-> r26 fp6 <-> f7
d7 <-> r27 fp7 <-> f8
a0 <-> r5
a1 <-> r6
a2 <-> r28
a3 <-> r29
a4 <-> r2
a5 <-> r30
a6 <-> r31
```

Now there are some macros inside Stormc:ASM-Include/powerpc/powerpc.i (or StormC:ASM-Includes/powerpc/powerpc.i, i recommend copying the two Include-Paths together to one directory) to do a Contextswitch.

RUN68k

This macro takes two parameters. The first one is the Library base (in case of a Shared Library function), the second one the library function offset

(in case of a Shared Library function). The parameters have to be put to the correct PowerPC registers, like needed for the register mapping described above.

You have to XREF `_PowerPCBase`. Of other Bases, `_DOSBase` and `_SysBase` can also be XREF'ed, other bases have to be opened manually.

Note, that RUN68k can only use `d0/d1/a0/a1/fp0/fp1` for parameters. If you need other registers also, you have to use `RUN68K_XL`, which produces a bit slower and bigger code. `RUN68K_XL` supports ALL functions.

If you do not call a library function but a "normal" function, the function name takes the place of the Library Base in this macro (`RUN68K test`, for example).

`RUN68K_XL` trashes registers `r7-r10`, `RUN68K` trashes `r4-r10`.

Both Macros have some extra parameters.

`RUN68K`:

3rd parameter: Flags (defined in `powerpc.i`, normally you won't need them).

4th parameters: FPU (if you specify "FPU" as 4th parameter, the FPU registers will also be available to the 68k funcion).

`RUN68K_XL`:

3rd parameter: Flags (defined in `powerpc.i`, normally you won't need them).

4th parameter: Stacksize that should be provided to the context-switch. Normally you should not provide stack-space during a context-switch.

Using registers is much faster :).

5th parameters: FPU (if you specify "FPU" as 5th parameter, the FPU registers will also be available to the 68k funcion).

You can leave out parameters like that: `RUN68k test,,FPU`

Also, `a4<->r2` always exchange the `SmallDataBase` (or `_LinkerDB`), so that the PPC can access the Variables (`LinkerDB` can be XREF'ed)

You should be VERY careful with allocated memory, that should be used on the PPC. Best only use memory allocated with the functions of `powerpc.library` (this memory will be correctly aligned). Be especially CAREFUL, when using functions in a context-switch that allocate memory.

Example:

```
prolog
```

```
...
```

```
la r6,intname ;libname->a1
```

```
li r3,0 ;libversion->d0
```

```
RUN68K _SysBase.OpenLibrary ;OpenLibrary
```

```
sw r3,_IntuitionBase ;save _IntuitionBase
```

```
tstw r3 ;library successfully opened?
```

```
beq .exit ;no -> exit
```

```

li r5,0 ;a0 = NULL
RUN68K _IntuitionBase,DisplayBeep ;call DisplayBeep of intuition.library
...
epilog
intbase: dc.l 0
intname: dc.b 'intuition.library',0
even

```

## 1.21 MagicSNs PowerPC ASM Coding Tutorial

Calling PPC Code from 68k Code (Contextswitch)

A Contextswitch can call PPC Code out of a 68k program. This causes a Contextswitch (a Contextswitch is a special messaging-function which looks that there are no Cache-Problems between the two CPUs).

There is no way

to do PPC<->68k communication WITHOUT Contextswitches in a efficient way, even if some people assumed so on the comp.sys.amiga.misc discussions.

Often in these discussions "Messaging" like the announced Messaging-Concept of Phase 5 was described as being superior to Context-Switches. It should be once more clarified, that Context-Switching is nothing more than a "special case" of Messaging. It was also discussed, that running PPC and 68k parallel might help. In most cases this is not true. As soon as the two use common data there are serious problems coming out of the hardware design of the Boards, which will slow down the code below 68060 speed. And even in the best cases a speedup of more than 20% is not possible, even with a 060. If you really KNOW what you are doing, you COULD still try running the two CPUs parallel, like described in

[Asynchrone Contextswitches](#) . Don't try it, when you don't

know EXACTLY what you are doing !!! To clarify it once more: This has nothing to do with problems of certain kernel implementations, it is just a problem of the existing hardware. It behaves like this on both implementations for the current hardware. So only use the asynchrone version, if you know that the advantages will be better than the disadvantages in your specific coding problem.

A Contextswitch consumes about 0.5 milliseconds, so you should be VERY careful, when to use it.

Also note, that other tasks can continue while the context-switch waits for the second CPU to complete the Cache-stuff EVEN WITH SYNCHRONOUS CONTEXT-SWITCHES.

When you use the contextswitch, the registers are mapped the following way:

```
d0 <-> r3 fp0 <-> f1
d1 <-> r4 fp1 <-> f2
d2 <-> r22 fp2 <-> f3
d3 <-> r23 fp3 <-> f4
d4 <-> r24 fp4 <-> f5
d5 <-> r25 fp5 <-> f6
d6 <-> r26 fp6 <-> f7
d7 <-> r27 fp7 <-> f8
a0 <-> r5
a1 <-> r6
a2 <-> r28
a3 <-> r29
a4 <-> r2
a5 <-> r30
a6 <-> r31
```

To call PPC functions from 68k, you have at first to open the powerpc.library and initialize WarpOS. You should do this at the beginning of your code like that:

- include stormc:asm-includes/powerpc/powerpc.i (might also be at stormc:asm-include/powerpc/powerpc.i, i recommend copying the contents of both directories together)

- Define \_SysBase as \$4 (the Macros need this). In case you link everything together with StormPowerASM, you don't need to define \_SysBase, but simply can XREF it.

- put the Macro POWERDATA into the datasection

- use the Macro OPENPOWERPC (opens powerpc.library) at the start of your code

- use the Macro CLOSEPOWRPC (closes powerpc.library) at the end of your code

This has not to be done, if you call your PPC Code from a PPC- or Mixed/Fat-Binary C Program. StormC already initializes the powerpc.library automatically !!!

Example:

```
Include "stormc:asm-includes"
```

```
Include "powerpc/powerpc.i"
```

```
_SysBase EQU $4
```

```
start:
```

```
movem.l d1-a6,-(sp)
```

```
OPENPOWERPC
```

```
...
```

```
CLOSEPOWERPC
```

```
movem.l (sp)+,d1-a6
```

```
moveq #0,d0
```

```
rts
```

```
section data
```

```
POWERDATA
```

If you need a specific version of powerpc.library (for example at least Version 12), you can also do

```
OPENPOWERPC 12
```

If the Variable (defined by the Macros) `_PowerPCBase` contains 0 after `OPENPOWERPC`, then `powerpc.library` could not be opened, or not in the correct version number.

As to the call of the PPC function from the 68k Source, you have to do:

- export the name of the PPC-function using XDEF in the PPC-Source
- import the name of the PPC-function using XREF into the 68k-Source
- Start the PPC-Function using the Macro `RUNPOWERPC` in the 68k Source

Example:

```
test.p:
```

```
XDEF PPCTest
```

```
PPCTest:
```

```
prolog
```

```
epilog
```

```
test.asm:
```

```
XREF PPCTest
```

```
Include "stormc:asm-includes"
```

```
Include "powerpc/powerpc.i"
```

```
_SysBase EQU $4
```

```
start:
```

```
movem.l d1-a6,-(sp)
```

```
OPENPOWERPC
```

```
RUNPOWERPC PPCTest
```

```
CLOSEPOWERPC
```

```
movem.l (sp)+,d1-a6
```

```
moveq #0,d0
```

```
rts
```

```
section data
```

```
POWERDATA
```

To tell the truth, there are two `RUNPOWERPC`-Macros, `RUNPOWERPC` and `RUNPOWERPC_XL`.

The difference between the two is, that for `RUNPOWERPC` only the registers `d0/d1/a0/a1/fp0/fp1` will be translated, for `RUNPOWERPC_XL` (which is a bit slower and produces a bit bigger source) all registers will be translated.

Both functions support some parameters:

RUNPOWERPC:

- 1.Parameter: Functionname
- 2.Parameter: Flags (defined in powerpc.i)
- 3.Parameter: "FPU" (if FPU is specified, the FPU-registers also will be translated)

RUNPOWERPC\_XL:

- 1.Parameter: Functionname
- 2.Parameter: Flags (like above)
- 3.Parameter: Stacksize (using this you can translate stackareas, which is discouraged though, as it will cause quite a slowdown)
- 4.Parameter: "FPU" (like above)

Also, a4<->r2 always exchange the SmallDataBase (or \_LinkerDB), so that the PPC can access the Variables (LinkerDB can be XREF'ed)

Example:

```
RUNPOWERPC_XL PPCTest,,1024
```

## 1.22 MagicSNs PowerPC ASM Coding Tutorial

Sushi

Sushi is a nice utility for debugging purposes. You can download it from Aminet. To use it, do:

```
setdb 3
```

```
sushi
```

Note, that setdb 3 opens the powerpc.library (setdb comes with your version of StormPowerASM and sets the debug level). Also note, that you can of course put the sushi output to a file, for example:

```
setdb 3
```

```
sushi >work:sushi-log
```

You quit sushi/close the file by pressing CTRL-C.

Sushi logs for you all messages that are exchanged by Context-Switches and such stuff. Usefull :)

If Enforcer is running, Sushi provides extra information.

Note, though, that running Sushi and debug level 3 will slow down things, certainly.

## 1.23 MagicSNs PowerPC ASM Coding Tutorial

Calling PPC Shared Libraries from PPC Code

Of course you do not need a contextswitch for this. It is completely PPC Native to call a PPC Shared Library function from PPC Code. Currently,



there are not many PPC Shared Libraries available. There is powerpc.library, of course, which contains some important functions like Memory Allocation functions for PPC Native programs. I recommend STRONGLY that you read the Autodocs of the powerpc.library, when your program needs some exec- or utility-style functionality (Taglists, lists and messageports are for example available PPC Native, also memory-protection on PPC-Side).

Then there is rtgmaster.library PPC version. Well, it is not yet released to the public, but it soon will be. Asides from these two libs, there is no PPC Shared Library yet released, as far as i know. But hopefully soon there will be more. Maybe even one day the OS functions of the Amiga !!!

To call a PPC function you have to do the following:

- XREF the symbol \_PowerPCBase
- create a valid stackframe, like described in [Coding Subroutines](#)
- include powerpc\_lib.i (which will also include ppcmacros.i)
- use CALLPOWERPC of powerpc\_lib.i to call the function.

Note: At least on my version of StormPowerASM, which apparently was quite an early version (I got it on the Computer '97 directly from H&P) there is a bit a chaos as to the includes on the CD. Some ASM-includes are in stormc:asm-include and some in stormc:asm-includes. I simply copied those in stormc:asm-include to stormc:asm-includes.

Example (from the StormPowerASM Docs, after some include path fixes)

```
incdir "stormc:asm-includes"
include exec/memory.i
include "stormc:asm-includes/LVOs/powerpc_lib.i"
XREF _PowerPCBase
executable
version 7
start
prolog ;build stackframe
li r4,4096 ;4096 bytes to allocated
liw r5,MEMF_PUBLIC!MEMF_CLEAR ;memory flags
li r6,0 ;no alignment restrictions
CALLPOWERPC AllocVecPPC ;allocated memory
tstw r3 ;test result
beq .exit ;if 0 then error
mr r4,r3 ;move to r4
CALLPOWERPC FreeVecPPC ;free allocated memory
.exit
epilog ;remove stackframe
```

Note: The command "version 7" simply ensures, that at least powerpc.library V7 will be tried to be opened. As you see, you do not have to open powerpc.library manually, the system will do this for you. Other PPC Shared Libraries will be opened manually, of course. As OpenLibrary() is a 68k function, you would do this in the 68k Part of your code, of course (For example, if you want to use rtgmaster.library PPC version...).

## 1.24 MagicSNs PowerPC ASM Coding Tutorial

Additional StormPowerASM Macros

Please note, that you have to include stormc:include/powerpc/ppcmacros.i to use these macros. These macros are only defined for 32 Bit operations.

bitchg Rn,const

bittst Rn,const

bitclr Rn,const

bitset Rn,const

These are the Bit-commands, like known from the 68k. You have to be careful, though. The highest bit on the \*68k\* is 31, on the \*PPC\* this is \*0\* !!! bittst stores its results in cr0.

clrb Rn

clrh Rn

clrw Rn

Clears the Low-Byte/Halfword/Word of a register, like clr on the 68k. These commands support the . notation, like defined in **Important things** .

mb Rd,Rs

mh Rd,Rs

These commands copy the Low-Byte/Low-Halfword from Rs to Rd. The other Bytes of Rd stay unchanged. These commands support the . notation.

setb Rn

seth Rn

setw Rn

These commands move the value -1 to register Rn (by setting all bits of the lowest Byte/Halfword/Word to 1).

tstb Rn

tsth Rn

tstw Rn

These commands compare Rn with 0, like on the 68k. cr0 is set accordingly.

There are also some useful macros to access registers in ppcmmacros.i:

local = r13

```
base = r2
stack = r1
trash = r0
_d0 equr r3
_d1 equr r4
_d2 equr r22
_d3 equr r23
_d4 equr r24
_d5 equr r25
_d6 equr r26
_d7 equr r27
_a0 equr r5
_a1 equr r6
_a2 equr r28
_a3 equr r29
_a4 equr base
_a5 equr r30
_a6 equr r31
_fp0 fequr f1
_fp1 fequr f2
_fp2 fequr f3
_fp3 fequr f4
_fp4 fequr f5
_fp5 fequr f6
_fp6 fequr f7
_fp7 fequr f8
```

This is especially useful, when you do context-switches and don't want to have a look at the register table all the time.

## 1.25 MagicSNs PowerPC ASM Coding Tutorial

History

1.0 First Version

1.1 Added Branch+Compare Commands, Logic Commands, Extension Commands, Multiple Load/Store Operations and Branch Prediction Optimizations.  
Added arithmetic commands (still incomplete, Mul/Div still missing).

Some Bugfixes.

1.2 Bugfixed Arithmetic Commands, added Mul/Div commands, Some Bugfixes.

1.3 Added Stuff about Rotating/Shifting (Rotate\_32 Commands might contain

bugs, did not understand that completely myself, yet), added information about relative/absolute branches, added condition-logic commands.

1.4 Added info about PowerOpen Standard and function-coding, added info about a lot of StormPowerASM macros, clarified use of the macro executable in "A First Program"

1.41 Fixed Small bug in stmw description and bug concerning 64 Bit operations (Read **Important Things** ).

1.42 Fixed small bug in description of Extension Commands concerning 32/64 Bit Implementations

1.5 Added information about calling functions from PPC Shared Libraries  
Added information concerning sushi

Added information concerning Contextswitches (Synchronous and Asynchronous)

Added information about Mixed/Fat Binary

1.6 Added extended commands for condition bit commands

Added Special Register Access Commands

Added Extended Commands for Shift/Rotate

Added Multi Precision Shifts

The Integer-Part is now complete, at least as to the functions that are interesting for programmers who do not want to code a new OS in PPC ASM.

I also left out everything interrupt-concerning.

I will do the FPU-Part next.

1.7 Started FPU Part

Did not yet handle Normalization in Descriptions

2.0 With the Exception of the Optimizing Chapter the tutorial is done now.

As to FPU Exceptions and Normalization i chose only to provide basic info which is interesting for the common programmer. In the - very improbable - case you need more info about these themes, read in the Motorola Docs about it. I did not want to write this whole document about stuff the usual programmer does not need, anyways. This tutorial should help programmers to find an easy start... it should not replace the Motorola Docs as the "complete docs to PPC where everything is found".

2.1 Fixed a lot of Bugs in the Documentation, and added some more info, thanks to Sam Jordan for reading the whole stuff and bug-reporting... :)

2.2 Fixed Bugs concerning Carry/Borrow and another silly bug in the 68k->PPC Porting section, used the highly Optimized Shift with Upper Word unchanged combination from Sam Jordan (the one i hacked in fast before the release of the last version had a bug).

2.3 Well, Sam found out, that the way he explained the Carry/Borrow Business to me was not completely correct. And i never used a subtract with Carry Command before in a PPC Source :) Well, now this document is really fixed and also describes the Subtract-With-Carry Stuff like it really works :)

---

## 1.26 MagicSNs PowerPC ASM Coding Tutorial

### Multiple Load/Store Operations

Note: According to Sam Jordan you should avoid these functions "as they can cause problems". Use the POP/PUSH Macros to save registers...

`lmw Rt,d16(Ra)`

This Command loads 32-Rt Words into SEVERAL registers. It only loads into the Low-Word of these registers, the high-word is set to 0.

To use this command, the EA (Ra+16 Bit Offset) has to be divisible by 4.

If Ra is set to r0, EA is set to 0+16 Bit Offset. The command starts the loading with EA and Rt. Ra is not allowed to be one of the loaded registers, and Rt and Ra are not allowed to be both 0.

Example:

`lzw r5,27`

`lmw r5,10(r17)`

This loads 5 Words into the registers r5-r9, started from 10(r17).

`stmw Rs,d16(Ra)`

This command stores 32-Rt Words from SEVERAL registers to the memory.

It only stores the Low-Words from these registers. To use this command, the EA (Ra+16 Bit Offset) has to be divisible by 4. If Ra is set to r0, EA is set to 0+16 Bit Offset. The command starts the storing with EA and Rt.

Example:

`lzw r5,27`

`stmw r5,10(r17)`

This stores 5 Words from the registers r5-r9 into 10(r17).

`lswi Rt,Ra,NB`

Rt to Rt+nr-1 with nr=CEIL(n:4) and n = NB (32, if NB = 0) will be loaded with Bytes into the Low-Order, the High-Order will be set to 0.

Loading will be done from Left to Right.

This will be done with Register Wrap-Around (if Rt+nr-1>32). If there is not enough data to fill the last register, the Bytes of the last register which are not filled, will be filled with 0. The EA has NOT to be divisible by 4. Ra can be set to r0 to use 0 instead of r0. This function is best optimized, if Rt=Ra=5 and Rt+nr-1<=12.

`lswx Rt,Ra,Rb`

This does the same like lswi, but n=XER. If n = 0 => Rt undefined. Ra is not allowed to be the same like Rb, and Rt and Ra are not allowed to be both r0 (which means 0, like always).

stswi Rt,Ra,NB

EA (which does not have to be divisibly by 4) is set to Ra or 0 (if Ra=r0).

The low-words of Registers Rt to Rt+nr-1 (with possible register-wraparound) are filled with data from the EA, where nr=CEIL(n:4) and n = NB (32, if NB = 0).

If there is not enough data to fill the last register, the Bytes of the last register which are not filled, will be filled with 0. This function is best optimized, if Rt=Ra=5 and Rt+nr-1<=12.

stswx Rt,Ra,Rb

This works the same like stswi, but n=XER. If n = 0 => Rt undefined. Ra is not allowed to be the same like Rb, and Rt and Ra are not allowed to be both r0 (which means 0, like always).

## 1.27 MagicSNs PowerPC ASM Coding Tutorial

Integer Arithmetic Commands

There are some sorts of Integer Arithmetic Commands on the PowerPC:

**Normal Additions/Subtractions**

**Add/Sub with Carry**

**Add/Sub using Carry**

**Multiplications/Divisions**

## 1.28 MagicSNs PowerPC ASM Coding Tutorial

Normal Additions/Subtractions

addi Rt,Ra,SI

addis Rt,Ra,SI

add Rt,Ra,Rb

subi Rx,Ry,val

subis Rx,Ry,val

subf Rt,Ra,Rb

sub Rt,Ra,Rb

These commands do addition and subtraction. In fact the subtractions are only extended commands defined by adding a negative value. the \*i commands are used to add/subtract (signed) integers, the \*is for (signed) integers, but the Constant is shifted 16 Bit Left before it is used - the constant itself can only be 16 Bit, because of this the \*i and \*is versions are supported.

The plain add/sub is for adding/subtracting register values. These commands use 16 Bit Integers as SI. The . and o Notation, like outlined in

**Important things** is valid with add, subf and sub, but not

for addi/addis/subi/subis. subf does  $Rt=Rb-Ra$ , sub does  $Rt=Ra-Rb$ .

neg Rt,Ra

This negates Ra and saves the result into Rt. (2-Complement). The . and o notations are both supported.

If Ra is the lowest possible number, 0x8000 0000 0000 0000 (or 0x8000 0000 in 32 Bit Implementations), Rt is set to this number, and if the o notation was used, SO is set to 1.

There are some extended commands, defined by the above commands:

li Rx,val = addi Rx,0,val

(read the comments about this command in the section about the Load/Store commands, where i put it, because i found it more fitting to be put there).

la Rx,Disp(Ry) = addi Rx,Ry,Disp

(Read the comments about this command also in the section about the Load/Store commands, where i put it, because i found it more fitting to be put there).

## 1.29 MagicSNs PowerPC ASM Coding Tutorial

Add/Sub with Carry

Note: For Add's the Carry is used as Carry-Bit. For Subs, the mechanism is not the same like for 68k (Borrow-Bit). The mechanism with Borrow-Handling was not described correctly in earlier versions of this document, so be careful !!!

What really happens, is, that, if a Subtract Command would set the Borrow Bit on the \*68k\*, the \*PPC\* sets the Carry Bit to the \*inverted Borrow Bit\*. This is done in this way, as the PPC implements a subtraction as a negative addition, while the 68k has real subtraction commands.

Example for using the Carry/Borrow with 64 Bit Numbers in two registers:

subfc r5,r4,r5

subfe r7,r6,r7 ;64-Bit-Number (r7:r5) - 64-Bit-Number (r6:r4)

Now as to the command list for Add/Sub with Carry:

addic Rt,Ra,SI

$Rt=Ra+SI$ , Carry will be set correctly, SI is a unsigned 16 Bit Integer.

subic Rt,Ra,SI

$Rt=Ra-SI$ , Carry will be set correctly, SI is a unsigned 16 Bit Integer.

subfic Rt,Ra,SI

$Rt=SI-Ra$ , Carry will be set correctly, SI is a signed 16 Bit Integer.

addic and subic support the . Notation, like outlined in **Important things** ,

but not the o notation. subfic supports none of the two.

addc Rt,Ra,Rb

subfc Rt,Ra,Rb

subc Rt,Ra,Rb

These commands work similar to the above three commands, only that the third parameter is not an Integer, but a register also. They support both the . and the o Notation.

## 1.30 MagicSNs PowerPC ASM Coding Tutorial

Add/Sub using Carry

Note: For Add's the Carry is used as Carry-Bit. For Subs, the mechanism is not the same like for 68k (Borrow-Bit). The mechanism with Borrow-Handling was not described correctly in earlier versions of this document, so be careful !!!

What really happens, is, that, if a Subtract Command would set the Borrow Bit on the \*68k\*, the \*PPC\* sets the Carry Bit to the \*inverted Borrow Bit\*. This is done in this way, as the PPC implements a subtraction as a negative addition, while the 68k has real subtraction commands.

Example for using the Carry/Borrow with 64 Bit Numbers in two registers:

```
subfc r5,r4,r5
```

```
subfe r7,r6,r7 ;64-Bit-Number (r7:r5) - 64-Bit-Number (r6:r4)
```

Now as to the command list for Add/Sub using Carry:

```
adde Rt,Ra,Rb
```

```
subfe Rt,Ra,Rb
```

```
addme Rt,Ra
```

```
subfme Rt,Ra
```

```
addze Rt,Ra
```

```
subfze Rt,Ra
```

These commands add the Carry-Flag to the result of the operation. adde is a simple addition of Ra and Rb (and Carry), subfe is Ra-Rb (+ Carry), addme is the -1+Ra+Carry, subfme is -1-Ra+Carry, addze is Ra+Carry, subfze is -Ra+Carry. All these commands support the . and o notations, like outlined in [Important things](#) .

## 1.31 MagicSNs PowerPC ASM Coding Tutorial

Multiplications/Divisions

[Multiplications](#)

[Divisions](#)

## 1.32 MagicSNs PowerPC ASM Coding Tutorial

Multiplications

```
mulli Rt,Ra,SI
```

```
mullw Rt,Ra,Rb
```

```
mulld Rt,Ra,Rb
```

```
mullhd Rt,Ra,Rb
```



mullhdu Rt,Ra,Rb

mullhw Rt,Ra,Rb

mullhwu Rt,Ra,Rb

These are the multiplication commands.

mulli:

32 Bit Implementations:

32\*32 Bit => Low 32 Bit of the result.

64 Bit Implementations:

64\*64 Bit => Low 64 Bit of the result. A full 128-Bit product can be calculated with mulli and one of mulhd/mulhdu/mulhw/mulhwu.

mullw:

32\*32 Bit => 64 Bit. The low-order parts of the parameters are used, only.

Together with mulhw/mulhwu a complete 64-Bit product can be calculated.

The low-order 32 Bits are the correct 32-Bit-Product for 32 Bit Implementations and in 32 Bit Mode also for 64 Bit implementations.

muld:

64\*64 Bit => Low 64 Bit of the result. Only exists on 64 Bit implementations.

mullhw:

32\*32 Bit => High 32 Bit of the result. The parameters are taken as signed integer, here. The high 64-Bit of the result are placed into the LOW Word.

The HIGH word is undefined. This command can be used together with mulli/mullw to calculate a full 128/64 Bit product.

mullhd:

64\*64 Bit => High 64 Bits of the result. Asides from being 64 Bit, it works the same like mullhw. This command only exists on 64 Bit Implementations.

mulhwu:

Unsigned Version of mullhw.

mulhdu:

Unsigned Version of mullhd. Only exists on 64 Bit Implementations.

mullw and muld supports both the . and o notation, like defined in

**Important things** , mulli none of them, all other Multiplication commands only the . notation.

### 1.33 MagicSNs PowerPC ASM Coding Tutorial

Divisions

divw Rt,Ra,Rb

divd Rt,Ra,Rb

divwu Rt,Ra,Rb

divdu Rt,Ra,Rb

divw: Divides signed parameters Ra:Rb and puts the result into Rt. The Dividend is the 64 Bit Sign-Extended version of the 32 Bit Low-word of Ra, the divisor simply is the 32 Bit Low-word of Rb. The result is put into the Low-Word of Rt, the High-Word is undefined. No remainder will be calculated.

divd: Divides signed parameters Ra:Rb and puts the result into Rt.

The Dividend is the 64 Bit Doubleword Ra, the divisor the 64 Bit Doubleword Rb. The result is also 64 Bit. No remainder will be calculated.

Only exists on 64 Bit implementations.

divwu+divdu: Unsigned versions of divw and divd. divdu only exists on 64 Bit implementations.

If you need the remainder, you have to calculate it yourselves.

The division commands support both the . and the o Notation, like specified in **Important things** .

Note: Divisions by  $2^n$  can be quickly done, by combining a Shift Right Algebraic and a addze instruction (adding the Carry-Flag to the Shifted Result). The Shift-Command already sets the Carry correctly.

## 1.34 MagicSNs PowerPC ASM Coding Tutorial

Logic Commands

andi Ra,Rs,UI

andis Ra,Rs,UI

xori Ra,Rs,UI

xoris Ra,Rs,UI

ori Ra,Rs,UI

oris Ra,Rs,UI

These commands perform logic operations with Integer-Variables.

The difference between the \*i and \*is versions is, that with the \*is the constant will be Shifted 16 Bit Left before the operation (This solution is taken, as UI only can be 16 Bit, and it might be needed to access all 32 Bit). The target is register Ra. andi and andis

also support the . notation, like outlined in **Important Things** .

UI is a 16 Bit constant.

and Ra,Rs,Rb

xor Ra,Rs,Rb

or Ra,Rs,Rb

nand Ra,Rs,Rb

nor Ra,Rs,Rb

eqv Ra,Rs,Rb

andc Ra,Rs,Rb

orc Ra,Rs,Rb

These are command to perform logic operations between Registers.

The target is register Ra. Eqv is the Complement of Xor, andc is the Complement of and, orc is the Complement of or. All these commands support the . notation.

mr Rx,Ry

This is an extended command, which is defined as or Rx,Ry,Ry and which copies Register Ry to Register Rx.

not Rx,Ry

This is an extended command, which is defined as nor Rx,Ry,Ry, and which negates Ry (to Rx).

nop

This is an extended command, which is defined as ori 0,0,0

Examples:

andis. r4,r5,r6

mr r8,r11

## 1.35 MagicSNs PowerPC ASM Coding Tutorial

A Quick Start

As you all know, the Amiga was at it's end. There were no new fast Amigas. But now Phase 5 built that great PowerUP Accelerators, 5-10 times faster than a 060.

The Problem: 68k ASM source won't run on them (at least not on the PPC part of the Dual-Processor-System... on the 68k Part of course, it will run, but it won't be faster than on any 68k Accelerator there). They are PowerPCs.

For application programmers this is no problem. They code in C, anyways.

But i decided to write this tutorial to help the game/demo coders to adapt to the new platform. And i will tell you: When you coded a while for the PPC in ASM, you won't miss the 68k !!!

About this document: I am assuming you are acknowledged with 68k ASM.

So i won't explain what dc.l is doing and such stuff... Thanks to Sam

Jordan, the Pseudo-Op-codes of the PPC ASM look very much like those of the 68k ASM. I will concentrate on the PPC commands themselves.

I also assume you have WarpUP and StormPowerASM installed. If you do NOT have it:

- WarpUP is available at: <ftp://ftp.haage-partner.com/WarpUP/WarpUP.lha>
- There is also an update on this site, which gives you the latest version
- StormPowerASM is a commercial PPC ASM by Haage&Partner. In fact the only PPC ASM on Amiga currently. And it is great !!!

Just to make it once more clear. The coding conventions used here are for WarpUP, not for ppc.library. Also StormPowerASM only runs together with WarpUP. WarpUP is the better PPC Software, in my opinion, anyways. This document is also mainly intended for 68k ASM coders, who want to port some of their code to PPC or optimize it for PPC. As will be more detailed described later, you don't need to port more than 10-20% of your code to PPC ASM in most cases...

## 1.36 MagicSNs PowerPC ASM Coding Tutorial

Extension Commands

extsb Ra,Rs

extsh Ra,Rs

extsw Ra,Rs

These commands extend Rs (filling the 0-Bits with the highest Bit).

These commands extend to 64 Bit, anyways, what Bit-Size it originally was. These commands support the . notation, like outlined in

**Important things** . For 32 Bit Implementations the register is extended to 32 Bit, of course. extsw only exists on 64 Bit Implementations.

cntlzw Ra,Rs

cntlzd Ra,Rs

These commands count the leading zeros in Rs. The cntlzd only exists on 64 Bit Implementations (not on 603e, for example), the cntlzw also exists on 32 Bit Implementations. These commands support the . notation also.

## 1.37 MagicSNs PowerPC ASM Coding Tutorial

A First Program

Create the following file named first.p :

```
include "stormc:include/powerpc/ppcmacros.i"
executable
start:
prolog
liw r3,$1000000
```

```
mtctr r3
.loop
bdnz .loop
epilog
```

You see, PPC programs look like that:

1. The Pseudo-Opcode executable
2. The Pseudo-Opcode prolog
3. The Main Program
4. The Pseudo-Opcode epilog

prolog builds a "Stack Frame", epilog removes it again (The PPC always needs a correctly initialized Stackpointer !!!) If your programs use Subfunctions, then you even might need multiple "prolog" and "epilog" macros. I will come to this later. ppcmacros.i defines (among others) prolog and epilog.

To ASM it, you type (in the CLI) : powerasm first.p to first

Then start it. Well, it does not do anything really great.

When you understand how you have to build a PPC ASM program, I suggest you read about the commands that the PPC supports and then code something :)

You always have to include the macro "executable", if you want, that your program will be ASM'ed as an executable.

## 1.38 MagicSNs PowerPC ASM Coding Tutorial

### Important Things

The PPC has the following registers (Note: On 32 Bit PPCs like the 603e or the 604e r0-r31 are only 32 Bit, and 64 Bit commands are not available !!!) I listed only the Usermode registers. If you need the Supervisor mode registers, have a look at Motorola Documentation. But with the given Coding Systems on PowerUP, you normally don't have to handle the Supervisor mode registers yourselves.

f0-f31 : 64 Bit Wide FPU-Registers

cr : Condition Register

fpSCR : Floating Point Condition and Control Register

xer : 32 Bit Wide Another Condition Register

lr : 32 Bit Wide Link Register

ctr : 32 Bit Wide Counter Register

r0-r31 : 64 Bit Wide General Purpose Registers (32 Bit on current PowerUP)

tbl : Time Base Register

tbu : Time Base Register

The rename Buffers are used, if the Destination of one command is the source of another

---

command, for optimization purposes. The xer contains the carry and overflow bits. The lr contains the branch destination and the return address, when appropriate (for branch and link instructions). The ctr is decremented and tested automatically as result of branch and counter operations. The Timebase registers can be read by userlevel software, but written only by supervisorlevel software.

The condition register is split into 8 4-bit parts, named cr0-cr7. For each crn, the bits are defined as follows:

0 LT "less than"

1 GT "greater than"

2 EQ "equal"

3 SO "sum overflow"

Normally you use cr0 to store this information, but you can also use other CRs. You should be aware, that, if you use the below specified . Notation, all Integer Commands update cr0 according to the result of the operations, and all FPU Commands update the cr1 with the current setting of the FEX/FX/VX/OX Bits like described in the sections about the FPU Exceptions and the FPSCR. Also the o Notation (also described below) modifies (for Integer Commands) the SO Bit of cr0, and (for FPU Commands) the OX Bit of cr1. Asides from that you are completely free to use whatever crn you want for your comparisions and other cr-using operations. Usually you simply use cr0, though...

Read more about this at [FPSCR and XER](#) .

Concerning the condition fields cr0-cr7 and concerning XER, it has to be noted, that some commands have special "modes". If an o is appended to the command name, it reacts on "sum overflow" and "overflow" by setting the SO and OV Bits in the XER to the correct value. If a . is appended to the command name, it updates Bits 0-3 of cr0 for Integer-Commands or Bits 0-3 of cr1 for FPU-Commands (to the corresponding bits in the FPSCR). Like outlined above, this would be LT/GT/EQ/SO in case of an Integer command, but FEX/FX/VX/OX for FPU-commands (which have a totally different meaning). Always remember, that the . or o Notation are not supported by ALL commands. This might sound VERY complicated now to you: Well, those . and o notations are additional features, in normal code you usually don't have to use them.

Examples:

```
add r6,r7,r8
```

```
addo r6,r7,r8
```

```
add. r6,r7,r8
```

```
addo. r6,r7,r8
```

This is not possible for all commands, and some commands do not support all modes.

Some commands additionally have a 32 and a 64 Bit modes. Also it should be noted, that the current PPC CPUs used on Amiga PPC Boards (PPC 603e and 604e) are 32 Bit CPUs, they

do *\*not\** have the 64 Bit commands. Also they do not have the 64 Bit versions that are available for some commands (like ldx for example).

All these things are notified in the command descriptions. Always have a close look in the description, though, if it only supports the . notation or also the o-notation.

Some commands only support the . notation, but not the o-notation.

Please consider: All commands using 64 Bit Command width (d) only exist on 64 Bit Implementations. This means: NOT ON POWERUP !!! There might be 64 Bit versions of PowerUP or other PPC Boards for the Amiga in the future, though.

On PowerUP, the PowerPC *\*always\** runs in Big-Endian mode (like you are used to from the 68k) !!!

Also always note: The Bit numbering on PPC is just the other way round like on 68k !!!

On PPC Bit 0 is the Bit to the left, while this is the Bit to the right on 68k !!!

Also, there are some important things about register usage:

- \* r0 has a special meaning. Do not use it, if you do not know EXACTLY what you do.
- \* r1 is the stack pointer
- \* r2 is used for Global Variable Stuff
- \* r3 is the Library Base Pointer (like a6 on 68k !!!)
- \* r13 is the local stack pointer
- \* If you change r13-r31, you have to restore the old values
- \* Never forget to do the Stackframe with prolog/epilog
- \* Never forget the executable command
- \* Function-Parameters are transferred on r4-r10 (and possible the stack also)
- \* When there is talk about macros, the macros in powerpc/ppcmacros.i are meant

## 1.39 MagicSNs PowerPC ASM Coding Tutorial

Branch and Compare Commands

The PPC defines it's branch commands in a rather abstract way. The full syntax of the bc command for example looks like that:

bc BO,BI,target (bc 12,1,target would mean bgt target, for example)

where BO are special options like options to activate a dbra-like (dbra is a well known 68k ASM command) function and BI is the condition code as an number.

A Compare would look like this:

cmpi BF,L,RA,SI or cmp BF,L,RA,RB or cmpli BF,L,RA,UI or cmp BF,L,RA,RB

Because of that i won't go into

detail for the abstract definition, but i will only specify the "extended"

commands which are much more easy to use. I do not want to completely handle the PPC ASM, i only want to handle the things which make sense on the existing Amiga-PowerPC-WarpUP system.

Simple Branch Commands

Branch Commands with EA in LR

Branch Commands with EA in CTRL

Compare Commands

Counter Commands

Branch Prediction

Coding Subroutines

Coding Library Functions

## 1.40 MagicSNs PowerPC ASM Coding Tutorial

Branch Prediction

The PowerPC has Branch Prediction, which means it can predict, in which direction a branch will PROBABLY go. If this comes true, then, you will enhance the speed of your code (yes, this is a special RISC feature !!!). The code will run correctly also if the predictions are wrong, though. But if they are correct, it will run FASTER. It won't run much faster, though, unless really consequently used (like an optimizer of a compiler would do...). Probably one of the optimization techniques only a real compiler can use really effectively. But, anyways... here it is (there are other optimization techniques that also can be used for "human" coders, on which i will provide information later):

On default, the jump prediction mechanism assumes:

- \* conditioned branches, which jump to an address in LR or CTRL will be predicted that they will NOT be taken
- \* conditioned branches, that branch backward, will be assumed TO BE TAKEN
- \* conditioned branches, that branch forward, will be assumed to NOT be taken

Now it is possible that the prediction is wrong. Because of that you can manually modify the branch prediction. If you add a + to the command, the branch will probably be taken, if you add a -, probably not.

Remember: Even if you do NOT think about branch prediction, your code wil STILL RUN (which is NOT the case for all RISC CPUs, but luckily for the PPC).

Examples:

blt- label

beq+ test



## 1.41 MagicSNs PowerPC ASM Coding Tutorial

### Counter Commands

I used the term "Counter Commands" for commands that work like the dbra of the 68k.

"Counter Commands" can be used with **Simple Branch Commands** , with

**Branch Commands with EA in LR** and with Branch Commands with

EA in CTR" link eactr}. To use a "Counter Command", you have to insert one of the

following after the 'b' of the branch-command. Note: To use the Counter

Commands with t+f+..., you have to use **Condition Bit Addressing Mode** .

Conditioned Branches cannot use Counter commands !!! Something like beqntz is illegal !!!

t - jump, if condition is true

f - jump, if condition is false

dnz - decrement CTR and jump, if CTR is not 0

dz - decrement CTR and jump, if CTR is 0

dnzt - decrement CTR and jump, if CTR is not 0 AND condition is true

dzt - decrement CTR and jump, if CTR is 0 AND condition is true

dnzf - decrement CTR and jump, if CTR is not 0 AND condition is false

dztf - decrement CTR and jump, if CTR is 0 AND condition is false

The commands bccctr and bccctrl only support the t and f counter commands.

Examples:

bdnz label

bdnzt EQ,label

bdnzt 4\*cr3+LT,label

bt EQ,label

btl 4\*cr5+EQ,label

bdnzflr

## 1.42 MagicSNs PowerPC ASM Coding Tutorial

### Branch Commands

The simple branch commands that will be useful for use, would be:

b target

bl target

bcc target

bccl target

b is a relative branch. Something like bra on the 68k.

bl is a jump to subroutine (like bsr on the 68k).

bcc is a conditional branch (like bcc on the 68k).

bccl is a conditional jump to a subroutine (does not exist on the 68k).

You can also use absolute branches by adding a `a` to the branch command (for example `bcca`), but this is normally not that much used, as if you jump to a certain label, this is always done in relative mode (the assembler calculates the relative address for a label, not the absolute address). But well, commands for absolute branching exist.

The possible conditions to use as 'cc' in `bcc` and `bcl` would be:

- lt - Jump, if first operand < second operand
- le - Jump, if first operand <= second operand
- eq - Jump, if first operand = second operand
- ge - Jump, if first operand >= second operand
- gt - Jump, if first operand > second operand
- nl - Jump, if first operand >= second operand
- ne - Jump, if first operand <> second operand
- ng - Jump, if first operand <= second operand
- so - Jump, if Sum Overflow happened
- ns - Jump, if no Sum Overflow happened
- un - Jump, if FP-Compare with NAN's (look at still to be written FPU-Docs)
- nu - Jum, if no FP-Compare with NAN's (look at still to be written FPU-Docs)

Examples:

```
beq check
bgt .startit
b goforit
blel calculate
bl test
```

All Subroutine opcodes will put the address to jump back to into the Link-register.

More information about subroutines (especially subroutines calling subroutines needs some extra-code) is given in a special chapter of these docs.

All Branch Commands support the **Condition Field Addressing mode** and the **Condition Bit Addressing mode** .

## 1.43 MagicSNs PowerPC ASM Coding Tutorial

### Condition Bit Addressing Mode

With the "Condition Bit Addressing Mode" you can access special bits in special condition fields.

For example:

```
bdnz 4*cr2+EQ,label
```

uses the EQ-Bit of `cr2`, instead of the default field `cr0`.

If you leave out the "`4*crn`", then a bit of `cr0` will be used, for example:

```
bdzn EQ,label
```

Note: Condition Bit Addressing Mode does not make sense with Extended Commands like `beq`. It is not legal to USE it with Extended commands like `beq`.

## 1.44 MagicSNs PowerPC ASM Coding Tutorial

Branch Commands with EA in LR

blr

blrl

bcclr

bccrlr

blr jumps back from a subroutine to the address found in the link-register.

blrl jumps to the address found in the link-register, and before

it jumps, it puts the address of the command after the current command into the link register, so it is possible to jump back to this place AGAIN.

bcclr and bccrlr are simply conditioned versions of blr and blrl.

Plain spoken these commands overtake the role, what rts would be on the 68k, but they can do a bit more stuff than the simple rts.

Examples:

beqlr

blr

bnelrl

## 1.45 MagicSNs PowerPC ASM Coding Tutorial

Branch Commands with EA in CTR

bctr

bctrl

bccctr

bccctrl

These functions nearly do the same like the "Branch Commands with EA in LR". The only difference is, that these commands take the EA to jump to out of the Counter-Register.

Examples:

beqctr

bctr

bnctrl

## 1.46 MagicSNs PowerPC ASM Coding Tutorial

Compare Commands

cmpwi Rx,val

cmpdi Rx,val

These commands compare the value of the register with an integer and sets the Condition

---

Flags accordingly. After this conditioned branches can be used to jump accordingly.

These functions do a signed compare.

In case you need an unsigned compare, use:

```
cmplwi Rx,val
```

```
cmpldi Rx,val
```

You also can compare registers directly, using:

```
cmpw Rx,Ry
```

```
cmpd Rx,Ry
```

```
cmplw Rx,Ry
```

```
cmpld Rx,Ry
```

There are no 8 Bit or 16 Bit Compare Commands.

It is also possible to use **Condition Field Addressing Mode** .

Examples:

```
cmplwi r3,1
```

```
cmpw cr3,r4,r5
```

## 1.47 MagicSNs PowerPC ASM Coding Tutorial

FPSCR and XER

The FPSCR stores some information about the state of the FPU, even if you don't chose a command with CR Update (. Notation). In a similar way, XER serves the Integer-Commands, BTW.

If CR Update is chosen 4 Bits of the FPSCR are put to CR1:

FX -> Bit 0

VEX -> Bit 1

VX -> Bit 2

OX -> Bit 3

FPSCR consists of (i did not write much about it, as it is not interesting for the most programmers, if you need more info, read pem64b.pdf Chapter 2). WarpOS handles all Exception Stuff...

Status Bits:

0 FX: Floating Point Exception Bit

1 FEX: Floating Point Enabled Exception Summary Bit

2 VX: Floating Point Invalid Operation Exception Bit.

3 OX: Floating Point Overflow Bit

4 UX: Floating Point Underflow Bit

5 ZX: Floating Point Zero Devide Bit

6 XX: Floating Point Inexact Exception Bit

7 VXSNaN: Floating Point Invalid Operation for SNAN Bit

- 8 VXISI: Floating Point Invalid Operation for Infinity-Infinity Bit
  - 9 VXIDI: Floating Point Invalid Operation for Infinity/Infinity Bit
  - 10 VXZDZ: Floating Point Invalid Operation for Zero/Zero Bit
  - 11 VXZMDZ: Floating Point Invalid Operation for Infinity\*Zero Bit
  - 12 VXVC: Floating Point Invalid Operation for Invalid Compare Bit
  - 13 FR: Floating Point Fraction Rounded
  - 14 FI: Floating Point Fraction Inexact
  - 15 FPRF: 15: Floating Point Result Class
  - 16: FL (<)
  - 17: FG (>)
  - 18: FE (=)
  - 19: FU (UN)
  - 20 Reserved
  - 21 VXSOFT Floating Point Invalid Operation for Software Request Bit
  - 22 VXSQRT Floating Point Invalid Operation for Invalid Square Root Bit
  - 23 VXCVI Floating Point Invalid Operation for Invalid Integer Convert
- FEX=(VX&VE)|(OX&OE)|(UX&UE)|(ZX&ZE)|(XX&XE)

Control Bits:

- 24 VE Floating Point Invalid Operation Exception Enable
- 25 OE IEEE Floating Point Overflow Exception Enable
- 26 UE IEEE Floating Point Underflow Exception Enable
- 27 ZE IEEE Floating Point Zero Divide Exception Enable
- 28 XE IEEE Floating Point Inexact Exception Enable
- 29 NI Floating Point Non-IEEE mode
- 30 RN 30+31: Floating Point Rounding Control

- 00 Round to Nearest
- 01 Round towards Zero
- 10 Round towards +Infinity
- 11 Round towards -Infinity

With the exception of FEX and VX a FPSCR Bit remains set until it is cleared using mcrfs, mtfsi or mtfsb0. FEX and VX are logical ORs of other Bits. They are affected by every function that affects these Bits.

Overview of Result Classes:

C <> = ? Class

=====

- 1 0 0 0 1 Quiet NAN
- 0 1 0 0 1 -Infinity
- 0 1 0 0 0 -Normalized Number
- 1 1 0 0 0 -Denormalized Number

1 0 0 1 0 -Zero  
 0 0 0 1 0 +Zero  
 1 0 1 0 0 +Denormalized Number  
 0 0 1 0 0 +Normalized Number  
 0 0 1 0 1 +Infinity

In a similar way the XER stores data that result of Integer Arithmetics. If CR Update is enabled, some of these results are also written to CR0. The XER normally should not be accessed directly by the user

0 SO Summary Overflow Bit  
 1 OV Overflow Bit  
 2 CA Carry Bit  
 25 25-31: Number of Bytes to be transferred by a lswx or stswx command

## 1.48 MagicSNs PowerPC ASM Coding Tutorial

Condition Field Addressing Mode"

This addressing mode enables you to use a different condition field than the default field cr0. Please remember, that cr1 is used for FPU-usage. If you use the . or o notation (well, you probably won't use it...), you have to remember, that it changes cr1.

You have cr0 to cr7. To specify which field to use, you can do like this (using a compare command as example, works the same with other commands using the condition register, for example logic operations):

```
cmp cr3,r3,r4
```

An Example for a branch command would be:

```
beq cr3,label
```

## 1.49 MagicSNs PowerPC ASM Coding Tutorial

Load/Store Commands

The PPC is a RISC. Probably all RISCs have a "Load/Store Architecture". Now, what does this mean ? You have a LOT of commands to "put something from memory to a register" (Load) or to "put something from a register to the memory" (Store). More complex Addressing modes are often missing.

Now, what sort of Load/Store can the PPC do ?

In the following b will mean Byte-Access will mean 8 Bit Access.

In the following h will mean Halfword-Access will mean 16 Bit Access.

In the following w will mean Word-Access will mean 32 Bit Access.

In the following d will mean Doubleword-Access will mean 64 Bit Access.

This listing is not complete. It only lists the commands that are valid with StormPowerASM and make sense with StormPowerASM (i hope it is in THIS WAY complete... it is a large list : ) )

All load unsigned Load Commands fill the rest of the register, in case no 64 Bit (32 Bit for 603e/604e) commands are used with 0. With signed data, the rest is filled with the sign. This does not apply to Store Commands (where only the really accessed data is changed).

Please note, that 64 Bit Load/Store commands (d) have to use data, which address is divisible by 4 !!!

First we split the commands into:

Load Zero Commands

Load Zero Indexed Commands

Load Zero with Update Commands

Load Zero with Update Indexed Commands

Load Algebraic Commands

Load Algebraic Indexed Commands

Load Algebraic with Update Commands

Load Algebraic with Update Indexed Commands

Simple Store Commands

Indexed Store Commands

Store Commands with Update

Indexed Store Commands with Update

Load Commands with Little/Big Endian Conversion

Store Commands with Little/Big Endian Conversion

The la Command

Additional we have some Macros to do:

Load Variable Commands

Load Signed Variable Commands

Load Signed Byte Commands

Load Constant Commands

Store Register to Variable Commands

Store Constant to Variable Commands

Store Constant to Memory Commands

Store Constant to Memory Indexed Commands

Store Constant to Memory with Update Commands

Store Const. to Mem. Indexed with Update

It should be noted, that the ONLY legal way to handle variables is using these Macros. You should not try this without these Macros.

## 1.50 MagicSNs PowerPC ASM Coding Tutorial

Load Zero Commands

```
lbz Rx,d16(Ry)
```

```
lhz Rx,d16(Ry)
```

```
lwz Rx,d16(Ry)
```

```
ld Rx,d16(Ry)
```

These commands take a value specified by an effective address which is contained partially in a register Ry, partially given by a 16 Bit Offset, and put it into a register Rx. These commands only should be used for unsigned data. For signed data, the lba Macro should be used. 16/32/64 Bit could be done by a

```
lhz Rx,d16(Ry)
```

```
extsh Rx,Rx
```

(for w on 32 Bit or d on 64 Bit the Extension command can simply be left out).

Alternatively you can use the processor "Load Algebraic Commands".

Examples:

```
lwz r4,40000(r5)
```

```
lhz r10,$a0(r7)
```

If you use r0 as Base Register (Ry), then the value 0 will be used instead.

## 1.51 MagicSNs PowerPC ASM Coding Tutorial

Load Zero Indexed Commands

```
lbzx Rt,Ra,Rb
```

```
lhzx Rt,Ra,Rb
```

```
lwzx Rt,Ra,Rb
```

```
ldx Rt,Ra,Rb
```

These commands will read a Byte/Halfword/Word/Doubleword from the address calculated by adding the values of Ra and Rb and put this into Rt. The data is put into the low Byte/Halfword/Word, if needed, the rest filled with 0 Bits. If r0 is used, 0 is used instead. These commands should only be used for unsigned data. For signed data the macro lbax should be used. For greater widths like 8 Bit with lba, the same like for lba applies.

Example:

```
lwzx r3,r4,r8
```

---



## 1.52 MagicSNs PowerPC ASM Coding Tutorial

Load Zero with Update Commands

```
lbzu Rx,d16(Ry)
```

```
lhzu Rx,d16(Ry)
```

```
lwzu Rx,d16(Ry)
```

```
ldu Rx,d16(Ry)
```

These commands read from the address calculated by adding a 16 Bit Offset to the content of Ry and put this into Rx. After the data is loaded, Ry will be incremented by d16, using this command (Update). You only should use this command for unsigned data. For Signed data you can use the macro lbau or lhzu Rx,d16(Ry)

```
extsh Rx,Rx
```

For w on 32 Bit or d on 64 Bit, the Extension Command is not needed.

Alternatively you can use the processor "Load Algebraic with Update Commands".

It is not allowed to use r0 as base register. It is not allowed to use the same register for Rx and Ry.

Example:

```
lhzu r5,$45(r15)
```

## 1.53 MagicSNs PowerPC ASM Coding Tutorial

Load Zero with Update Indexed Commands

```
lbzux Rt,Ra,Rb
```

```
lhzux Rt,Ra,Rb
```

```
lwzux Rt,Ra,Rb
```

```
ldux Rt,Ra,Rb
```

These commands read from the address calculated by adding Ra and Rb and put the data found there to Rt then. After this Ra is set to Ra+Rb (Update). You should only use these commands for unsigned data. For signed Data you can use the macro lbaux or

```
lhzux Rx,d16(Ry)
```

```
extsh Rx,Rx
```

For w on 32 Bit or d on 64 Bit, the Extension Command is not needed.

Alternatively you can use the processor "Load Algebraic Indexed with Update Commands".

It is not allowed to use Ra=r0 or the same register for Rt and Ra.

Example:

```
ldux r30,r15,r28
```

## 1.54 MagicSNs PowerPC ASM Coding Tutorial

Load Algebraic Commands

```
lha Rx,d16(Ry)
```

```
lwa Rx,d16(Ry)
```

These commands read from the address calculated by Ry, with 16 Bit Offset d16 added and put the data found there to Rx. The data should be signed. For unsigned data use the "Load Zero Commands". The l\*a Commands do not have a 64 Bit Version.

For 8 Bit the Macro lba should be used.

In case r0 is used as Ry, the value 0 is used instead.

Example:

```
lwa r5,0(r21)
```

## 1.55 MagicSNs PowerPC ASM Coding Tutorial

Load Algebraic Indexed Commands

```
lhax Rt,Ra,Rb
```

```
lwax Rt,Ra,Rb
```

These commands read from the address calculated by adding Ra and Rb and put the data found there to Rx. The data should be signed. For unsigned data use the "Load Zero Indexed Commands". The l\*ax Commands do not have a 64 Bit Version.

For 8 Bit the Macro lbax should be used.

In case r0 is used for Ra, the value 0 is used instead.

Example:

```
lhax r3,r7,r8
```

## 1.56 MagicSNs PowerPC ASM Coding Tutorial

Load Algebraic with Update Indexed Commands

```
lhaux Rt,Ra,Rb
```

```
lwaux Rt,Ra,Rb
```

These commands read from the address calculated by adding Ra and Rb and put the data found there to Rx. The data should be signed. For unsigned data use the "Load Zero with Update Indexed Commands". The l\*aux Commands do not have a 32 Bit Version. For 8 Bit the Macro lbaux should be used.

r0 is not allowed as Ra. Ra should not be the same register like Rt.

Example:

```
lwaux r3,r7,r8
```

---

## 1.57 MagicSNs PowerPC ASM Coding Tutorial

Simple Store Commands

```
stb Rx,d16(Ry)
```

```
sth Rx,d16(Ry)
```

```
stw Rx,d16(Ry)
```

```
std Rx,d16(Ry)
```

These commands take the data in Rx and store it in the address calculated by adding Ry and a 16 Bit Offset. If you use r0, 0 is used instead.

Example:

```
stw r5,0(r4)
```

## 1.58 MagicSNs PowerPC ASM Coding Tutorial

Indexed Store Commands

```
stbx Rt,Ra,Rb
```

```
sthx Rt,Ra,Rb
```

```
stwx Rt,Ra,Rb
```

```
stdx Rt,Ra,Rb
```

These commands take the data in Rt and store it in the address calculated by adding Ra and Rb. If you use r0, 0 is used instead.

Example:

```
stdx r5,r7,r8
```

## 1.59 MagicSNs PowerPC ASM Coding Tutorial

Store Commands with Update

```
stbu Rx,d16(Ry)
```

```
sthv Rx,d16(Ry)
```

```
stwu Rx,d16(Ry)
```

```
stdv Rx,d16(Ry)
```

These commands take the data in Rx and store it in the address calculated by adding Ry and a 16 Bit Offset. After that, d16 is added to Ry.

Example:

```
stwu r5,$10(r4)
```

---

## 1.60 MagicSNs PowerPC ASM Coding Tutorial

Indexed Store Commands with Update

```
stbux Rt,Ra,Rb
```

```
sthux Rt,Ra,Rb
```

```
stwux Rt,Ra,Rb
```

```
stdux Rt,Ra,Rb
```

These commands take the data in Rt and store it in the address calculated by adding Ra and Rb. After that Rb is added to Ra.

Example:

```
sthux r5,r7,r8
```

## 1.61 MagicSNs PowerPC ASM Coding Tutorial

Load Commands with Little/Big Endian Conversion

On WarpUP, the PowerPC operates in its Big Endian mode (the same mode like the 68k uses also). Now PCs use Little Endian (Byte-Swapped) mode. There is a Load Command which automatically swaps Bytes/Halfwords.

```
lhbrx Rt,Ra,Rb
```

```
lwbrx Rt,Ra,Rb
```

The data at Ra+Rb is taken, then it will be Byteswapped/Halfwordswapped and then written to Rt. Useful for coding PC Emulators. :)

There is no 64 Bit Version of this command, and there is no lhbrux or lhbr or lhbru, only lhbrx and lwbrx. Certainly no 8 Bit version also (for 8 Bit, you do not need to swap...) If r0 is used as base register (Ra), the value 0 is used instead.

## 1.62 MagicSNs PowerPC ASM Coding Tutorial

Store Commands with Little/Big Endian Conversion

On WarpUP, the PowerPC operates in its Big Endian mode (the same mode like the 68k uses also). Now PCs use Little Endian (Byte-Swapped) mode. There is a Store Command which automatically swaps Bytes/Halfwords.

```
sthbrx Rt,Ra,Rb
```

```
stwbrx Rt,Ra,Rb
```

Rt is taken and written to the address calculated by Ra+Rb. Before it is written, the data will be Byteswapped/Halfwordswapped. Useful for coding PC Emulators. :)

There is no 64 Bit Version of this command, and there is no sthbrux or sthbr or sthbru, only lhbrx and lwbrx. Certainly no 8 Bit version also (for 8 Bit, you do not need to swap...) If r0 is used as base register (Ra), the value 0 is used instead.

## 1.63 MagicSNs PowerPC ASM Coding Tutorial

Load Variable Commands

`lb Rx,Variable`

`lh Rx,Variable`

`lw Rx,Variable`

These macros load a Variable to a register. There is no 64 Bit Version of these

Macros. Variables NEVER should be loaded without using the Macros. Rx should NEVER be r0. Variable should be unsigned, if the 8/16 Bit Version is used.

The Variable is loaded to the Bottom part of the registers. The rest is filled with 0-Bits.

Example:

```
lw r7,test
```

## 1.64 MagicSNs PowerPC ASM Coding Tutorial

Load Signed Variable Commands

`lbs Rx,Variable`

`lhs Rx,Variable`

These macros load a Variable to a register. There is no 64 Bit Version of these

Macros. For Signed 32 Bit Variables the `lw` Macro can be used.

Variables NEVER should be loaded without using the Macros. Rx should NEVER be r0. Variable should be unsigned. The Variable is loaded to the Bottom part of the registers, the rest is filled with 0-Bits.

Example:

```
lbs r18,test
```

## 1.65 MagicSNs PowerPC ASM Coding Tutorial

Load Signed Byte Commands

`lba Rx,d16(Ry)`

`lbax Rt,Ra,Rb`

`lbau Rx,d16(Ry)`

`lbaux Rt,Ra,Rb`

These Macros add an 8 Bit Version of the "Algebraic Commands". Please read in the section about "Algebraic Commands" about more information.

---

## 1.66 MagicSNs PowerPC ASM Coding Tutorial

Load Constant Commands

```
li Rx,d16
```

```
lis Rx,d16
```

```
liw Rx,d32
```

These commands are used to load 16/32 Bit Constants into registers. li/lis deals with unsigned constants or signed constants <32767. If a signed constant is bigger than 32767, the macro liw should be used (li/lis is a CPU command, liw a macro).

liw can be used for all sorts of 32 Bit Constants.

Example:

```
liw, r3,40
```

## 1.67 MagicSNs PowerPC ASM Coding Tutorial

The la Command

```
la Rx,variable
```

la loads the address of a variable to a register. With this commands you can access a variable like this:

```
la r3,test
```

```
lwz r4,0(r3)
```

This is, according to the StormPowerASM Docs, inefficient, so you should better use the Macros instead. And don't try to use r0 with this command.

## 1.68 MagicSNs PowerPC ASM Coding Tutorial

Store Register to Variable Commands

```
sb Rx,variable
```

```
sh Rx,variable
```

```
sw Rx,variable
```

These macros store a register into a variable. Please do not use CPU commands for this job, use these macros. Do not use r0 as Rx. There is no 64 Bit Version of these commands.

Example:

```
sw r5,test
```

## 1.69 MagicSNs PowerPC ASM Coding Tutorial

Store Constant to Variable Commands

```
sbi const,variable
```

```
shi const,variable
```

```
swi const,variable
```

These macros store a constant into a variable. Please do not use CPU Commands for this job, use these macros. There is no 64 Bit Version of these commands.

Example:

```
shi $FFA0,test
```

## 1.70 MagicSNs PowerPC ASM Coding Tutorial

Store Constant to Memory Commands

```
stbi const,d16(Rx)
```

```
sthi const,d16(Rx)
```

```
stwi const,d16(Rx)
```

These macros store a constant to memory. There is no 64 Bit Version of these commands.

It could be easily defined, though. If you use r0, the value 0 is used instead.

The memory address is calculated by adding a 16 Bit Offset to Rx.

Example:

```
sthi 46,16(r17)
```

## 1.71 MagicSNs PowerPC ASM Coding Tutorial

Store Constant to Memory Indexed Commands

```
stbix const,Ra,Rb
```

```
sthix const,Ra,Rb
```

```
stwix const,Ra,Rb
```

These macros store a constant to memory. There is no 64 Bit Version of these commands.

It could be easily defined, though. If you use r0, the value 0 is used instead.

The Memory Address is calculated by adding Ra and Rb.

Example:

```
stbix $20,16(r17)
```

---

## 1.72 MagicSNs PowerPC ASM Coding Tutorial

Store Constant to Memory with Update Commands

```
stbiu const,d16(Rx)
```

```
sthiu const,d16(Rx)
```

```
stwiu const,d16(Rx)
```

These macros store a constant to memory. There is no 64 Bit Version of these commands.

It could be easily defined, though. The memory address is calculated by adding a

16 Bit Offset to Rx. r0 is not allowed as Base Register (Rx). After the store,

d16+Rx is written to Rx (Update).

Example:

```
sthiu 46,16(r17)
```

## 1.73 MagicSNs PowerPC ASM Coding Tutorial

Store Constant to Memory Indexed with Update Commands

```
stbiux const,Ra,Rb
```

```
sthiux const,Ra,Rb
```

```
stwiux const,Ra,Rb
```

These macros store a constant to memory. There is no 64 Bit Version of these commands.

It could be easily defined, though. The Memory Address is calculated by adding Ra

and Rb. r0 is not allowed as Base Register (Ra). After the store, Ra+Rb is written

to Ra (Update).

Example:

```
stbiux $20,16(r17)
```

## 1.74 MagicSNs PowerPC ASM Coding Tutorial

Some words about Optimizing

The PowerPC is a RISC, but it is a very special RISC. There are many RISC CPUs, which

will produce bad code, if you do not handle the Pipelines correctly. The PowerPC does

NOT behave this way. You can code it like the 68k. There are even a lot of parallels

to the 68k ASM language which make learning PPC ASM easy, if you already know 68k ASM.

If you do "Scheduling" (optimizing with a look at the pipelines) you can speed up your

code about 50%, though. I probably will write something about optimizing, later.

Also have a look at the chapter about [Branch prediction](#) for optimization.



## 1.75 MagicSNs PowerPC ASM Coding Tutorial

Some words about 68k->PPC ASM Porting

Porting 68k ASM sources to PPC ASM is \*easy\*. I'll show you with an example (it is a simple WritePixel Function, part of rtgmaster, in the 68k and PPC version).

BTW: Nobody said you have to port the WHOLE program. Test the speed of the different functions of your program, to find out which take the most CPU time. In most programs these are quite few and often quite small functions (like for example Rendering Loops).

The rest you still run as 68k code, then.

One thing about 68k->PPC Porting should be additionally noted:

The PPC supports a lot of commands only in 32 (and sometimes in 64) Bit. For example there is \*no\* shift-command which works in the lower 16 Bit and lets the upper 16 Bit unchanged. Something like:

```
lsl.w #3,d0
```

would be

```
rlwinm r0,r3,3,16,31 ;shift 3 left + mask out with $ffff
```

```
inslwi r0,r3,16,0 ;copy Bitfield r3[0-15] to r0[0-15]
```

```
mr r3,r0 ;put result to r3
```

on PPC. So if you do 68k/PPC parallel development, it is strongly advised, that you only use .L commands, if possible (or at least allow the Upper 16 Bit Word to be changed ALWAYS or at least document in the source while writing it, if the Upper 16 Bit Word has to be preserved or not).

Another useful hint: Often a mr is not really needed (well, in the above case it IS needed...) and can be replaced by using a 3+ operand command.

For example, instead of:

```
add r5,r5,r6
```

```
mr r7,r5
```

do a:

```
add r7,r5,r6
```

You'll get used to it :)

In case you need help with 68k->PPC Porting, if you do not want to do this yourselves, please contact me:

MagicSN@Birdland.es.bawue.de

Steffen Haeuser

Limburgstr. 127

73265 Dettingen/Teck

Germany

Phone: 07021/51787 (Ask for Steffen)

I am interested in helping, especially with commercial game and freeware demo

projects :)

Now the Example:

68k: PPC:

====

```
include "rtgmaster/rtgCGX.i" include "rtgmaster/rtgCGX.i"
```

```
include "powerpc/ppcmacros.i"
```

```
XDEF _WriteRtgPixel XDEF _WriteRtgPixel
```

```
_WriteRtgPixel: _WriteRtgPixel:
```

```
; a0 = RtgScreen ; r4 = RtgScreen
```

```
; a1 = BufferAdr ; r5 = BufferAdr
```

```
; d0 = posx ; r6 = posx
```

```
; d1 = posy ; r7 = posy
```

```
; d2 = color ; r8 = color
```

```
movem.l d3/d4,-(a7) prolog
```

```
move.l rsCGX_Bytes(a0),d3 lwz r9,rsCGX_Bytes(r4)
```

```
move.l rsCGX_Width(a0),d4 lwz r10,rsCGX_Width(r4)
```

```
cmp.l #1,rsCGX_Bytes(a0) cmpwi r9,1
```

```
beq .Chunky beq .Chunky
```

```
bra .Exit b .Exit
```

```
.Chunky: .Chunky:
```

```
move.l a1,a0 mr r4,r5
```

```
add.l d0,a0 add r4,r6,r4
```

```
mulu d1,d4 mullw r10,r7,r10
```

```
add.l d4,a0 add r4,r10,r4
```

```
move.b d2,(a0) stb r8,0(r4)
```

```
.Exit: .Exit:
```

```
movem.l (a7)+,d3/d4 epilog
```

```
rts
```

## 1.76 MagicSNs PowerPC ASM Coding Tutorial

Rotate and Shift Commands

[Masking](#)

[Rotate Commands](#)

[Shift Commands](#)

[Extended Commands](#)

[Multiple Precision Shifts](#)

## 1.77 MagicSNs PowerPC ASM Coding Tutorial

### Multiple Precision Shifts

A Multiprecision Shift is a Shift of a n-Word Quantity (for example you could Shift a 96-Bit-Number Left 13 Bits). Of course this data does not fit to a Single register. Because of that, it is put into SEVERAL registers. I am giving the 32 Bit Examples out of the Motorola PowerPC Documentation in this section. I am not giving the 64 Bit Examples here, as the 32 Bit ones are already enough code to read, and as there is currently no 64 Bit PowerPC (PPC 620 ...) used on the Amiga. PowerPC 604e is still a 32 Bit PowerPC.

The Examples use  $n=2$  and  $n=3$ . It is possible to code multiprecision shifts in a similar way for higher  $n$ , though. A Multiprecision Shift takes  $2n-1$  (immediate Shifts) or  $3n-1$  (non-immediate Shifts) commands, if coded optimal. The Shift Amount is allowed to be 0-63, if  $n=2$ . If  $n>2$ , it is only allowed to be 0-31.

In the examples r2,r3 and r4 contain the data to be shifted. The result will be put in the same registers, again. The lowest-numbered register contains the highest-order data, and the highest-numbered register the lowest-order data. For non-immediate shifts, the shift amount is assumed in Bits 58-63 of r6. r0-r31 are used as scratch registers (principally, in this example only r0,r2,r3,r4,r6 and r31 are used).

Note: Multi Precision Shifts look VERY complicated !!! Don't wonder, if you don't understand them at once...

Also note: r2 and r3 have special meaning on WarpOS. You have to store the original values or use other registers for the Multiple Precision Shift.

Examples:

Shift Left Immediate,  $n = 3$

```
rlwinm r2,r2,sh,0,31-sh
```

```
rlwimi r2,r3,sh,32-sh,31
```

```
rlwinm r3,r3,sh,0,31-sh
```

```
rlwimi r3,r4,sh,32-sh,31
```

```
rlwinm r4,r4,sh,0,31-sh
```

Shift Left,  $n = 2$

```
subfic r31,r6,32
```

```
slw r2,r2,r6
```

```
srw r0,r3,r31
```

```
or r2,r2,r0
```

```
addi r31,r6,-32
```

```
slw r0,r3,r31
```

---

```
or r2,r2,r0
slw r3,r3,r6
Shift Left, n = 3
subfic r31,r6,32
slw r2,r2,r6
srw r0,r3,r31
or r2,r2,r0
slw r3,r3,r6
srw r0,r4,r31
or r3,r3,r0
slw r4,r4,r6
Shift Right Immediate, n = 3
rlwinm r4,r4,32-sh,sh,31
rlwimi r4,r3,32-sh,0,sh-1
rlwinm r3,r3,32-sh,sh,31
rlwimi r3,r2,32-sh,0,sh-1
rlwinm r2,r2,32-sh,sh,31
Shift Right, n = 2
subfic r31,r6,32
srw r3,r3,r6
slw r0,r2,r31
or r3,r3,r0
addi r31,r6,-32
srw r0,r2,r31
or r3,r3,r0
srw r2,r2,r6
Shift Right, n = 3
subfic r31,r6,-32
srw r4,r4,r6
slw r0,r3,r31
or r4,r4,r0
srw r3,r3,r6
slw r0,r2,r31
or r3,r3,r0
srw r2,r2,r6
Shift Right Algebraic Immediate (Sign-Correct), n = 3
rlwinm r4,r4,32-sh,sh,31
rlwimi r4,r3,32-sh,0,sh-1
rlwinm r3,r3,32-sh,sh,31
```

---

```

rlwimi r3,r2,32-sh,0,sh-1
srawi r2,r2,sh
Shift Right Algebraic (Sign-Correct), n = 2
subfic r31,r6,32
srw r3,r3,r6
slw r0,r2,r31
or r3,r3,r0
addi r31,r6,-32
srw r0,r2,r31
ble $+8 ; jump 8 Bytes from current position
ori r3,r0,0
sraw r2,r2,r6
Shift Right Algebraic (Sign-Correct), n = 3
subfic r31,r6,32
srw r4,r4,r6
slw r0,r3,r31
or r4,r4,r0
srw r3,r3,r6
slw r0,r2,r31
or r3,r3,r0
sraw r2,r2,r6

```

## 1.78 MagicSNs PowerPC ASM Coding Tutorial

### Extended Commands

Principally there are 6 types of Rotate/Shift Extended Commands. All these commands support the . Notation like described in **Important things** . They don't support the o Notation, though.

#### Extract:

Select a Field of n Bits starting at bit position b in the Source register; left or right justify this field in the target register and set the other bits of the target register to 0.

64 Bit: extldi Ra,Rs,n,b (n>0) (Left Justify)

extrdi Ra,Rs,n,b (n>0) (Right Justify)

32 Bit: extlwi Ra,Rs,n,b (n>0) (Left Justify)

extrwi Ra,Rs,n,b (n>0) (Right Justify)

#### Insert:

Select a left-justified or right-justified field of n Bits in the Source register; insert this field starting at bit position b of

the target register; leave other bits of the target register unchanged. For 64 Bit only the Right Justified Version exists.

64 Bit: `insrwi Ra,Rs,n,b (n>0)` (Right Justified)

32 Bit: `inslwi Ra,Rs,n,b (n>0)` (Left Justified)

`insrwi Ra,Rs,n,b (n>0)` (Right Justified)

Rotate:

Rotate the register Left or Right n Bits without Masking.

64 Bit: `rotldi Ra,Rs,n` (Left)

`rotrdi Ra,Rs,n` (Right)

`rotld Ra,Rs,Rb` (Left)

32 Bit: `rotlwi Ra,Rs,n` (Left)

`rotrwi Ra,Rs,n` (Right)

`rotld Ra,Rs,Rb` (Left)

Shift:

Shift the contents of a register Left or Right n Bits, clearing Bits to the Right/to the Left.

64 Bit: `sldi Ra,Rs,n (n<64)` (Shift Left)

`srdi Ra,Rs,n (n<64)` (Shift Right)

32 Bit: `slwi Ra,Rs,n (n<32)` (Shift Left)

`srwi Ra,Rs,n (n<32)` (Shift Right)

Clear:

Clear the leftmost or rightmost n Bits of a register.

64 Bit: `clrldi Ra,Rs,n (n<64)` (Clear Left)

`clrrdi Ra,Rs,n (n<64)` (Clear Right)

32 Bit: `clrlwi Ra,Rs,n (n<32)` (Clear Left)

`clrrwi Ra,Rs,n (n<32)` (Clear Right)

Clear Left and Shift Left:

Clear the Leftmost b Bits of a register, then shift the register left by n Bits (useful for scaling a non-negative array-index by the width of an element).

64 Bit: `clrslwi Ra,Rs,b,n (n<=b<=63)`

32 Bit: `clrslwi Ra,Rs,b,n (n<=b<=31)`

## 1.79 MagicSNs PowerPC ASM Coding Tutorial

### Masking

All PPC Rotate and Shift Commands use Masks. In the following the construction of these masks will be discussed.

A Mask is defined by a Start-Bit and a Stop-Bit. If for example

the Start-Bit is 20, and the Stop-Bit is 28, the Mask looks like this (Example for a 32 Bit Implementation to avoid a lot of leading 0s for this simple example : )

```
%00000000 00000000 00001111 11111000
```

Bits 20-28 are set to 1, all other bits to 0. (You notice again: Bit numbering on PPC is exactly the other way round like on 68k... Bit 0 is to the left, Bit 31 to the right...).

The Mask Construction of the PPC also supports Wrap-Around (for example, a Bits 28-5 Mask would be possible).

The Mask will then be AND'ed with the result of a Rotate/Shift Command, and this will make the final result, then.

## 1.80 MagicSNs PowerPC ASM Coding Tutorial

### Rotate Commands

All rotate commands support the . notation, like outlined in Important things" link imp}. The o Notation is not supported.

```
rldicl Ra,Rs,SH,MB
```

This command rotates Rs left SH Bits, the Mask has Bits MB to 63 set to 1, the rest to 0. The result is AND'ed with the Mask and put into Ra. Only implemented on 64 Bit Implementations.

```
rldicr Ra,Rs,SH,ME
```

The same like before, but the mask is with Bits 0-ME set to 1, the rest to 0.

Only implemented on 64 Bit Implementations.

```
rldic Ra,Rs,SH,MB
```

The same like before, but the mask is with Bits MB-(63-SH) set to 1, the rest to 0. Only implemented on 64 Bit Implementations.

```
rlwinm Ra,Rs,SH,MB,ME
```

This command rotates Rs left SH Bits, the Mask has Bits MB+32 to ME+32 set to 1, the rest to 0. The rotated data is AND'ed with the mask and saved into Ra. The high-word of the result is always cleared (because of the +32 with MB and ME).

This quite complicated command can especially be used to:

- \* to extract an a-Bit-Field that starts at bit-position b in the low-order of Rs, right justified into Ra, set SH = b+a, MB = 32-a and ME = 31.

- \* to extract an a-Bit-Field that starts at bit-position b in the low-order of Rs, left justified into Ra, set SH = b, MB = 0 and ME = a-1.

- \* to rotate the contents of a register left (or right) a bits, set SH = a (32-a), MB = 0, ME = 31.

\* to shift the contents of a register right by a bits, set  $SH=32-a$ ,  $MB = a$  and  $ME = 31$ . To clear the high-order b bits of a register and then shift the result left by a bits, set  $SH=a$ ,  $MB = b-a$  and  $ME = 31-a$

\* to clear the low-order a bits of a register, set  $SH=0$ ,  $MB=0$  and  $ME = 31-a$

Note: This command is VERY tricky. Take yourselves a piece of paper and try to work out the given examples, and try to understand them. I also recommend to have a look at the Extended Shift/Rotate command. A lot of special usages of `rlwinm` can simply be replaced by extended commands.

`rldcl Ra,Rs,Rb,MB`

This command rotates `Rs` the number of bits left, like specified in the low-order 6 Bits of `Rb`. The Mask is Bit `MB` to Bit 63 set to 1, the rest to 0. The Mask and the Result are AND'ed like always, to form `Ra`.

Only implemented on 64 Bit Implementations.

`rldcr Ra,Rs,Rb,ME`

The same like `rldcl`, only that the Mask is Bit 0 to Bit `ME` set to 1, the rest to 0. Only implemented on 64 Bit Implementations.

`rlwnm Ra,Rs,Rb,MB,ME`

The same like `rlwinm`, only that instead of `SH` the low-order 5 Bits of `Rb` are used.

This quite complicated command can especially be used to:

\* to extract an a-Bit-Field that starts at bit-position b in the high-order of `Rs`, right justified into `Ra`, set  $Rb = b+a$ ,  $MB = 32-a$  and  $ME = 31$ .

\* to extract an a-Bit-Field that starts at bit-position b in the high-order of `Rs`, left justified into `Ra`, set  $Rb = b$ ,  $MB = 0$  and  $ME = a-1$ .

\* to rotate the contents of a register left (or right) a bits, set  $Rb = a (32-a)$ ,  $MB = 0$ ,  $ME = 31$ .

Note: This command is VERY tricky. Take yourselves a piece of paper and try to work out the given examples, and try to understand them. I also recommend to have a look at the Extended Shift/Rotate command. A lot of special usages of `rlwnm` can simply be replaced by extended commands.

`rlwimi Ra,Rs,SH,MB,ME`

This command rotates `Rs` `SH` Bits Left, then specifies a Mask with `MB+32` to `ME+32` Bits set to 1, the rest to 0. Then it inserts the result of the rotation at the mask positions into `Ra`. Contrary to the similar `rlwinm` it does not do an AND, but it INSERTS the Bits into `Ra`, and leaves the Bits of `Ra`, where the Mask is 0 unchanged.

This quite complicated command can especially be used to:



\* to insert an a-bit field, that is left-justified in the low-order 32 Bits of Rs, into the high-order 32 Bits of Ra, starting at bit-position b, set SH = 32-b, MB = b, and ME = (b+a)-1

\* to insert an a-bit field, that is right-justified in the low-order 32 Bits of Rs, into the high-order 32 Bits of Ra starting at bit-position b, set SH = 32-(b+a), MB =b, and ME = (b+a)-1

Note: This command is VERY tricky. Take yourselves a piece of paper and try to work out the given examples, and try to understand them. I also recommend to have a look at the Extended Shift/Rotate commands.

rldimi Ra,Rs,SH,MB

This command is similar to the last one, only that the mask has Bits MB to 63-SH set to 1, rest to 0. It only exists on 64 Bit Implementations.

It can for example be used, to insert an a-bit field, that is right-justified in Rs, into Ra, starting at bit-position b, by setting SH=64-(b+a) and MB=b.

I recommend also to have a look at the Extended Shift/Rotate Commands.

## 1.81 MagicSNs PowerPC ASM Coding Tutorial

Shift Commands

Note: Divisions by  $2^n$  can be quickly done, by combining a Shift Right Algebraic and a addze instruction (adding the Carry-Flag to the Shifted Result). The Shift-Command already sets the Carry correctly.

All shift commands support the . notation, like outlined in **Important things** . The o Notation is not supported.

sld rA,Rs,Rb

srd rA,Rs,Rb

Shifts Rs left/right by the number of bits specified by the low-order 7 Bits of Rb, and puts the result to Ra. Shift amounts of 64-127 give a 0 result. Only exists on 64 Bit Implementations (64 Bit Shift).

slw Ra,Rs,Rb

srw Ra,Rs,Rb

Shifts the low-order word of Rs left/right by the number of bits specified by the low-order 6 Bits of Rb, and puts the result into Ra. Bits shifted out of the Low-Order word (or out of the register for 32 Bit Implementations) will be lost. In 64 Bit Implementations, the high-order 32 Bits are cleared, and shift amounts of 32-63 give a 0 result.

sradi Ra,Rs,SH

Rs is shifted SH Bits right. Bit 0 of Rs is used to fill new leading Bit-Positions of the result. The result is placed into Ra. If Rs contains a negative number and any Bits are shifted out of position 63, the Carry is set, otherwise it is cleared.

If SH=0, Ra is simply set to Rs, and the Carry is cleared. Only exists on 64 Bit Implementations.

srawi Ra,Rs,SH

The low-order 32 Bit of Rs is shifted SH Bits right. Bits shifted out of position 63 (position 31 for 32 Bit Implementations) are lost. Bit 32 of Rs fills new positions to the left for 64 Bit Implementations. The result will be sign-extended and then placed into the low-order 32 Bits of Ra.

srad Ra,Rs,Rb

Rs will be shifted right the number of Bits specified by the low-order seven Bits of Rb. Bits shifted out of position 63 are lost. Bit 0 of Rs is used to fill new positions to the left. The result is placed into Ra. Only exists on 64 Bit Implementations.

sraw Ra,Rs,Rb

The low-order 32 Bit of Rs is shifted the low-order six Bits of Rb Bits right. Bits shifted out of position 63 (31 for 32 Bit Implementations) are lost. Bit 32 of Rs is used to fill new positions to the left for 64 Bit Implementations. The result will be placed into the low-order 32 Bits of Ra.

## 1.82 MagicSNs PowerPC ASM Coding Tutorial

Condition-Logic Commands

crand crbD,crbA,crbB

cror crbD,crbA,crbB

crxor crbD,crbA,crbB

crnand crbD,crbA,crbB

crnor crbD,crbA,crbB

creqv crbD,crbA,crbB

crandc crbD,crbA,crbB

crorc crbD,crbA,crbB

These commands take two parameters crbA and crbB in

**Condition Bit Addressing Mode** and do a

AND/OR/XOR/NAND/NOR/EQV/NANDC/ORC with them, like defined in the

**Logic Commands** with them, and store the result

in the Condition Bit crbD. The Condition-Logic Commands don't support the . or o notation, though.

mcrf crfD,crfS

This command takes crfS in **Condition Field Addressing Mode**

and stores it into another condition field crfD.

There are also some extended commands:

crset crbD

crclr crbD

Sets/Clears a specific condition Bit.

crnot crbD,crbA

Does a negation on crbA and puts the negated crbA to crbD (crbA itself remains unchanged).

crmove crbD,crbA

copies condition Bit crbA to condition Bit crbD.

## 1.83 MagicSNs PowerPC ASM Coding Tutorial

Coding Subroutines

As you can read in the section about branches, on the PowerPC you jump into a subroutine, using the bl or bccl commands, and back, using the blr or bcclr commands. The bl/bccl stores the back-address on the link-register, the blr or bcclr jumps back to the EA found in the link-register.

The problem is: There is only one link-register. To enable to jump to a subfunction from a subfunction you need a so-called "Stack-Frame". Luckily StormPowerASM offers some macros to create such a "Stack-Frame" automatically. The Stack-Frame also offers you space for local variables and for saving non-scratch registers. You can leave out the stack-frame, if you exactly know how to do this (so to make things easier: Better don't leave it out !!!)

At this place i want to talk about the calling-conventions on the PPC, called the "PowerOpen standard" (this is not only used on the Amiga !!! The PowerMac also uses it, for example).

Parameters are generally given in r3 to r10. If a library base is needed, this is given in r3. r2 contains a pointer to the SmallData base, which is mainly needed, if you code a library function. r1 is the global stackpointer, r13 the local stackpointer. r3/r4 gives results back (r4 will only be used, if 64 Bit Values will be returned). If you change r13-r31 or f14-f31, you have to restore their old values before the function is finished. The only "special registers" you are allowed to overwrite without restoring the value before you leave the function are LR, CTR, XER, FPSCR and fields 0,1,5,6 and 7 of CR (the LR will be usually restored, though, the same to the CR).

Floating point parameters can be given in f1-f13, and FP return values can be given in f1-f4. Normally only f1 will be used. Using f1 and f2 you are able to return a 128 Bit or complex number, using f1-f4 you are able to return a complex 128 Bit number.

It has to be said clearly, that you HAVE to create a stackframe for your subfunction,

if you want to call it from another subfunction, even if you do not use the stack.

You could theoretically use the stackframe of the subfunction that called the new subfunction, though. But to be on the good side, better create a stackframe for ALL subfunctions.

in stormc:include/powerpc/ppcmacros.i exist the following macros for stackframes:

```
prolog [stacksize,[localSP,["TOC"]]]
```

This creates a stackframe with size stacksize (if you don't give the size, `__LOCALSIZE` will be used). If localSP is not present, the register `__LOCAL` (r13 as default) will be used as the local stackpointer. TOC determines, if r2 will be stored/restored, also.

This is needed for library functions.

```
epilog ["TOC"]
```

This removes the stackframe again, and also jumps back from the subroutine. TOC is needed for library functions.

```
setlocal stacksize,localSP
```

With this macro, you can set the defaults for `__LOCALSIZE` and `__LOCAL`.

```
push Rn
```

Stores Rn to the local stack. You get Rn back using `pop Rn`.

```
pushcr
```

Stores the CR to the stack. You get it back using `popcr`.

```
pushctr
```

Stores the CTR to the stack. You get it back using `popctr`.

```
pushf Fn
```

Stores Fn (a FPU register) on the stack. You get it back using `pop Fn`.

```
pushfpr ...
```

Stores several FPU registers on the stack. The syntax is like with the `movem`-command of the 68k CPU. The reverse is `popfpr`(without parameters).

```
pushgpr ...
```

Stores several registers on the stack. The syntax is like with the `movem`-command of the 68k CPU. The reverse is `popgpr`(without parameters).

```
pushlr
```

stores the link register on the stack. The reverse is `poplr`.

Note: For the `pop`-commands you can also leave out the parameter. It will automatically find out, which registers to pop again. But you also can use the parameters, if you want.

All push/pop commands only work, if you have a valid stackframe. They can also be used to store non-scratch-registers (r13-r31, f14-f31).

To create local variables, there also exist:

```
lnk Rn,offset
```

stores the stackpointer to the local stack, gives the new value of the

stackpointer to Rn and adds Offset to the stackpointer. The offset should always be negative. Rn points to the top of this memory.

unlnk Rn

Give the memory allocated by lnk back to the system.

Example:

```
Include "stormc:include/powerpc/ppcmacros.i"
```

```
test:
```

```
prolog 4096
```

```
lnk r10,-20
```

```
pushgpr r14-r17/r28
```

```
pushfpr f15-f20
```

```
...
```

```
popfpr f15-f20
```

```
popgpr r14-r17/r28
```

```
unlnk r10
```

```
epilog
```

This is quite a big example, for small subfunction

```
test:
```

```
prolog
```

```
...
```

```
epilog
```

will be enough.

## 1.84 MagicSNs PowerPC ASM Coding Tutorial

Coding Library Functions

Principally a PPC library function looks like following:

```
libtest:
```

```
prolog 256,,TOC
```

```
pushgpr <register list>
```

```
lwz r2,disp(r3)
```

```
...
```

```
popgpr
```

```
epilog TOC
```

Often it also looks like:

```
libtest:
```

```
prolog 256,,TOC
```

```
...
```

```
epilog TOC
```

---

I strongly recommend coding a library in C, and linking the ASM functions together, instead of writing the whole library in ASM. This is much easier. Of course it is possible to write the whole library in ASM, but this is much more complicated, and needs 68k and PPC ASM, then, as the Initialization still has to be done in 68k (the Open-function of the library has to load powerpc.library, before any PPC Code can be done). I won't go into detail, here. Coding of PPC Native Libraries in 100% ASM is documented in the StormPowerASM docs, for those, whole still want to do it this way.

Of course, Libraries can also be done Mixed Binary. If you use StormC, this is quite easy.

- \* Library Core in C (using StormC)
- \* PPC functions in PPC Assembler (using StormPowerASM)
- \* 68k functions in 68k Assembler (using PhxAss)

Then link everything together...

BTW: PPC-Library-Functions get their Library-Bases in r3 after the call !!!