GNUPRO® TOOLKIT

User's Guide for Altera Nios™

June 2000
Version 1.0

Copyright © 2000 Red Hat®, Inc. All rights reserved.

Red Hat®, the Red Hat Shadow Man logo, GNUPro®, and the GNUPro® logo are all registered trademarks of Red Hat, Inc.

HP-UX® is a registered trademark of Hewlett-Packard® Company.

Solaris™ is a trademark of Sun® Microsystems, Inc.

Windows® is a registered trademark of Microsoft® Corporation, Inc.

UNIX® is a registered trademark of The Open Group.

All other brand and product names, trademarks, and copyrights are the property of their respective owners.

No part of this document may be reproduced in any form or by any means without the prior express written consent of Red Hat, Inc.

No part of this document may be changed an/or modified without the prior express written consent of Red Hat, Inc.

GNUPro Warranty

The GNUPro Toolkit is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. This version of GNUPro Toolkit is supported for customers of Red Hat.

For non-customers, GNUPro Toolkit software has NO WARRANTY.

Because this software is licensed free of charge, there are no warranties for it, to the extent permitted by applicable law. Except when otherwise stated in writing, the copyright holders and/or other parties provide the software “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event, unless required by applicable law or agreed to in writing, will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

How to Contact Red Hat

Red Hat Corporate Headquarters
2600 Meridian Parkway
Durham, NC 27713 USA
Telephone (toll free): +1 888 REDHAT 1
Telephone (main line): +1 919 547 0012
Telephone (FAX line): +1 919 547 0024
Website: `http://www.redhat.com/`

Part #: 300-400-1010099-1.0

Contents

How to Contact Red Hat iii

Introduction 1

Tool Naming Conventions 1
Windows Environment Settings 2
Case Sensitivity 3
Toolkit Features 3
Processor Version 3
Targets Supported 3
Hosts Supported 4
Object File Format 4

Tutorials 5

Windows NT Operating System 7
Create Source Code 7
Compile, Assemble and Link from Source Code 7
Run the Executable on the Stand-alone Simulator 8
Run Under the Debugger 8
Debug with the Built-in Simulator 9
Assembler Listing from Source Code 11
Solaris 2.5.1 Operating System 12
Create Source Code 12
Compile, Assemble and Link from Source Code 12

Run the Executable on the Stand-alone Simulator 13
Run Under the Debugger...13
Debug with the Built-in Simulator 14
Assembler Listing from Source Code 16

Reference...17

Compiler 18
Nios Command Line Options...18
Preprocessor Symbols 18
Nios Attributes...18
Limitations...18
ABI Summary 18
Data Type Sizes and Alignments...19
Floating-point 19
Registers 19
Leaf Procedures..20
Stack Frame 20
Argument Passing...23
Return Values...23
Assembler 24
Nios Command Line Options...24
Syntax 24
Special Characters 24
Register Names..24
Addressing Modes 25
@h Modifier 25
Pseudo-ops..26
Data Alignment 27
Condition Code Specifiers...27
Floating-point 28
Opcodes 29
Linker..29
Nios Command Line Options...29
Linker Script...29
Debugger...30
Nios Command Line Options...30
Simulator...31

Bibliography 33

Introduction

The GNUPro® Toolkit from Red Hat is a complete solution for C and C++ development for the Altera Nios™. The tools include the compiler, interactive debugger and utilities libraries. The User's Guide for Altera Nios consists of the following sections:

- Introduction
  An introduction to the features of the GNUPro Toolkit.
- Reference
  Nios-specific features of the main GNUPro Tools.
- Bibliography

Tool Naming Conventions

Cross-development tools in the Red Hat GNUPro Toolkit normally have names that reflect the target processor and the object file format that is output by the tools (for example, ELF). This makes it possible to install more than one set of tools in the same binary directory, including both native and cross-development tools.

The complete tool name is a three-part hyphenated string. The first part indicates the

processor family and the mode of operation (`nios`). The second part indicates the file format output by the tool (`elf`). The third part is the generic tool name (`gcc`). For example, the GCC ELF compiler for the Altera Nios is `nios-elf-gcc`.

The binaries for a Windows 95/NT hosted toolchain are installed with an `.exe` suffix. However, the `.exe` suffix does not need to be specified when running the executable.

The Nios package includes the following supported tools:

Tool Description	***Tool Name***
GCC compiler	`nios-elf-gcc`
C++ compiler	`nios-elf-g++`
GAS assembler	`nios-elf-as`
GLD linker	`nios-elf-ld`
Standalone simulator	`nios-elf-run`

Binary Utilities	`nios-elf-ar`
	`nios-elf-dlltool`
	`nios-elf-nm`
	`nios-elf-objcopy`
	`nios-elf-objdump`
	`nios-elf-ranlib`
	`nios-elf-readelf`
	`nios-elf-size`
	`nios-elf-strings`
	`nios-elf-strip`

GDB debugger	`nios-elf-gdb`

Windows Environment Settings

For the Windows 95/NT toolchain the libraries are installed in different locations. Therefore, the Windows 95/NT hosted toolchain requires the following environmental settings to function properly. Assume the release is installed in `C:\REDHAT`.

```
SET PROOT=C:\redhat\nios-yymmdd
SET PATH=%PROOT%\H-i686-cygwin32\BIN;%PATH%
SET INFOPATH=%PROOT%\info
REM Set TMPDIR to point to a ramdisk if you have one
SET TMPDIR=%PROOT%
```

NOTE: The variable *yymmdd* indicates the release date printed on the CD-ROM.

Case Sensitivity

The following strings are case sensitive under UNIX and Windows 95/NT:

- command line options
- assembler labels
- linker script commands
- section names
- file names within makefiles
- file names are case sensitive under UNIX

The following strings are not case sensitive under UNIX or Windows 95/NT:

- GDB commands
- assembler instructions and register names

Case sensitivity for Windows 95/NT is dependent on system configuration. By default, file names under Windows 95/NT are not case sensitive.

It is important to remember that the GNUPro Toolkit is case sensitive. Therefore, enter all commands and options exactly as indicated in this document.

Toolkit Features

The following describes Altera Nios-specific features of the GNUPro Toolkit.

Processor Version

Altera Nios

Targets Supported

GNUPro Instruction Set Simulator

Hosts Supported

CPU	***Operating System***	***Vendor***
PA-RISC	HPUX-10.20 and HPUX-11.0	Hewlett-Packard
SPARC	Solaris 2.5.1 and Solaris 2.6	Sun
x86	Windows NT	Microsoft

Object File Format

The Nios tools support the ELF object file format. Refer to Chapter 4, ***System V Application Binary Interface*** (Prentice Hall, 1990.). Use `ld` (refer to *Using LD* in ***GNUPro Utilities***) or `objcopy` (refer to *The GNU Binary Utilities* in ***GNUPro Utilities***) to produce S-records.

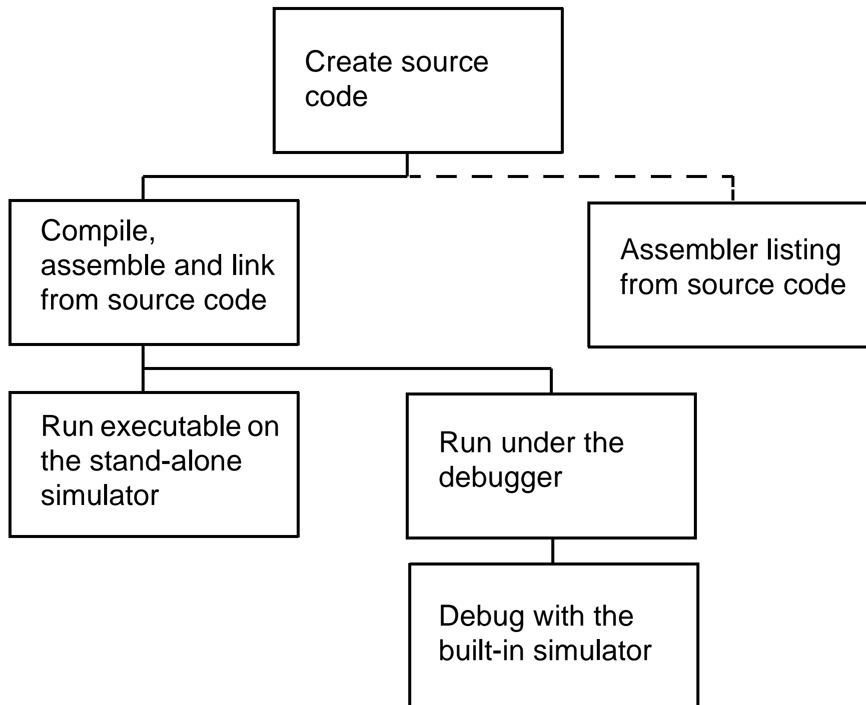
1

Tutorials

This section gives examples of how to use the main utilities. For more detail, refer to the individual utility manuals.

It is important to remember that the GNUPro Toolkit is case sensitive on all operating systems. Therefore, enter all commands and options exactly as indicated in this document.

The following chart outlines the sequence of steps in the tutorial. The assembler listing from source code is optional.



Windows NT Operating System

The following examples for the Windows NT operating system were created using GDB (GNUPro Debugger) in command-line mode. They may also be reproduced using the command prompt in the **Console Window** of Red Hat Insight (the GUI interface to the GNUPro Debugger).

Create Source Code

Create the following sample source code and save it as `hello.c`. Use this program to verify correct installation.

```
#include <stdio.h>

int a, c;

void foo(int b)
{
  c = a + b;
  printf("%d + %d = %d\n", a, b, c);
}

int main()
{
  int b;

  a = 3;
  b = 4;
  printf("Hello, world!\n");
  foo(b);
  return 0;
}
```

Compile, Assemble and Link from Source Code

Throughout these examples, screen samples are shown with a gray background. Code input is shown in plain monofont. Code output is shown in bold monofont. For Windows NT the command prompt is shown as **`C:\>`**.

To compile this example to run on the simulator:

```
C:\> nios-elf-gcc -Tsim.ld -g hello.c -o hello.exe
```

To link an executable for the simulator, the correct linker script must be specified with the `-Tsim.ld` option.

The `-g` option generates debugging information and the `-o` option specifies the name of the executable to be produced. Other useful options include `-O` for standard optimization, and `-O2` for extensive optimization. When no optimization option is

specified GCC will not optimize. Refer to “GNU CC Command Options” in *Using GNU CC* in ***GNUPro Compiler Tools*** for a complete list of available options.

Run the Executable on the Stand-alone Simulator

To run this program on the stand-alone simulator, enter:

```
C:\> nios-elf-run hello.exe
hello world!
3 + 4 = 7
C:\>
```

The simulator executes the program, and returns when the program exits.

Run Under the Debugger

GDB can debug programs by using the built-in simulator (this does not require access to any hardware). To start GDB enter the command:

```
nios-elf-gdb -nw hello.exe
```

After the initial copyright and configuration information GDB returns its own prompt: **(gdb)**.

```
C:\> nios-elf-gdb -nw hello.exe
GNU gdb 4.17-nios-990519 Copyright 1999 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions. There is absolutely no warranty for
GDB.  Type "show warranty" for details. This GDB was configured as
"--host=i686-cygwin --target=nios-elf"...

(gdb)
```

In our examples, the `-nw` option was used to select the command line interface to GDB (the Red Hat Insight interface is the default). The `-nw` is useful for making transcripts such as the one above. The `-nw` option is also useful when you wish to report a bug in GDB, because a sequence of commands is simpler to reproduce.

To exit GDB, enter the `quit` command at the **(gdb)** prompt.

```
gdb) quit
C:\>
```

Debug with the Built-in Simulator

The following is a sample debugging session using the `target sim` command to specify the GNUPro Instruction Set Simulator as the target:

```
C:\> nios-elf-gdb -nw hello.exe
GNU gdb 4.17-nios-990519 Copyright 1999 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions. There is absolutely no warranty for
GDB.  Type "show warranty" for details. This GDB was configured as
"--host=i686-cygwin --target=nios-elf"...

(gdb) target sim
Connected to the simulator.

(gdb) load
Loading section .init, size 0x10 lma 0x0
Loading section .text, size 0xad6e lma 0x10
Loading section .fini, size 0x8 lma 0xad7e
Loading section .rodata, size 0x372 lma 0xad88
Loading section .data, size 0x3d6 lma 0xb0fc
Loading section .ctors, size 0x4 lma 0xb4d2
Loading section .dtors, size 0x4 lma 0xb4d6
Loading section .eh_frame, size 0x1054 lma 0xff04
Start address 0x10
Transfer rate: 403792 bits in <1 sec.

(gdb) break main
Breakpoint 1 at 0x132: file hello.c, line 15.

(gdb) run
Starting program: C:\hello.exe
Breakpoint 1, main () at hello.c:15
15        a = 3;

(gdb) print a
$1 = 0

(gdb) step
16        b = 4;

(gdb) print a
$2 = 3

(gdb) list
11      int main()
12      {
```

```
13        int b;
14
15        a = 3;
16        b = 4;
17        printf("Hello, world!\n");
18        foo(b);
19        return 0;
20      }

(gdb) list foo
1       #include <stdio.h>
2
3       int a, c;
4
5       void foo(int b)
6       {
7         c = a + b;
8         printf("%d + %d = %d\n", a, b, c);
9       }
10

(gdb) break 7
Breakpoint 2 at 0xf4: file hello.c, line 7.

(gdb) continue
Continuing.
Hello, world!
Breakpoint 2, foo (b=4) at hello.c:7
7         c = a + b;

(gdb) step
8         printf("%d + %d = %d\n", a, b, c);

(gdb) print c
$3 = 7

(gdb) next
3 + 4 = 7
9       }

(gdb) backtrace
#0  foo (b=4) at hello.c:9
#1  0x15c in main () at hello.c:18

(gdb) quit
The program is running.  Quit anyway (and kill it)? (y or n) y
C:\>
```

Assembler Listing from Source Code

The following command produces an assembler listing:

```
nios-elf-gcc -g -O2 -Wa,-al -c hello.c
```

The compiler debugging option `-g` gives the assembler the necessary debugging information. The `-O2` option produces optimized code output. The `-Wa` option tells the compiler to pass the text immediately following the comma as a command line to the assembler. The assembler option `-al` requests an assembler listing. The `-c` option tells GCC to compile or assemble the source files, but not to link. Here is a partial excerpt of the output.

```
57 001d 00          .text
58                  .p2align 1
59                  .globl main
60                  .type      main,@function
61             main:
62             .LFB2:
63             .LM5:
64
65                  ; start prologue
66 0024 2E78        save %sp, -46
67             .LCFI1:
68                  ; end prologue
69             .LM6:
70
71             .LBB2:
72             .LM7:
73
74 0026 0098        pfx  %hi(a)
75 0028 1034        movi %l0, %lo(a)
76 002a 6134        movi %g1, 3
77 002c 01A0        stp  [%l0, 0], %g1
78             .LM8:
79
80 002e 0098        pfx  %hi(.LC1)
81 0030 0834        movi %o0, %lo(.LC1)
82 0032 0098        pfx  %hi(printf@h)
83 0034 0134        movi %g1, %lo(printf@h)
84 0036 E17F        call %g1
85 0038 0030        nop
86             .LM9:
87
88 003a 8834        movi %o0, 4
89 003c 0098        pfx  %hi(foo@h)
90 003e 0134        movi %g1, %lo(foo@h)
91 0040 E17F        call %g1
92 0042 0030        nop
```

Solaris 2.5.1 Operating System

The following examples for the Solaris 2.5.1 operating system were created using GDB (GNUPro Debugger) in command-line mode. They may also be reproduced using the command prompt in the **Console Window** of Red Hat Insight (the GUI interface to the GNUPro Debugger).

Create Source Code

Create the following sample source code and save it as `hello.c`. Use this program to verify correct installation.

```
#include <stdio.h>

int a, c;

void foo(int b)
{
  c = a + b;
  printf("%d + %d = %d\n", a, b, c);
}

int main()
{
  int b;

  a = 3;
  b = 4;
  printf("Hello, world!\n");
  foo(b);
  return 0;
}
```

Compile, Assemble and Link from Source Code

Throughout these examples, screen samples are shown with a gray background. Code input is shown in plain monofont. Code output is shown in bold monofont. For Solaris 2.5.1, the command prompt is shown as `%`.

To compile this example to run on the simulator:

```
% nios-elf-gcc -Tsim.ld -g hello.c -o hello
```

To link an executable for the simulator, the correct linker script must be specified with the `-Tsim.ld` option.

The `-g` option generates debugging information and the `-o` option specifies the name of the executable to be produced. Other useful options include `-O` for standard optimization, and `-O2` for extensive optimization. When no optimization option is

specified GCC will not optimize. Refer to “GNU CC Command Options” in *Using GNU CC* in ***GNUPro Compiler Tools*** for a complete list of available options.

Run the Executable on the Stand-alone Simulator

To run this program on the stand-alone simulator, enter:

```
% nios-elf-run hello
hello world!
3 + 4 = 7
%
```

The simulator executes the program, and returns when the program exits.

Run Under the Debugger

GDB can be used to debug executables using the GNUPro Instruction Set Simulator. To start GDB enter the command:

```
nios-elf-gdb -nw hello
```

After the initial copyright and configuration information GDB returns its own prompt: **`(gdb)`**.

```
% nios-elf-gdb -nw hello
GNU gdb 4.17-nios-990519 Copyright 1999 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions. There is absolutely no warranty for
GDB.  Type "show warranty" for details. This GDB was configured as
"--host=sparc-sun-solaris2.5.1 --target=nios-elf"...

(gdb)
```

In our examples, the `-nw` option was used to select the command line interface to GDB (the Red Hat Insight interface is the default), which is useful for making transcripts such as the one above. The `-nw` option is also useful when you wish to report a bug in GDB, because a sequence of commands is simpler to reproduce.

To exit GDB, enter the `quit` command at the **`(gdb)`** prompt.

```
gdb) quit
%
```

Debug with the Built-in Simulator

The following is a sample debugging session using the `target sim` command to specify the GNUPro Instruction Set Simulator as the target:

```
% nios-elf-gdb -nw hello
GNU gdb 4.17-nios-990519 Copyright 1999 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions. There is absolutely no warranty for
GDB.  Type "show warranty" for details. This GDB was configured as
"--host=sparc-sun-solaris2.5.1 --target=nios-elf"...

(gdb) target sim
Connected to the simulator.

(gdb) load
Loading section .init, size 0x10 lma 0x0
Loading section .text, size 0xad6e lma 0x10
Loading section .fini, size 0x8 lma 0xad7e
Loading section .rodata, size 0x372 lma 0xad88
Loading section .data, size 0x3d6 lma 0xb0fc
Loading section .ctors, size 0x4 lma 0xb4d2
Loading section .dtors, size 0x4 lma 0xb4d6
Loading section .eh_frame, size 0x1054 lma 0xff04
Start address 0x10
Transfer rate: 403792 bits in <1 sec.

(gdb) break main
Breakpoint 1 at 0x132: file hello.c, line 15.

(gdb) run
Starting program: /home/nios/hello
Breakpoint 1, main () at hello.c:15
15          a = 3;

(gdb) print a
$1 = 0

(gdb) step
16          b = 4;

(gdb) print a
$2 = 3

(gdb) list
11        int main()
12        {
```

```
13        int b;
14
15        a = 3;
16        b = 4;
17        printf("Hello, world!\n");
18        foo(b);
19        return 0;
20      }

(gdb) list foo
1       #include <stdio.h>
2
3       int a, c;
4
5       void foo(int b)
6       {
7         c = a + b;
8         printf("%d + %d = %d\n", a, b, c);
9       }
10

(gdb) break 7
Breakpoint 2 at 0xf4: file hello.c, line 7.

(gdb) continue
Continuing.
Hello, world!
Breakpoint 2, foo (b=4) at hello.c:7
7         c = a + b;

(gdb) step
8         printf("%d + %d = %d\n", a, b, c);

(gdb) print c
$3 = 7

(gdb) next
3 + 4 = 7
9       }

(gdb) backtrace
#0  foo (b=4) at hello.c:9
#1  0x15c in main () at hello.c:18

(gdb) quit
The program is running.  Quit anyway (and kill it)? (y or n) y
%
```

Assembler Listing from Source Code

The following command produces an assembler listing:

```
nios-elf-gcc -g -O2 -Wa,-al -c hello.c
```

The compiler debugging option `-g` gives the assembler the necessary debugging information. The `-O2` option produces optimized code output. The `-Wa` option tells the compiler to pass the text immediately following the comma as a command line to the assembler. The assembler option `-al` requests an assembler listing. The `-c` option tells GCC to compile or assemble the source files, but not to link. Here is a partial excerpt of the output.

```
57 001d 00          .text
58                  .p2align 1
59                  .globl main
60                  .type      main,@function
61             main:
62             .LFB2:
63             .LM5:
64
65                  ; start prologue
66 0024 2E78        save %sp, -46
67             .LCFI1:
68                  ; end prologue
69             .LM6:
70
71             .LBB2:
72             .LM7:
73
74 0026 0098        pfx  %hi(a)
75 0028 1034        movi %l0, %lo(a)
76 002a 6134        movi %g1, 3
77 002c 01A0        stp  [%l0, 0], %g1
78             .LM8:
79
80 002e 0098        pfx  %hi(.LC1)
81 0030 0834        movi %o0, %lo(.LC1)
82 0032 0098        pfx  %hi(printf@h)
83 0034 0134        movi %g1, %lo(printf@h)
84 0036 E17F        call %g1
85 0038 0030        nop
86             .LM9:
87
88 003a 8834        movi %o0, 4
89 003c 0098        pfx  %hi(foo@h)
90 003e 0134        movi %g1, %lo(foo@h)
91 0040 E17F        call %g1
92 0042 0030        nop
```

2

Reference

This section describes the ABI and Nios-specific attributes of the main GNUPro tools.

- Compiler
- ABI Summary
- Assembler
- Linker
- Debugger
- Simulator

Compiler

This section describes Nios-specific features of the GNUPro Compiler.

Nios Command Line Options

For a list of available generic compiler options, refer to “GNU CC Command Options” in *Using GNU CC* in ***GNUPro Compiler Tools***. In addition, the following Nios-specific command line options are supported:

`-m16`
: Generate code for the Nios-16 processor (default).

`-m32`
: Generate code for the Nios-32 processor.

Preprocessor Symbols

The compiler supports the following preprocessor symbols:

`__nios__`
: Is always defined.

`__nios16__`
: Defined by default or if `-m16` is used.

`__nios32__`
: Defined if `-m32` is used.

Nios Attributes

There are no Nios-specific attributes. See “Declaring Attributes of Functions” and “Specifying Attributes of Variables” in “Extensions to the C Language Family” in Using GNU CC in ***GNUPro Compiler Tools*** for more information.

Limitations

The unary operator `&&`, a GNU extension which allows labels to be treated as values, is not supported for the Nios processor. This limitation stems from the fact that code pointers and data pointers are treated differently by the processor.

ABI Summary

The Altera Nios toolchain supports the Altera Nios ABI.

Data Type Sizes and Alignments

The following table shows the size and alignment for all data types:

Type	*Size (16-bit)*	*Size (32-bit)*
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	2 bytes	4 bytes
`long`	4 bytes	4 bytes
`long long`	8 bytes	8 bytes
`float`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`long double`	8 bytes	8 bytes

- For the Nios-16 processor, word and pointer size is two bytes.
- For the Nios-32 processor, word and pointer size is four bytes.
- Functions are aligned to two byte boundaries.
- Data elements are aligned to their natural boundary up to a maximum of 32 bits.
- Structures, unions and strings are always word-aligned.
- Bitfields inside structures are always word-aligned.
- All alignments are strict.

Floating-point

All floating-point values are emulated using IEEE floating-point conventions.

Registers

Out Registers		*Description*
`%o0`	`%r8`	Return value, call-clobbered
`%o1`	`%r9`	Call-clobbered
`%o2`	`%r10`	Call-clobbered
`%o3`	`%r11`	Call-clobbered
`%o4`	`%r12`	Call-clobbered
`%o5`	`%r13`	Call-clobbered
`%o6`	`%r14`	Stack pointer, call-preserved
`%o7`	`%r15`	Link register, call-preserved

Local Registers		*Description*
`%l0`	`%r16`	Call-preserved
`%l1`	`%r17`	Call-preserved
`%l2`	`%r18`	Call-preserved

%l3	%r19	Call-preserved
%l4	%r20	Call-preserved
%l5	%r21	Call-preserved
%l6	%r22	Call-preserved
%l7	%r23	Call-preserved

Global Registers		***Description***
%g0	%r0	Call-clobbered
%g1	%r1	Call-clobbered
%g2	%r2	Static chain, call-clobbered
%g3	%r3	Call-clobbered
%g4	%r4	Call-clobbered
%g5	%r5	Call-clobbered
%g6	%r6	Call-clobbered
%g7	%r7	Call-clobbered

In Registers		***Description***
%i0	%r24	Argument register 1, call-preserved
%i1	%r25	Argument register 2, call-preserved
%i2	%r26	Argument register 3, call-preserved
%i3	%r27	Argument register 4, call-preserved
%i4	%r28	Argument register 5, call-preserved
%i5	%r29	Argument register 6, call-preserved
%i6	%r30	Frame and argument pointer, call-preserved
%i7	%r31	Return address, call-preserved

Leaf Procedures

Leaf procedures are those procedures that do not call any other procedures. Some leaf procedures can be transformed to use their caller's register window and stack frame. This saves execution time and memory, and can be done when the candidate leaf procedure meets the following conditions:

- The procedure contains no references to `%sp`, except in its `save` instruction.
- It contains no references to `%fp`.
- It refers to no more than 15 of the 32 registers, including the return address register, `%o7`.

When optimized, a leaf procedure will use its caller's stack frame and registers, and it will only use registers `%o0` through `%o5`, `%o7`, and `%g0` through `%g7`.

Stack Frame

This section describes the Altera Nios stack frame.

The stack grows downwards from high addresses to low addresses.

A leaf function is not required to allocate a stack frame if one is not needed.

The ABI requires a frame pointer to be allocated if any stack is allocated. In other words, a leaf function that uses no stack does not allocate a frame pointer, but a leaf function that uses stack or a non-leaf function requires a frame pointer.

Normally a stack frame, is allocated for each procedure. Under certain conditions, optimization may enable a leaf procedure to use its caller's stack frame instead of one of its own. In that case, the procedure allocates no space of its own for a stack frame. The following description of the memory stack applies to all procedures, except leaf procedures which have been optimized in this way.

At compile time, sixteen words are always allocated in every procedure's stack frame, always starting at `%sp`, for saving the procedure's "in" and "local" registers, should a register window overflow occur.

At compile time, the following are allocated in the stack frames of non-leaf procedures:

- One word, for passing a hidden (implicit) parameter. This is used when the caller is expecting the callee to return a data aggregate by value. The hidden word contains the address of stack space allocated (if any) by the caller for that purpose.
- Six words, into which the callee may store parameters that must be addressable

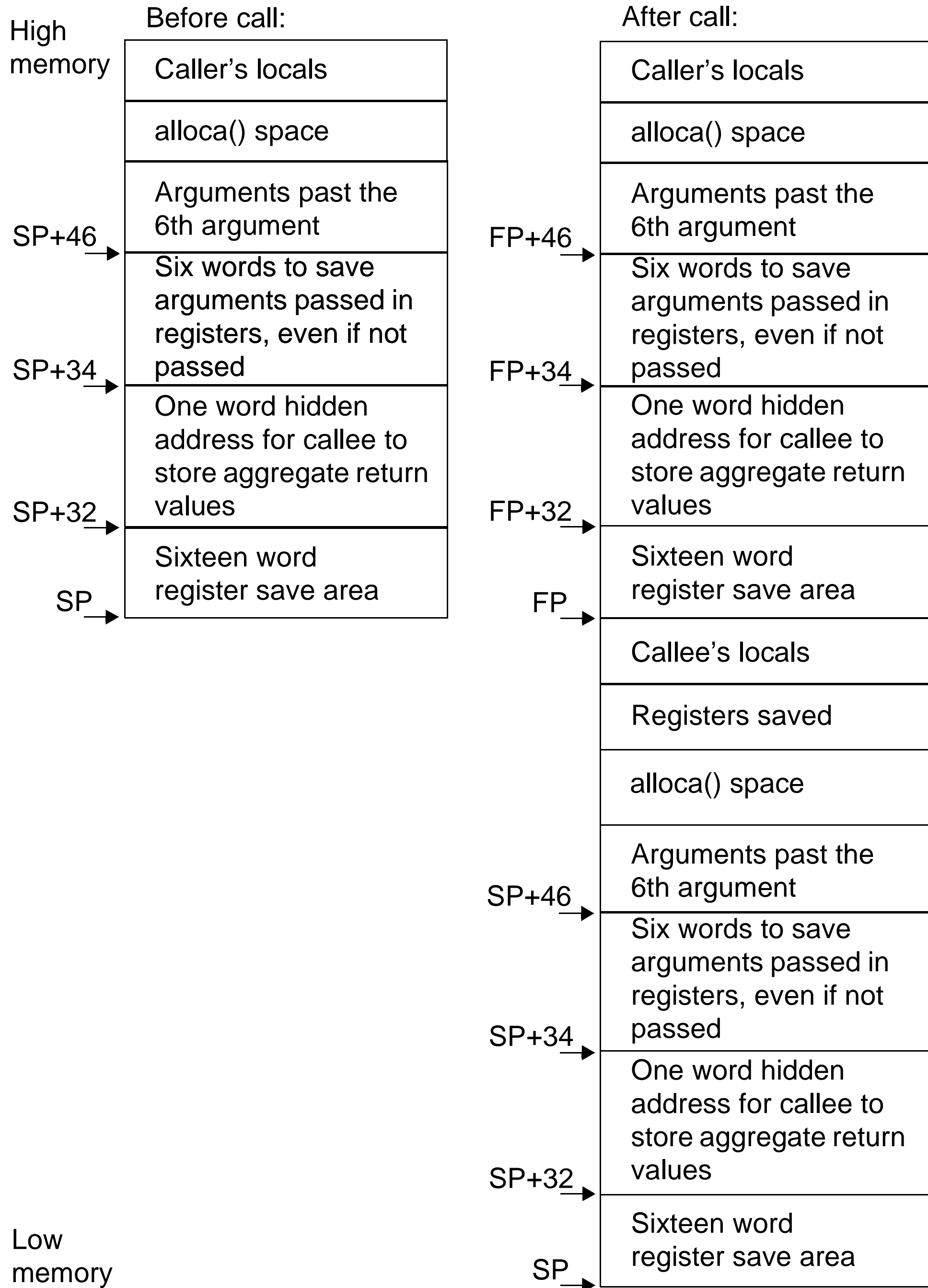
At compile time space is allocated as needed in the stack frame for the following:

- Outgoing parameters beyond the sixth
- All automatic arrays, automatic data aggregates, automatic scalars which must be addressable, and automatic scalars for which there is no room in registers
- Compiler-generated temporary values (typically when there are too many for the compiler to keep them all in registers)

At runtime, space can be allocated dynamically in the stack frame for memory allocated using the `alloca()` function of the C library.

Addressable automatic variables on the stack are addressed with negative offsets relative to `%fp`. Dynamically allocated space is addressed with positive offsets from the pointer returned by `alloca()`. Everything else in the stack frame is addressed with positive offsets relative to `%sp`.

The following stack frame diagram is for both functions that take a fixed number or variable number of arguments.



High memory
Before call:
Caller's locals
alloca() space
Arguments past the 6th argument
Six words to save arguments passed in registers, even if not passed
One word hidden address for callee to store aggregate return values
Sixteen word register save area
SP+46
SP+34
SP+32
SP
After call:
Caller's locals
alloca() space
Arguments past the 6th argument
Six words to save arguments passed in registers, even if not passed
One word hidden address for callee to store aggregate return values
Sixteen word register save area
Callee's locals
Registers saved
alloca() space
Arguments past the 6th argument
Six words to save arguments passed in registers, even if not passed
One word hidden address for callee to store aggregate return values
Sixteen word register save area
FP+46
FP+34
FP+32
FP
SP+46
SP+34
SP+32
SP
Low memory

Argument Passing

Function arguments are passed in registers `%o0` through `%o5`, with lower-numbered registers being allocated to earlier arguments. Registers are allocated consecutively, with no gaps. When all six registers have been filled, any remaining arguments are placed in the stack argument area, allocating from lower to higher addresses. Each argument starts on a word boundary.

- Any argument whose address is not needed and it is four words in size or smaller, whether scalar (e.g., integers, floats) or aggregate (e.g., structures, unions) is passed in consecutive registers. Lower-numbered registers hold less significant words of the value.

  For example, on Nios-16, if a function's first argument is a 64-bit `long long`, callers should pass it in `%o0`, `%o1`, `%o2` and `%o3`, with `%o3` carrying the most significant word.

  If a large argument cannot fit entirely in the remaining registers, the caller must split the argument between the remaining registers and the stack. The registers hold the less significant portion of the argument, and the stack holds the more significant portion. As before, lower-numbered registers and lower addresses hold less significant words.
- To pass any argument larger than four words, the caller must copy the argument to a buffer reserved for it in its own local variable area. The caller then passes the address of this buffer as the argument.

If the callee takes a variable number of arguments, it stores all its argument registers in an argument register save area. This area is six words long, just large enough to hold all the argument registers, and allocated just below any arguments received on the stack. Thus, once the registers have been saved, all the function's arguments appear in a contiguous block of memory, starting with the argument register save area. To walk the argument list, the callee needs only advance a pointer from lower to higher addresses.

Return Values

All values that fit in a word are returned in `%o0`. This includes structures and unions.

For values larger than a word, the caller allocates a buffer of the appropriate size in its own local variable area, and passes the address of this buffer to the callee in the hidden address stack slot. The callee must return the buffer's address in register `%o0`.

Assembler

This section describes Nios-specific features of the GNUPro Assembler.

Nios Command Line Options

For a list of available generic assembler options, refer to “Command Line Options” in *Using as* in ***GNUPro Utilities***. The Nios version of the assembler has only one machine dependent option.

`-m16`
: Assemble for the Nios-16 processor (default).

`-m32`
: Assemble for the Nios-32 processor

Syntax

There are no size modifiers for instructions, nor can they be run in parallel. Instructions are handled one at a time and are always 16 bits in size. Object files assembled for different processors cannot be linked together.

Special Characters

The Nios assembler supports the following special characters:

semicolon (`;`) and pound sign (`#`)
: Both characters are line comment characters when used in the zero column. The semicolon may also be used to start a comment anywhere within a line.

percent sign (`%`)
: This character is used as a prefix for register names. For example: `%r3`.

Register Names

You can use the predefined symbols `%r0` through `%r31` to refer to the Nios registers. You can also use `%sp` as an alias for `%r14`. Register names are not case sensitive.

Nios also has predefined symbols for these registers:

Usage	***Symbols***	***Registers***
In	`%i0-%i7`	`%r24-%r31`
Local	`%l0-%l7`	`%r16-%r23`
Out	`%o0-%o7`	`%r8-%r15`
Global	`%g0-%g7`	`%r0-%r7`

These predefined symbols are not recognized by the debugger or the disassembler.

Addressing Modes

The assembler understands the following addressing modes for the Nios. The symbol `Rn` in the following examples refers to any of the specifically numbered registers or register pairs, but not the control registers.

`%Rn`
: Register direct

`[%Rn]`
: Register indirect

`[%Rn, offset]`
: Register indirect with offset

`addr`
: PC relative address

`%hi(addr)`
: hi 11 bits of 16-bit absolute address

`%lo(addr)`
: lo 5 bits of 16-bit absolute address

`%xhi(addr)`
: hi 11 bits of 32-bit absolute address

`%xlo(addr)`
: lo 5 bits of top 16 bits of 32 bit absolute address

`#imm`
: Immediate value

Indirect references are always in square brackets. For example

```
 [%r2]
```

means the storage addressed by `%r2`.

Registers are always prefixed by the percent sign (`%`). Immediate values are generally prefixed by the pound sign (`#`) or nothing at all.

@h Modifier

The `@h` modifier is used as a suffix for a symbol identifier to get the symbol's address in halfwords

For example:

```
        pfx %hi(foo@h)
        movi %r4,%lo(foo@h)
        jmp %r4
        nop
 foo:
```

or

```
        ldc %r3,table
        jmp %r3
table:
        .nword foo@h
```

The `@h` modifier is required because `jmp` and `call` instructions all require addresses that have been shifted to the right by 1 bit. Regular addresses also have to be honored because storage accesses still use them, for example:

```
st [%r3],%r2
```

Thus, you need a way to refer to a symbol's address in either format. The `@h` optional suffix handles this.

`%hi()` gives the top 11 bits of a half-word, and `%lo()` gives the bottom 5 bits of a half-word. `%xhi()` gives the top 11 bits of a full-word and `%xlo()` gives the bottom 5 bits of the top halfword.

For example, if you wanted to load `r5` with `-42`:

```
pfx %hi(-42)
movi %r5,%lo(-42)
```

On a Nios-32 machine you have to load both the top and bottom halfwords so you have:

```
pfx %hi(-42)
movi %r5,%lo(-42)
pfx %xhi(-42)
movhi %r5,%xlo(-42)
```

Pseudo-ops

The pseudo-op `.nword` has been added. This pseudo-op means a native word-size of data. For Nios-16, it is two bytes, for Nios-32 it is four bytes. The native word-size should not be confused with a "word," which is 4-bytes for both platforms.

The following is the pseudo-ops list (in addition to the standard list):

`.word`
: 4 bytes

`.long`
: Same as `.word`

`.half`
: 2 bytes

`.short`
: Same as `.half`

`.nword`
: Same as `.word` for Nios-32, or same as `.half` for Nios-16

Data Alignment

`.align x`

Data is not aligned by default. The `.align` statement is required. The `.align` keyword aligns to 2 raised to the power of *x*, where *x* is the variable specified. For example

```
.align 3
```

aligns to an 8-byte word boundary

Condition Code Specifiers

The following `cc_` codes were added for both the SKPS and IFS instructions. The SKPS instruction “skips” the next instruction if the condition is true. The IFS instruction performs the next instruction “if” the condition is true. The following special variables have been added:

`cc_eq`
Check for equal

`cc_z`
Check for zero

`cc_ne`
Check for not equal

`cc_nz`
Check for not zero

`cc_gt`
Check for greater than

`cc_ge`
Check for greater than or equal

`cc_lt`
Check for less than

`cc_le`
Check for less than or equal

`cc_ls`
Check for less than same (unsigned comparison)

`cc_hi`
Check for higher than (unsigned comparison)

`cc_mi`
Check for negative

`cc_n`
Same as cc_mi

`cc_pl`
: Check for positive

`cc_p`
: Same as `cc_pl`

`cc_cc`
: Check for carry clear

`cc_nc`
: Same as `cc_cc`

`cc_cs`
: Check for carry set

`cc_c`
: Same as `cc_cs`

`cc_vc`
: Check for overflow clear

`cc_nv`
: Same as `cc_vc`

`cc_vs`
: Check for overflow set

`cc_v`
: Same as `cc_vs`

Implementation

SKPS and IFS use opposite condition codes.

To implement a branch not equal to `foo`, enter either:

```
skps cc_eq
br foo
```

or

```
ifs cc_ne
br foo
```

To branch to `foo` on greater than or equal, enter either:

```
ifs cc_ge
br foo
```

or

```
skps cc_lt
br foo
```

A `br` never specifies `@h` (you are branching to `foo`).

Floating-point

Although the Nios has no hardware floating-point, the `.float` and `.double` directives

generate IEEE floating point numbers for compatibility with other development tools.

Opcodes

The assembler implements all the standard Nios opcodes.

Linker

This section describes Nios-specific features of the GNUPro Linker.

Nios Command Line Options

For a list of available generic linker options, refer to “Linker scripts” in *Using ld* in ***GNUPro Utilities***. There are no Nios-specific command line linker options.

Linker Script

The GNU Linker uses a linker script to determine how to process each section in an object file, and how to lay out the executable. The linker script is a declarative program consisting of a number of directives. For instance, the `ENTRY()` directive specifies the symbol in the executable that will be the executable’s entry point.

When building C or C++ executables to run under the simulator, you must specify the simulator linker script `-Tsim.ld`.

Because this linker script refers to the C, simulator-syscall, and the gcc libraries, it is recommended that you link using the `nios-elf-gcc` command and specify `-Tsim.ld`. The `nios-elf-gcc` command sets up the necessary library paths so that the library references are properly resolved. For example:

```
nios-elf-gcc -c -g -m32 foomain.c
nios-elf-gcc -c -g -m32 fooextra.c
nios-elf-gcc -g -m32 -Tsim.ld -0 foo foomain.o fooextra.o
```

There are actually two simulator links scripts: one for `nios16` and the other for `nios32`. The `nios16` platform requires a special linker script because code may reside above the 64K line, whereas data must reside below 64K. The `nios32` script is similar to traditional elf linker scripts, where data such as the heap and stack, reside after the code. When you use `nios-elf-gcc` and specify `-Tsim.ld`, the appropriate `ld` script is used, based on the `-m` compiler option. Using `-m16` (or defaulting) causes the `nios16` version of `sim.ld` to be used. Using `-m32` causes the `nios32` version of `sim.ld` to be used.

The `nios16` linker script is located in `nios-elf/lib/sim.ld`. The `nios32` linker

script is located in `nios-elf/lib/m32/sim.ld`.

Debugger

This section describes Nios-specific features of the GNUPro Debugger. There are three ways for GDB to debug programs for an Nios target.ration of the specific evaluation board.

1. Simulator:
GDB's built-in software simulation of the Nios processor allows the debugging of programs compiled for the Nios without requiring any access to actual hardware. To activate this mode in GDB type `target sim`. Then load the code into the simulator by typing `load` and debug it in the normal fashion.

2. To download through the serial port, use the following GDB commands to load your program:

```
set remotebaud 38400
target remote port
load
```

First the baud rate is set to 38400 with the `set remotebaud` command. Then you must connect GDB to the program using the target remote command. Here *`port`* is the name of the serial port on your host. Then the program is loaded with the `load` command.To continue execution of your program use the GDB `continue` command, NOT the `run` command.

3. To download via ethernet, use the following GDB commands to load your program:

```
target remote device_name:ethernet_port
load
```

First you must connect GDB to the program using the `target remote` command using the above syntax. Then the program is loaded with the `load` command.To continue execution of your program use the GDB `continue` command, NOT the `run` command.

Nios Command Line Options

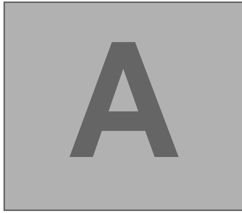
For the available generic debugger options, refer to *Debugging with GDB* in ***GNUPro Debugging Tools***. There are no Nios-specific debugger command line options.

Simulator

The following command can be used to start the simulator

```
nios-elf-run [-t] a.out
```

where `-t` causes a trace of the simulator to be output to `a.out`.



Bibliography

Getting Started with GNUPro Toolkit
(`http://www.redhat.com/apps/support`)

GNUPro Compiler Tools
(`http://www.redhat.com/apps/support`)

GNUPro Debugging Tools
(`http://www.redhat.com/apps/support`)

GNUPro Libraries
(`http://www.redhat.com/apps/support`)

GNUPro Utilities
(`http://www.redhat.com/apps/support`)

GNUPro Tools for Embedded Systems
(`http://www.redhat.com/apps/support`)