Qualcomm® Hexagon™ Application Binary Interface

User Guide

80-N2040-23 Rev. K

December 4, 2018

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Contents

Tables

1 Introduction

This document describes the application binary interface (ABI) for Qualcomm® Hexagon™ processor versions V4, V5x, and V6x.

The Hexagon ABI is based on the System V ABI. This document describes only the Hexagon-specific aspects of the ABI. For information on the complete ABI specification, see the following documents:

- *System V Application Binary Interface, Edition 4.1* (http://www.sco.com/developers/devspecs/gabi41.pdf)
- *Solaris Linker and Libraries Guide, chapter 8 (Thread-Local Storage)* (http://docs.oracle.com/cd/E19253-01/817-1984/)

1.1 Conventions

`Courier` font is used for computer text and code samples, for example, `hexagon_sim_read_pcycles()`.

The following notation is used to define command syntax:

- Square brackets enclose optional items, for example, `[`**`label`**`]`.
- **`Bold`** indicates literal symbols, for example, **`[comment]`**.
- The vertical bar character, `|`, indicates a choice of items.
- Parentheses enclose a choice of items, for example, **`add`**`|`**`del`**`)`.
- An ellipsis, `. . .` , follows items that can appear more than once.

1.2 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 ABI overview

The ABI defines a set of conventions that enables the inter-operation of code written in different languages and at different times. These conventions include:

- Hardware representation of C data types
- Software stack configuration
- Parameter passing
- Return values
- Register usage across function calls
- C++ exception handling
- Operating system interface
- Process initialization
- Program loading
- Object files
- Program headers
- Dynamic linking
- Thread-local storage

Coding examples are provided for some of the basic operations.

3 Data types

This chapter describes how C data types are represented on the Hexagon processor.

3.1 Basic data types

Table 3-1 lists the basic C data types and how they are represented in hardware on the Hexagon processor.

Table 3-1 Basic data type representation

C Type	Size (in bytes)	Hardware Representation
Char	1	Byte
Short	2	Halfword
Int	4	Word
Long	4	Word
Long long	8	Doubleword
Float	4	Word
Double	8	Doubleword
Long double	8	Doubleword
Enum	variable	variable
Pointer	4	Word

By default, the `char` data type is unsigned. If a `char` variable must be signed, declare the variable with the type `signed char`.

By default, enumeration types are assigned the smallest integer type that can store the enumeration values. For example, an `enum` type containing constants in the range 0–255 is stored in memory as a single byte.

NOTE: The command option `-fno-short-enums` causes the C compilers to allocate all enumeration types in 4 bytes.

3.2 Memory alignment

The Hexagon processor requires data types to be properly aligned in memory when they are accessed. An item of size N bytes is aligned when its memory address mod N yields 0. For example, a 4-byte memory access is aligned when the address is an integral multiple of 4.

Table 3-2 C data type alignment requirements

Data type	Alignment requirement
Scalar	Align to size of item (Section 3.1).
Enumeration	Align according to underlying data type.
Array	■ Align array according to data type of its array element. ■ Align array elements according to their data type.
Structure	■ Align structure to alignment of structure member with largest alignment. ■ Align members according to their data type. ■ If necessary, insert pad bytes between members. ■ If necessary, insert pad bytes after last member to make the structure size an integral multiple of the alignment requirement. (This ensures correct pointer arithmetic for pointers to structures.) **NOTE:** Members are allocated in memory in the order they appear in a `struct` definition.
Packed structure	■ Align structure to one-byte boundary. ■ Align members to one-byte boundaries. ■ Pad bytes not inserted between members. **NOTE:** Members are guaranteed only to be one-byte aligned.
Bit field	■ Same size and alignment rules as non-bit field structure members. ■ Allocated from right to left (i.e., least to most significant). ■ Must reside entirely within a storage unit appropriate for their data type. ■ Unsigned by default. To create signed bit fields use the `signed` type specifier.
Union	Align to alignment of member with largest alignment.

Data types are aligned only when they are accessed. Unaligned types are allowed and must be handled properly by the compilers. Problems arise only when dereferencing pointers are cast to data types with stricter alignment requirements than the original type. For example:

```
int g(int *p){
   return *p;
}

int f(void){
   unsigned char a[4];
   a[0] = 0xbeU;
   a[1] = 0xbaU;
   a[2] = 0xfeU;
   a[3] = 0xcaU;
   return g(a);
}
```

In this example, array `a` is guaranteed only to be 1-byte aligned, but function `g()` accesses `a` as an `int`, which (unless it is declared with the `packed` attribute) must be 4-byte aligned. Thus the compiler will generate a `memw` instruction in `g()`.

Alignment attributes

Nonstandard memory alignment requirements must be explicitly specified using the `aligned` attribute. For example:

```
int buffer[N] __attribute__((aligned(8)));
```

NOTE: You can specify any alignment value larger than the one required.

The `packed` attribute can be used to declare that a data type must not be aligned. `packed` and `aligned` can be used together to align a data type to more than 1 byte but less than the type's `natural` alignment. For example:

```
int i __attribute__((packed, aligned(2)));
```

In this example, `i` is 2-byte aligned.

3.2.1 Static allocation

Data types statically allocated in memory are subject to the alignment requirements listed in Section 3.2. No additional requirements exist.

3.2.2 Stack allocation

Data types allocated on the software stack (Chapter 4) are subject to the alignment requirements listed in Section 3.2.

Data allocated on the stack is only guaranteed to be up to 8-byte aligned. To allocate stack data with more than 8-byte alignment, the data address must be manually aligned. Follow these steps:

1. Declare a local `char` array with the following number of elements:

   ```
   (size(data) + alignment(data) - 1)
   ```

2. Cast the address of the array to an integer.
3. Clear the $\log_2$(alignment) low bits of the integer.
4. Cast the resulting value to a pointer to the type to be allocated.

NOTE: Currently, the compilers do not support stack realignment.

3.2.3 Heap allocation

Data types allocated on the heap are subject to the alignment requirements listed in Section 3.2.

The standard allocator functions (`malloc`, `realloc`, `calloc`) return a pointer that is aligned to the largest alignment required by any type in the absence of the aligned attribute. For the Hexagon processor this value is currently 8 bytes.

To allocate an object on the heap with alignment greater than 8 bytes, use the `memalign` function.

NOTE: `memalign` automatically performs the procedure described in Section 3.2.2.

4 Software stack

The software stack is divided into frames. In general, there is one frame for each active function. The exception to this rule is described later. First, we describe the layout of each frame.

The current stack frame is marked by the frame pointer register (R30) and the stack pointer register (R29). The stack grows toward smaller addresses in memory. Stack frames are always aligned on 8-byte boundaries. This enables the alignment of data within each frame.

4.1 Stack sections

Each stack frame contains the following sections, in decreasing memory order:

1. Saved R31, saved R30

   The `allocframe` instruction is used to save the values of R31 (the function return address) and R30 (frame pointer) during the function prologue.

2. Locals/spill area

   This optional area contains enough space for:

   - Any local variables that require stack space
   - Any register spills (including callee saved registers) that are required

   The size of this area is determined at compile time on a per function basis. To minimize the amount of padding between variables, QTI recommends (but does not require) that the data in this section is ordered.

3. Alloca area

   This optional area contains space allocated by `alloca`.

   The size of this area is determined at run time based on the `alloca` calls that are executed. Initially, this size is zero.

4. Outgoing memory arguments

   This optional area is used when the function calls one or more functions whose arguments cannot all be passed in registers.

   The size of this area is determined at compile time on a per function basis. Within the function, the compiler examines all function calls and determines the maximum space required for any call. This scheme makes function calls fast and simple at the expense of some wasted space.

Each section must be aligned to an 8-byte boundary. If necessary, add padding in the high memory address of each section.

Within each section, the normal data alignment rules apply (Section 3.2).

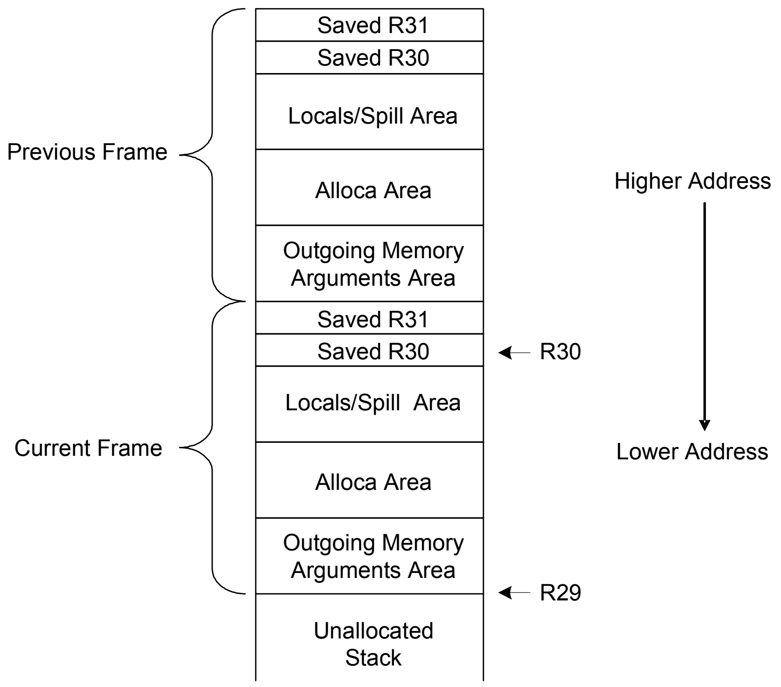

Figure 4-1 Stack in memory

NOTE: For performance reasons, certain functions are allowed to avoid the overhead of allocating and deallocating a frame. A function that does not make any calls is called a *leaf function*. A leaf function that does not require any stack space (i.e., the size of each optional area is zero) cannot allocate a frame.

5 Parameter passing

Parameters are passed to functions in up to six general purpose registers: R0–R5. Any parameters that cannot fit in these registers are passed on the software stack.

The calling function allocates space in the lower portion of its frame for these. Each function allocates the maximum amount of space required by any of its calls in its outgoing memory arguments area.

If the function does not make any calls, or if all of its calls can pass all parameters in registers, then the outgoing memory arguments area of the stack frame is not allocated.

5.1 Fixed argument list function calls

The most common case for function calls is a fixed argument length function. In C, these are functions with:

- A function prototype that contains a fixed number of arguments.
- No function prototype, also known as Kernighan and Ritchie (K&R) style.

A variable argument length function whose prototype is missing at the call site can lead to subtle program errors. This is due to the mismatch between the parameter passing of the caller and callee.

For each fixed argument length call, the parameters are processed from left to right as follows:

- Any parameter (including aggregates) with size up to 64 bits is a candidate to be passed in a register. Parameters up to 32 bits in size are passed in a single register. Larger parameters are passed in a register pair.
- Candidates are assigned to registers R0 through R5 in order.
- Candidates larger than 32 bits are passed in a register pair. If the next available register is an odd-numbered register, that register is skipped (left unused), and the candidate is placed in the next even-odd pair. If the next available register is R5, then the candidate is passed on the stack.
- Candidates larger than 64 bits are passed on the stack. The alignment rules are the same as for static data (Section 3.2).
- Do not attempt to recover skipped parameter registers by filling them with subsequent parameters. However, we will continue using parameter registers after a candidate larger than 64 bits has been placed on the stack.
- Once all the parameter registers have been filled or skipped, all remaining parameters are passed on the stack. The normal alignment rules apply (Section 3.2).

Example 1: scalars

```
extern int foo(short, float, int, double);
foo(i, a, b, c);
```

The parameters are passed as follows:

```
i R0
a R1
b R2
c R5:R4
```

Example 2: arrays and structures

```
extern int bar();
struct {int length, width;} st1;
struct {int a; int bvec[8];} st2;
bar(i, st1, st2);
```

The parameters are passed as follows:

```
i R0
st1 R3:R2
st2.a In memory, at R29+0
st2.bvec In memory, at R29+4
```

NOTE: R29 is the stack pointer (`sp`).

5.2 Variable argument list function calls

Variable argument list functions are a special type of function in C where the number and types of arguments can vary. The best-known example of this is the `printf` function in the C library, where the function's number and type of arguments change, depending on the first argument given: a format string. In general, variable argument functions determine the number and type of their arguments by programming conventions.

For these functions, we pass the named (and typed) parameters in the same manner as for fixed argument list functions. The remainder of the arguments are passed on the stack. The normal alignment rules apply (Section 3.2).

Example:

```
extern int vfoo(int, long long, short, ...);
vfoo(a1, a2, a3, a4, (double)a5, a6);
```

The parameters are passed as follows:

```
a1 R0
a2 R3:R2
a3 R4
a4 In memory, at R29+0
a5 In memory, at R29+8, due to 8 byte alignment
a6 In memory, at R29+16
```

NOTE: R29 is the stack pointer (`sp`).

5.3 Vector register usage for function calls

Vector registers V0 to V15 are used for input parameters. Registers W0 to W7 (V1:V0...) are used for Vector Pair arguments. The remaining vector parameters are on the stack. For example, the Passing Vector, VectorPair, uses v0, v2, v3. The v1 parameter is not used.

Vector registers V0 and V1:V0 are used for return results.

Variadic vector arguments are passed on the stack.

All vector registers remain *caller saves* as opposed to *callee saves*, which does not affect existing assembly or C Intrinsic applications. This includes vector predicate and vector registers.

6 Return values

Functions can return values up to 32 bits in size (including structures) in register R0 and up to 64 bits in size (including structures) in the R1:0 register pair. For return values larger than 64 bits, the caller allocates space in its frame large enough to hold the return value.

The space is aligned according to the normal alignment rules for the returned type (Section 3.2). The caller passes an extra argument at the beginning of the parameter list (in R0) containing an address of the allocated space.

Example:

```
typedef struct {
    int a;
    int b;
    int c;
} S;

S foo(int x)
{
    S s;
    ...
    return s;
}

void bar(int y)
{
    S s = foo(y);
}
```

This code will be automatically converted by the compiler to the following:

```
void foo(S *hidden, int x)
{
    S s;
    ...
    *hidden = s;
}

void bar(int y)
{
    S s;
    foo(&s, y);
}
```

7 Register usage across calls

The registers are divided into two groups, according to how they are used across function calls:

- Caller saved

  At a call site, the caller must assume that the callee changes the values of these registers. Therefore, the caller must save the values of these registers before making the call (if necessary), and restore them after the call.

- Callee saved

  At a call site, the caller can assume that the callee does *not* change the values of these registers. Therefore, the callee must save and restore the values of these registers, if necessary.

Table 7-1 Register usage across calls

Register	Usage	Saved by
R0 - R5	Parameters [a]	–
R6 - R15	Scratch [b]	Caller
R16 - R27	Scratch	Callee
R28	Scratch [b]	Caller
R29 - R31	Stack frames	Callee [c]
P3:0	Processor state	Caller

[a] The callee can change parameter values (Chapter 5).

[b] R14-R15 and R28 are used by the procedure linkage table (Section 14.3).

[c] R29-R31 are saved and restored by allocframe and deallocframe (Chapter 4).

7.1 Outgoing memory arguments

The outgoing-memory-arguments section of the caller's frame is caller-saved. In other words, the caller assumes that the callee will overwrite the values of parameters passed. This applies only to the outgoing memory arguments space used by the current call, and not to the total space allocated by the caller.

8 C++ exception handling

The exception handling model for the Hexagon processor is based on the DWARF2 stack unwinding mechanism.

Registers R0 to R3 are reserved to communicate between exception handling library routines and exception handlers.

The stack location for saved LR (see Figure 4-1) is used to store the address of an exception handler to which the procedure returns.

9 Operating system interface

Processes run in a 32-bit virtual address space. The Hexagon memory management hardware translates virtual addresses into physical addresses.

The memory management system supports the following page sizes (in bytes): 4K, 16K, 64K, 256K, 1M, 4M, 16M.

10 Process initialization

The initial state of a process is established by setting certain Hexagon processor registers to well-defined initial values.

10.1 Special registers

The GP register is set to the starting address of the process's small data area, as referenced by the program symbol, `_SDA_BASE_`.

The UGP register is set to the highest-memory-address-plus-one of the process's thread-local storage area (Chapter 15).

The R28 register is set to the address of a function which the process must call when it terminates. The process is responsible for saving this address so it can be called later on. After saving the address, the process can freely use the R28 register.

NOTE: All other special registers contain unspecified values at process initialization, and operating systems are free to specify their contents.

10.2 General registers

Except for R28 and R29 (also known as SP, which points to the top of the stack), the Hexagon general registers contain unspecified values at process initialization.

NOTE: The contents of stack memory below the top of stack is always undefined.

11 Program loading

When loading an object into memory, the relative positions of the object's several segments must be preserved.

Additionally, absolute objects must reside at the same virtual address that was used to build them.

11.1 Position-independent objects

A position-independent object must be loaded at a virtual address with address alignment equal to the alignment of its largest segment. This ensures the alignment of all the object's segments.

The small data area must be aligned to a 64-byte address boundary. Therefore, position-independent objects that contain a small data area must be loaded at a virtual address aligned to 64 bytes.

NOTE: The alignment requirement for position-independent objects might change in future versions of the Hexagon processor.

12 Object files

Hexagon object files are stored in the ELF file format (short for *Executable and Linkable Format*). This chapter describes object file compatibility and how various ELF elements are defined in Hexagon. It covers the following topics:

- Compatibility
- ELF header
- ELF sections
- Relocation

12.1 Compatibility

Object files for the Hexagon V4, V5x, and V6x processors are backward compatible when executed in User mode.

Kernel-mode object files are not backward compatible due to changes in the system-level instruction set.

12.2 ELF header

The ELF header member `e_machine` is set to the symbolic value `EM_HEXAGON` (which has the decimal numeric value 164).

The member `e_type` can additionally be set to the symbolic value `ET_HEXAGON_IR` (numeric value `0xff00`) to indicate that an object file contains compiler-generated intermediary representation language.

Bits [11:0] of the member `e_flags` indicate the version of the Hexagon processor that the object file was created for. Table 12-1 lists the possible values for this bit field.

Table 12-1 Object processor version flags

Name	Value	Processor Version
EF_HEXAGON_MACH_V4	0x3	Hexagon V4
EF_HEXAGON_MACH_V5	0x4	Hexagon V5
EF_HEXAGON_MACH_V55	0x5	Hexagon V55
EF_HEXAGON_MACH_V60	0x60	Hexagon V60
EF_HEXAGON_MACH_V61	0x61	Hexagon V61
EF_HEXAGON_MACH_V62	0x62	Hexagon V62
EF_HEXAGON_MACH_V65	0x65	Hexagon V65
EF_HEXAGON_MACH_V66	0x66	Hexagon V66
EF_HEXAGON_MACH_V67	0x67	Hexagon V67
EF_HEXAGON_MACH_V67T	0x8067	Hexagon V67 Small Core (V67t)

NOTE: ISA is short for *Instruction Set Architecture*. In this context, it is considered equivalent to *processor version*.

12.3 Sections

In addition to supporting the standard ELF section indexes, Hexagon object files can also include processor-specific section indexes for common symbols stored in the small data area (Section 10.1). These indexes are listed in Table 12-2.

The suffix on the processor-specific section index names indicates the access size (in bytes) of the data in the corresponding section. The one name lacking a suffix is used to specify sections that have an access size other than one of the explicitly-specified values.

Table 12-2 Section indexes for common small data

Name	Value	Description
SHN_HEXAGON_SCOMMON	0xff00	Other access sizes
SHN_HEXAGON_SCOMMON_1	0xff01	Byte-sized access
SHN_HEXAGON_SCOMMON_2	0xff02	Half-word-sized access
SHN_HEXAGON_SCOMMON_4	0xff03	Word-sized access
SHN_HEXAGON_SCOMMON_8	0xff04	Double-word-size access

To mark a section as residing in the small data area, the member, `sh_flags`, can be set to the symbolic value, `SHF_HEXAGON_GPREL` (numeric value `0x10000000`).

12.4 Relocation

Many different types of relocations are performed on the instructions and data stored in a Hexagon object file. The relocation types differ according to the instruction and data fields that are affected, and the kinds of calculations that are applied to the fields to perform the relocation.

12.4.1 Relocation fields

The relocation types are expressed using the data fields defined in Table 12-3:

- Name – Specifies the name of the relocatable field (referenced in Table 12-5).
- Width – Specifies the width (in bits) of the relocatable bit field. Depending on the relocation type, either all or some of these bits are modified by the relocation.
- Effective bits – Specifies how many of the bits in the relocation field are modified.
- Bitmap – Specifies a bit pattern which indicates the bits in the relocation field that are actually modified by the relocation.
- Byte address alignment – Specifies the byte address alignment (1, 2, or 4) that is required for the relocation field.

Table 12-3 Relocation fields

Name	Width	Effective bits	Bitmap	Byte address alignment
Word8	8	8	0xff	1
Word16	16	16	0xffff	2
Word32	32	32	0xffffffff	4
Word32_LO	32	16	0x00c03fff	4
Word32_HL	64 [a]	16 and 16	0x00c03fff 0x00c03fff	4
Word32_GP	32	16	N/A [b]	4
Word32_B7	32	7	0x00001f18	4
Word32_B9	32	9	0x003000fe	4
Word32_B13	32	13	0x00202ffe	4
Word32_B15	32	15	0x00df20fe	4
Word32_B22	32	22	0x01ff3ffe	
Word32_M21	32	21	0x0fff3fe0	4
Word32_M25	32	25	0x0fff3fef	4
Word32_R6	32	6	0x000007e0	4
Word32_U6	32	6	N/A[b]	4
Word32_U16	32	16	N/A[b]	4
Word32_X26	32	26	0x0fff3fff	4

[a] The relocation calculations are applied to two 32-bit words in sequential addresses: first "`(S + A) >> 16`", then "`(S + A)`".

[b] The bitmap varies according to the instruction opcode.

NOTE: Bitmaps are used to specify operands in instruction op-codes that must be relocated. When applying a relocation in these cases, the bits that are not specified in the bitmap must be protected from accidental modification.

12.4.2 Relocation symbols

Each relocation type is defined as a formula which is expressed in terms of the symbols listed in Table 12-4.

Table 12-4 Relocation symbols

Symbol	Description
A	Addend used to compute value of relocatable field.
B	Base address of object loaded into memory.
G	Offset into global offset table (Section 14.2) for the entry of a symbol.
GOT	Address of entry zero in global offset table (Section 14.2).
GP	Value of GP register, typically the base address of the small data area (`_SDA_BASE_`).
L	Offset into procedure linkage table (Section 14.3) for the entry of a symbol.
MB	Base address of all strings consumed by compiler message base optimization (`_MSG_BASE_`).
P	Place address of the field being relocated. **NOTE:** This value is computed using `r_offset` in the relocation entry.
S	Value of the symbol whose index resides in the relocation entry (unless the object is a shared object and the symbol index is `SHN_UNDEF`, in which case this is equivalent to symbol B).
TLS	Thread-pointer-relative offset to a thread-local symbol.
T	Base address of the static thread-local template that contains a thread-local symbol.

12.4.3 Relocation types

The relocation types are applied in a similar manner to relocatable, executable, and shared object files, except where noted otherwise.

Only `Elf32_Rela` relocation entries are used. Therefore, the original content of the instruction and data fields is irrelevant when calculating the relocation.

The relocation types are defined in Table 12-5:

- Name – Specifies the symbolic name of the relocation type.
- Value – Specifies the corresponding numeric identifier for the relocation type.
- Field – Specifies the data field affected by the relocation type (Table 12-3).
- Relocation – Sspecifies the algebraic formula used to perform the relocation (with the formula symbols defined in Table 12-4).
- Result – Specifies the numeric format (signed or unsigned) of the relocated value.
- Action – Specifies an additional operation performed on the relocated value as part of the relocation:
  - Truncate – Truncate the relocated value to fit the relocation field.
  - Verify – Check that the relocated value can fit in the relocation field, and generate an error if the value is too big to fit in its relocation field.
  - None – Perform no operation on the relocated value.

Table 12-5 Relocation types

Name	Value	Field	Relocation	Result	Action
R_HEX_NONE	0	None	None	None	None
R_HEX_B22_PCREL	1	Word32_B22	(S + A - P) >> 2	Signed	Verify
R_HEX_B15_PCREL	2	Word32_B15	(S + A - P) >> 2	Signed	Verify
R_HEX_B7_PCREL	3	Word32_B7	(S + A - P) >> 2	Signed	Verify
R_HEX_LO16	4	Word32_LO	(S + A)	Unsigned	Truncate
R_HEX_HI16	5	Word32_LO	(S + A) >> 16	Unsigned	Truncate
R_HEX_32	6	Word32	(S + A)	Unsigned	Truncate
R_HEX_16	7	Word16	(S + A)	Unsigned	Truncate
R_HEX_8	8	Word8	(S + A)	Unsigned	Truncate
R_HEX_GPREL16_0	9	Word32_GP	(S + A - GP)	Unsigned	Verify
R_HEX_GPREL16_1	10	Word32_GP	(S + A - GP) >> 1	Unsigned	Verify
R_HEX_GPREL16_2	11	Word32_GP	(S + A - GP) >> 2	Unsigned	Verify
R_HEX_GPREL16_3	12	Word32_GP	(S + A - GP) >> 3	Unsigned	Verify
R_HEX_HL16	13	Word32_HL	(S + A) >> 16 and (S + A)	Unsigned	Truncate
R_HEX_B13_PCREL	14	Word32_B13	(S + A - P) >> 2	Signed	Verify
R_HEX_B9_PCREL	15	Word32_B9	(S + A - P) >> 2	Signed	Verify

Table 12-5 Relocation types (cont.)

Name	Value	Field	Relocation	Result	Action
R_HEX_B32_PCREL_X	16	Word32_X26	(S + A - P) >> 6	Signed	Truncate
R_HEX_32_6_X	17	Word32_X26	(S + A) >> 6	Unsigned	Verify
R_HEX_B22_PCREL_X	18	Word32_B22	(S + A - P) & 0x3f	Signed	Verify
R_HEX_B15_PCREL_X	19	Word32_B15	(S + A - P) & 0x3f	Signed	Verify
R_HEX_B13_PCREL_X	20	Word32_B13	(S + A - P) & 0x3f	Signed	Verify
R_HEX_B9_PCREL_X	21	Word32_B9	(S + A - P) & 0x3f	Signed	Verify
R_HEX_B7_PCREL_X	22	Word32_B7	(S + A - P) & 0x3f	Signed	Verify
R_HEX_16_X	23	Word32_U6	(S + A)	Unsigned	Truncate
R_HEX_12_X	24	Word32_R6	(S + A)	Unsigned	Truncate
R_HEX_11_X	25	Word32_U6	(S + A)	Unsigned	Truncate
R_HEX_10_X	26	Word32_U6	(S + A)	Unsigned	Truncate
R_HEX_9_X	27	Word32_U6	(S + A)	Unsigned	Truncate
R_HEX_8_X	28	Word32_U6	(S + A)	Unsigned	Truncate
R_HEX_7_X	29	Word32_U6	(S + A)	Unsigned	Truncate
R_HEX_6_X	30	Word32_U6	(S + A)	Unsigned	Truncate
R_HEX_32_PCREL	31	Word32	(S + A - P)	Signed	Verify
R_HEX_COPY	32	Word32	(see below)		
R_HEX_GLOB_DAT	33	Word32	(S + A) (see below)	Unsigned	Truncate
R_HEX_JMP_SLOT	34	Word32	(S + A) (see below)	Unsigned	Truncate
R_HEX_RELATIVE	35	Word32	(B + A) (see below)	Unsigned	Truncate
R_HEX_PLT_B22_PCREL	36	Word32_B22	(L + A - P) >> 2	Signed	Verify
R_HEX_GOTREL_LO16	37	Word32_LO	(S + A - GOT)	Signed	Truncate
R_HEX_GOTREL_HI16	38	Word32_LO	(S + A - GOT) >> 16	Signed	Truncate
R_HEX_GOTREL_32	39	Word32	(S + A - GOT)	Signed	Truncate
R_HEX_GOT_LO16	40	Word32_LO	(G)	Signed	Truncate
R_HEX_GOT_HI16	41	Word32_LO	(G) >> 16	Signed	Truncate
R_HEX_GOT_32	42	Word32	(G)	Signed	Truncate
R_HEX_GOT_16	43	Word32_U16	(G)	Signed	Verify
R_HEX_DTPMOD_32	44	Word32			
R_HEX_DTPREL_LO16	45	Word32_LO	(S + A - T)	Signed	Truncate
R_HEX_DTPREL_HI16	46	Word32_LO	(S + A - T) >> 16	Signed	Truncate
R_HEX_DTPREL_32	47	Word32	(S + A - T)	Signed	Truncate
R_HEX_DTPREL_16	48	Word32_U16	(S + A - T)	Signed	Verify
R_HEX_GD_PLT_B22_PCREL	49	Word32_B22	(L + A - P) >> 2	Signed	Verify
R_HEX_GD_GOT_LO16	50	Word32_LO	(G)	Signed	Truncate
R_HEX_GD_GOT_HI16	51	Word32_LO	(G) >> 16	Signed	Truncate

Table 12-5 Relocation types (cont.)

Name	Value	Field	Relocation	Result	Action
R_HEX_GD_GOT_32	52	Word32	(G)	Signed	Truncate
R_HEX_GD_GOT_16	53	Word32_U16	(G)	Signed	Verify
R_HEX_IE_LO16	54	Word32_LO	(G + GOT)	Signed	Truncate
R_HEX_IE_HI16	55	Word32_LO	(G + GOT) >> 16	Signed	Truncate
R_HEX_IE_32	56	Word32	(G + GOT)	Signed	Truncate
R_HEX_IE_GOT_LO16	57	Word32_LO	(G)	Signed	Truncate
R_HEX_IE_GOT_HI16	58	Word32_LO	(G) >> 16	Signed	Truncate
R_HEX_IE_GOT_32	59	Word32	(G)	Signed	Truncate
R_HEX_IE_GOT_16	60	Word32_U16	(G)	Signed	Verify
R_HEX_TPREL_LO16	61	Word32_LO	(TLS - S - A)	Signed	Truncate
R_HEX_TPREL_HI16	62	Word32_LO	(TLS - S - A) >> 16	Signed	Truncate
R_HEX_TPREL_32	63	Word32	(TLS - S - A)	Signed	Truncate
R_HEX_TPREL_16	64	Word32_U16	(TLS - S - A)	Signed	Verify
R_HEX_6_PCREL_X	65	Word32_U6	(S + A - P)	Unsigned	Truncate
R_HEX_GOTREL_32_6_X	66	Word32_X26	(S + A - GOT) >> 6	Signed	Truncate
R_HEX_GOTREL_16_X	67	Word32_U6	(S + A - GOT)	Unsigned	Truncate
R_HEX_GOTREL_11_X	68	Word32_U6	(S + A - GOT)	Unsigned	Truncate
R_HEX_GOT_32_6_X	69	Word32_X26	(G) >> 6	Signed	Truncate
R_HEX_GOT_16_X	70	Word32_U6	(G)	Signed	Truncate
R_HEX_GOT_11_X	71	Word32_U6	(G)	Unsigned	Truncate
R_HEX_DTPREL_32_6_X	72	Word32_X26	(S + A - T) >> 6	Signed	Truncate
R_HEX_DTPREL_16_X	73	Word32_U6	(S + A - T)	Unsigned	Truncate
R_HEX_DTPREL_11_X	74	Word32_U6	(S + A - T)	Unsigned	Truncate
R_HEX_GD_GOT_32_6_X	75	Word32_X26	(G) >> 6	Signed	Truncate
R_HEX_GD_GOT_16_X	76	Word32_U6	(G)	Unsigned	Truncate
R_HEX_GD_GOT_11_X	77	Word32_U6	(G)	Unsigned	Truncate
R_HEX_IE_32_6_X	78	Word32_X26	(G + GOT) >> 6	Signed	Truncate
R_HEX_IE_16_X	79	Word32_U6	(G + GOT)	Unsigned	Truncate
R_HEX_IE_GOT_32_6_X	80	Word32_X26	(G) >> 6	Signed	Truncate
R_HEX_IE_GOT_16_X	81	Word32_U6	(G)	Unsigned	Truncate
R_HEX_IE_GOT_11_X	82	Word32_U6	(G)	Unsigned	Truncate
R_HEX_TPREL_32_6_X	83	Word32_X26	(TLS - S - A) >> 6	Signed	Truncate
R_HEX_TPREL_16_X	84	Word32_U6	(TLS - S - A)	Unsigned	Truncate
R_HEX_TPREL_11_X	85	Word32_U6	(TLS - S - A)	Unsigned	Truncate
R_HEX_LD_PLT_B22_PCREL	86	Word32_B22	(L + A - P) >> 2	Signed	Verify
R_HEX_LD_GOT_LO16	87	Word32_LO	(G)	Signed	Truncate

Table 12-5 Relocation types (cont.)

Name	Value	Field	Relocation	Result	Action
R_HEX_LD_GOT_HI16	88	Word32_LO	(G) >> 16	Signed	Truncate
R_HEX_LD_GOT_32	89	Word32	(G)	Signed	Truncate
R_HEX_LD_GOT_16	90	Word32_R16	(G)	Signed	Verify
R_HEX_LD_GOT_32_6_X	91	Word32_X26	(G) >> 6	Signed	Truncate
R_HEX_LD_GOT_16_X	92	Word32_U6	(G)	Unsigned	Truncate
R_HEX_LD_GOT_11_X	93	Word32_U6	(G)	Unsigned	Truncate
R_HEX_23_REG	94	Word32_B21	(S + A - MB) >> 2	Unsigned	Verify
R_HEX_GD_PLT_B22_PCREL_X	95	Word32_B22	(S + A - P) & 0x3f	Signed	Truncate
R_HEX_GD_PLT_B32_PCREL_X	96	Word32_X26	(S + A - P) >> 6	Signed	Truncate
R_HEX_LD_PLT_B22_PCREL_X	97	Word32_B22	(S + A - P) & 0x3f	Signed	Truncate
R_HEX_LD_PLT_B32_PCREL_X	98	Word32_X26	(S + A - P) >> 6	Signed	Truncate
R_HEX_27_REG	99	Word32_M25	(S + A - MB) >> 2	Unsigned	Verify

NOTE: If a relocation formula contains a reference to either G or GOT, an entry for the indicated symbol is implicitly created in the global offset table (GOT).

NOTE: If a relocation formula contains a reference to L, an entry for the indicated symbol is implicitly created in the procedure linkage table (PLT).

12.4.4 Special relocation types

Four of the relocation types listed in Table 12-5 have special semantics – they are the only types the loader supports, while generating an error for the other relocation types.

Table 12-6 Special relocation types

Name	Description
R_HEX_COPY	Dynamic linking support – The offset refers to a location in a writable segment, and the symbol table index specifies a symbol that exists in both the current object file and a shared object. During execution, the dynamic linker copies the field associated with the shared object's symbol into the location in the current object specified by the offset.
R_HEX_GLOB_DAT	Similar to relocation type `R_HEX_32`, except that it sets a GOT entry to the address of the specified symbol.
R_HEX_JMP_SLOT	Dynamic linking support – The offset refers to the location of a GOT entry, which the dynamic linker modifies with the address of the specified symbol.
R_HEX_RELATIVE	Dynamic linking support – The offset refers to a location in a shared object that contains a value representing a relative address. The loader computes the corresponding virtual address by adding the relative address to the virtual address that the shared object was loaded at. **NOTE:** The symbol table index must be set to 0.

12.5 Linker-generated symbols

The following special symbols are generated by the linker and affect runtime.

Table 12-7 Linker-generated symbols

Name	Description
_SDA_BASE_	Denotes the value of the GP register and the beginning of the small data area.
_TLS_START_	Denotes the beginning of the TLS template area in the current image.
_TLS_DATA_END_	Denotes the end of the initialized TLS data within the TLS template area of the current image.
_TLS_END_	Denotes the end of the TLS template area of the current image.
__end	Denotes one byte past the highest virtual address occupied by the present image.

13 Program headers

When a loadable segment is loaded into memory, the contents of element `p_flags` specify the corresponding segment permissions.

14 Dynamic linking

Dynamic linking involves loading objects into an application program (and linking them) at runtime rather than compile time.

The mechanics of shared object loading are specific to the target operating system. Hexagon object files use the same data structures for dynamic linking that are used in the System V ABI.

Shared objects are often intended to be referenced by more than one client object, which requires that their code be executable in a reentrant manner. Reentrant objects are commonly implemented by maintaining separate instances of the object data section for each reference to the shared object.

The mechanism used for creating multiple data section instances is implementation-defined. Here are two examples of how the data sections can be implemented:

1. The object loader marks all pages for a data section as read-only, and relies on the virtual memory manager to create write-able copies of the pages, which are written to privately from the calling-process context (a policy known as *copy-on-write*).
2. The object loader creates a new instance of the shared object for each reference.

14.1 Dynamic section

In addition to supporting the standard ELF dynamic array tags, Hexagon object files can also include processor-specific dynamic array tags to support dynamic linking.

Table 14-1 lists the processor-specific dynamic array tags (along with the standard ELF dynamic array tag `DT_PLTGOT`):

- Name – Specifies the name of the tag.
- Value – Specifies the numeric identifier of the tag.
- d_un – Specifies whether the tag represents an integer or program virtual address. (d_un is the name of a union that is used to store elements in a dynamic section.)
- Executable – Specifies whether or not the dynamic linking array for an executable object file must contain a tag entry of the specified type. Optional indicates that an entry can appear in the array, but is not required.
- Shared object – Specifies whether or not the dynamic linking array for a shared object file must contain a tag entry of the specified type.

Table 14-1 Dynamic array tags

Name	Value	d_un	Executable	Shared object
DT_PLTGOT	0x00000003	d_ptr	Optional	Optional
DT_HEXAGON_SYMSZ	0x70000000	d_val	Optional	Optional
DT_HEXAGON_VER	0x70000001	d_val	Mandatory	Mandatory
DT_HEXAGON_PLT	0x70000002	d_val	Optional	Optional

These tags have the following descriptions:

- `DT_PLTGOT` – Image offset of the GOT (Section 14.2).
- `DT_HEXAGON_SYMSZ` – Size (in bytes) of the symbol table pointed by `DT_SYMTAB`. This value is equivalent to the value of `DT_SYMENT` multiplied by the value of field `nchain` in the hash table pointed by `DT_HASH`.
- `DT_HEXAGON_VER` – Version of interface with dynamic linker (Section 14.3). Currently, it can be set to the integer value 2 or 3. For the object to be compatible with the Hexagon ABI, it must be set to 3. The default is 2.
- `DT_HEXAGON_PLT` – Image offset of the PLT (Section 14.3).

14.2 Global offset table

The global offset table (GOT) is an array of absolute addresses which enables an object to use position-independent code.

At load-time, the relocations related to GOT-relative dynamic symbols are resolved, and the GOT is populated with their absolute addresses.

The zero'th GOT entry (index = 0) is reserved for holding the address of the dynamic structure, which is referenced through the symbol `_DYNAMIC`. The three subsequent entries are reserved for use by the dynamic linker.

The GOT is referenced through the symbol `_GLOBAL_OFFSET_TABLE_`. It can reside anywhere in the section, `.got`, and it can accept positive and negative indexes given its declaration as:

```
extern Elf32_Addr _GLOBAL_OFFSET_TABLE_ [];
```

NOTE: Any change to the format of the GOT will be reflected in `DT_HEXAGON_VER` (Section 14.1).

14.3 Procedure linkage table

The procedure linkage table (PLT) enables execution transfers from one object to another, with the resolution of function symbols performed at runtime.

Each object that references external functions has a PLT, which is composed of an array of stubs for each of the external functions. The dynamic linker determines the absolute addresses of the destinations and stores them in the GOT, from which they are loaded by the stub code in the PLT entry.

A relocation table is associated with the PLT. The `DT_JMPREL` tag in the `_DYNAMIC` array indicates the location of the first relocation entry. The relocation table entries, after any reserved entry, match the PLT entries in a one-to-one correspondence – for example, relocation table entry 16 applies to the 16th PLT entry after the reserved PLT entries.

The relocation type associated with each non-reserved PLT entry must be `R_HEX_JMP_SLOT`. The relocation offset must specify the address of a GOT entry containing the address of the function, and the symbol table index must reference the appropriate symbol.

The resolution of external function symbols is beyond the scope of this specification; however, the PLT is designed to allow on-demand resolution of such symbols. In other words, the external function symbols need not be resolved before execution begins: each symbol is resolved only when it is called (a policy known as *lazy binding*, which avoids the overhead of resolving and relocating symbols that are never called).

The interface between the PLT and dynamic linker is performed via registers R14 and R15, which respectively contain the PLT relocation table entry index and the object identification. Additionally, R28 is available for use as a scratch register by the PLT stub code.

The details of the PLT are implementation-defined. For example, the following code example presents a possible PLT design, with the first four entries reserved for interfacing with the dynamic linker to perform lazy binding.

```
.PLT0:                          // entry reserved for dyn linker
r15 = pc                        // address of .PLT0
r28.h = #hi (.PLT0@GOT)          // offset of .PLT0 from GOT
r28.l = #lo (.PLT0@GOT)
r28 = sub (r15, r28)            // address of GOT
r15 = memw (r28 + #8)           // object ID at GOT [2]
r14 = sub (r14, r28)            // offset of @GOT from GOT

.PLT1:                          // entry reserved for dyn linker
r14 = add (r14, #-16)           // subtract reserved PLT entries
r14 = asr (r14, #2)             // index of PLT relocation
r28 = memw (r28 + #4)           // dynamic linker at GOT [1]
jumpr r28                       // call dynamic linker
nop
nop

.PLT2:                          // entry reserved for dyn linker
...

.PLT3:                          // entry reserved for dyn linker
...
```

```
.PLT4:                         // entry for name1
r15 = pc                       // address of .PLT4
r14.h = #hi (name1's GOT - .PLT4) // offset of name1 GOT from entry
r14.l = #lo (name1's GOT - .PLT4)
r14 = add (r15, r14)           // address of name1 GOT
r28 = memw (r14)               // contents of name1 GOT
jumpr r28                      // call it

.PLT5:                         // entry for name2
r15 = pc
r14.h = #hi (name2's GOT - .PLT5)
r14.l = #lo (name2's GOT - .PLT5)
r14 = add (r15, r14)
r28 = memw (r14)
jumpr r28

.PLT6:
...
```

NOTE: For details on the `@GOT` suffix see Section 16.2.

The following steps are assumed in resolving external function symbols. This procedure uses 24-byte long PLT entries.

1. The program loader sets the second entry of the GOT to the address of the dynamic linker (in order to resolve external function symbols), and the third entry of the GOT to some identifying information unique to the object.
2. The linker initializes the first few PLT entries reserved to it with code that will marshal arguments to the dynamic linker.

   In this example, the code at the beginning of the first PLT entry finds the offset from it to the GOT (as set by the linker). The offset is then used to calculate the absolute address of the GOT. Register R14 is set to index into the PLT relocation table for function `name1`, and R15 is set to the object ID. The address of the dynamic linker is loaded from a GOT entry and called.
3. When the function `name1` is called, execution is transferred to its assigned PLT entry (in this example, `.PLT4`).
4. The PLT entry contains a stub that sets R15 to its address and then calculates the address of the GOT entry for the symbol, `name1`.

   Control is then transferred indirectly to the contents of this GOT entry, which initially contains the address of the first PLT entry (`.PLT0`).
5. The preceding PLT code example is executed, starting at label `.PLT0`.

   With the PLT entry index for `name1` and its corresponding GOT entry offset, the dynamic linker can find the associated relocation entry for the symbol, whose offset points to the GOT entry for the symbol, and whose symbol index points to the function symbol (`name1` in this example).

6. When the dynamic linker resolves the symbol `name1`, it modifies the GOT entry associated with its PLT entry with its actual address, and then transfers control to it.

   Subsequent calls to `name1` land immediately at its address (after a stop at its PLT entry), but without calling the dynamic linker again.

NOTE: Any change to the interface with the dynamic linker will be reflected in `DT_HEXAGON_VER` (Section 14.1).

15 Thread-local storage

The Hexagon development tools support thread-local storage (TLS), which is data that is allocated at runtime and accessible only by the thread that allocates it.

15.1 Thread-local storage section

Both initialized and uninitialized TLS data can be specified. At runtime, when a thread is created the initialized data must be provided. This data is provided in the sections identified with `SHF_TLS` flag:

- `.tdata` – Initialized TLS data
- `.tbss` – Uninitialized data defined as `COMMON` symbols

These combined sections form the TLS template that is used when a thread is created.

Symbols associated with the TLS data are assigned type `STT_TLS`. Their offset is relative to the beginning of the TLS template – this offset is used in the `st_value` field of defined TLS symbols.

The TLS template resides in a segment marked with `PT_TLS`.

15.2 Runtime allocation

The TLS area is allocated in the following situations:

- Whenever a thread is created (including the main executable thread)
- Whenever the TLS area of a shared object is referenced for the first time after an executable was started

When creating the TLS area for a thread (including the main executable thread), the runtime linker combines the TLS templates for all loaded shared objects – including the executable and shared libraries – into a single TLS template. This single TLS template is then used to initialize the TLS area allocated for each new thread.

15.3 Load-time allocation

When a shared object is loaded after process startup, and the object contains a TLS template, a TLS area might be allocated when the TLS data is first referenced.

When a shared object is unloaded, the memory associated with its TLS template is freed (depending on the access method used – see Section 15.5 for details).

15.4 Interface

The TLS area is accessed at the processor level through the special register UGP (Section 10.1). This register is set to the address one location above the TLS area, which grows downwards from UGP.

Depending on the TLS access method in use (Section 15.5), different methods are used to locate the TLS area.

The following (implementation-defined) function is assumed to be available:

```
typedef struct
{
  size_t ti_moduleid;
  ptrdiff_t ti_tlsoffset;
} TLS_index;

extern void *__tls_get_addr (TLS_index *);
```

This function returns a pointer to the address of the specified TLS data.

15.5 Thread-local storage access

The location of specific data items in an object's TLS area is known at link time, and this information can be used to directly access the data. However, shared objects containing a TLS template can be loaded after process startup. In this case, more general methods are required for an executable object to access its TLS data.

The following general methods can be used to access TLS data – they are listed here in order from the most general to the most restrictive (with the most restrictive methods also being the most efficient):

- General dynamic (GD)
- Local dynamic (LD)
- Initial executable (IE)
- Local executable (LE)

NOTE: Subject to the specified constraints, any of these methods can be used in a given shared object – the choice depends on which method is the most convenient to use in a specific instance.

15.5.1 General dynamic (GD)

In the GD method, TLS data can be referenced from either a shared object or an executable object.

To obtain the address of a variable in the TLS area, the code must call the function `__tls_get_addr ()` with the address of a `TLS_index` structure (Section 15.4).

NOTE: In a shared object, the TLS template can be loaded only after process startup. Otherwise, the (standard ELF) dynamic tag `DT_FLAGS` is set to the value `DF_STATIC_TLS`, which prevents the TLS data from being loaded after process startup.

15.5.2 Local dynamic (LD)

In the LD method, if a shared object contains TLS data that is protected or has local visibility, the location of the data in the TLS template is known at link time, and this location information can be used directly.

To obtain the address of a variable in the TLS area, the code must first obtain the TLS base address by calling the function `__tls_get_addr ()` with the address of a `TLS_index` structure (Section 15.4) that has a `ti_tlsoffset` member with a value of zero. The TLS variables can be accessed using the LD method through offsets from the obtained base address.

NOTE: In a shared object, the TLS template can be loaded only after process startup. Otherwise, the (standard ELF) dynamic tag `DT_FLAGS` is set to the value `DF_STATIC_TLS`, which prevents the TLS data from being loaded after process startup.

15.5.3 Initial executable (IE)

In the IE method, an object can access TLS data only if it is part of the initial TLS template established at link time. When a shared object is loaded after process startup, its TLS data cannot be accessed using this method.

To obtain the address of a variable in the TLS area, the dynamic linker creates a GOT entry to store the variable's offset in the initial TLS area.

NOTE: When a shared object is loaded after process startup, its TLS data is not part of the initial TLS template. Therefore, if a shared object tries to use this method to access its TLS data, the access must be rejected.

15.5.4 Local executable (LE)

In the LE method, an object can access the TLS data of executable objects only.

To obtain the address of a variable in the initial TLS area, simply use a static offset from the top of the initial TLS area (since the variable location is known at link time).

NOTE: If an object tries to use this method to access the TLS data of a non-executable object, the access must be rejected.

16 Coding examples

This chapter presents examples showing how the following operations can be coded in Hexagon assembly code:

- Function prologs and epilogs
- Function calls (direct and indirect)
- Branches (direct and indirect)
- Data access
- Thread-local storage

Most of the examples include separate coding solutions for two cases:

- Absolute code (which must be loaded at a specific address)
- Position-independent code (which can be loaded at an arbitrary address)

Position-independent code is used in shared objects, which typically execute in the context of another object. Pay attention to the differences between the coding solutions for absolute and position-independent code.

NOTE: The examples are not intended as definitive coding solutions – they merely demonstrate how certain basic operations can be implemented. Optimal coding solutions can be derived from the examples.

16.1 Register use

The examples in this chapter use the following register conventions:

- The absolute address of the GOT (Section 14.2) is stored in R24
- The absolute address of the application TLS area (Section 15.5) is stored in R25

Both R24 and R25 are initialized in the function prolog (Section 16.4).

NOTE: These register choices are arbitrary, and do not imply any standard coding convention for Hexagon assembly code.

16.2 Assembler symbols

Table 16-1 Special assembler symbols used in examples

Name	Description
`_GLOBAL_OFFSET_TABLE_`	Reference symbol in the global offset table (Section 14.2). Implicit address of the zero'th (index = 0) GOT entry.
`name@PCREL`	Offset to position of the symbol `name` from the current location.
`name@GOT`	Offset of GOT entry for the symbol `name` from `_GLOBAL_OFFSET_TABLE_`. Instructs linker to create GOT entry for the symbol `name`.
`name@GOTREL`	Offset to location of the symbol `name` from `_GLOBAL_OFFSET_TABLE_`.
`name@PLT`	Offset to PLT entry for the symbol `name` from the current code location. Instructs linker to create a PLT entry for the symbol `name`.
`name@GDPLT`	Offset to PLT entry for the function that returns the address of a TLS symbol `name` from the current code location, using the GD method (Section 15.5.1). This function takes the address of a `TLS_index` structure as its argument (Section 15.4).
`name@GDGOT`	Offset of first GOT entry for the `TLS_index` structure for the symbol `name` from `_GLOBAL_OFFSET_TABLE_`, using the GD method (Section 15.5.1). Instructs linker to create GOT entries for `TLS_index` structure members `ti_moduleid` and `ti_tlsoffset`.
`name@LDPLT`	Offset to PLT entry for the function that returns the address of a TLS symbol `name` from the current code location, using the LD method (Section 15.5.2). This function takes the address of a `TLS_index` structure as its argument (Section 15.4).
`name@LDGOT`	Offset of first GOT entry for the `TLS_index` structure for the symbol `name` from `_GLOBAL_OFFSET_TABLE_`, using the LD method (Section 15.5.2). Instructs linker to create GOT entries for `TLS_index` structure members `ti_moduleid` and `ti_tlsoffset`, with the latter zeroed.
`name@DTPREL`	Offset to position of the symbol `name` from base of TLS template, using the LD method (Section 15.5.2).
`name@IE`	Address of the GOT entry for the offset into the TLS area for the symbol `name`, using the IE method (Section 15.5.2). Instructs linker to create a GOT entry for the offset into the TLS area for the symbol `name`.
`name@IEGOT`	Offset of the GOT entry for the offset into the TLS area for the symbol `name from` `_GLOBAL_OFFSET_TABLE_`, using the IE method (Section 15.5.3). Instructs linker to create a GOT entry for the offset into the TLS area for the symbol `name`.
`name@TPREL`	Offset to position of the symbol `name` from base of the TLS area, using the LE method (Section 15.5.4).

16.3 Addressing constraints

The Hexagon processor architecture has certain addressing constraints which affect how data accesses are coded in Hexagon assembly language. The constraints depend on the instruction and data size:

- Some branching instructions use a 7- to 22-bit operand as a signed offset from the current program-counter (PC).
- When accessing data in the small data area (using `GP`-relative addressing), a 16-bit value is used as an unsigned offset from the `GP` register. This value is shifted left before it is added to the `GP` register – the shift amount is determined by the size of the accessed data:
    - 0 for bytes
    - 1 for half-words
    - 2 for words
    - 3 for double-words

  This limits the range that `GP`-relative addressing can access: 64K for bytes, 128K for half-words, 256K for words and 512K for double-words.
- When indexing an entry in the GOT (Section 14.2), instead of using a larger and slower 32-bit index, a program can restrict itself to just 16 bits for indexing a GOT entry. If the GOT size grows larger than 64KB, then 32-bit indexes must be used.

16.4 Function prologs and epilogs

Function *prologs* and *epilogs* are standard code sections that appear at the beginning and end of a function body. Their purpose is to manage the use of certain Hexagon processor registers within the function (Section 16.1).

This section presents examples showing how function prologs and epilogs can be coded in Hexagon assembly language.

16.4.1 Absolute

If a function calls another function, it is defined as a *non-leaf* function. In this case, in order to save the link register, a stack frame must be allocated even if no local variables are allocated from the stack.

If a function does not call another function, no stack frame is necessary (though it might be useful for traversing previous stack frames during debugging).

Table 16-2 Function prolog and epilog (absolute)

C	Hexagon assembly language
`void foo (void)`	`foo:`
`{`	`allocframe (#...)`
`...`	`...`
`}`	`dealloc_return`

When using the GD or LD method for accessing TLS (Section 15.5), the absolute address of the GOT (Section 14.2) can be stored in a register for convenience.

Table 16-3 Function prolog and epilog (absolute – GD or LD access)

C	Hexagon assembly language
`void foo ()`	`foo:`

`{`	`allocframe(#8)`
	`memw(r29+#4) = r24`
	`r24 = add(pc,##_GLOBAL_OFFSET_TABLE_@PCREL)`

`...`	`...`
`}`	`dealloc_return`

When using the IE or LE method for accessing TLS, the UGP register (Section 10.1) can be copied to a general-purpose register for convenience.

Table 16-4 Function prolog and epilog (absolute – IE or LE access)

C	Hexagon assembly language
`void foo (void)`	`foo:`

`{`	`allocframe (#...)`
	`r25 = ugp`

`...`	`...`
`}`	`dealloc_return`

16.5 Direct function call

This section presents examples showing how direct function calls can be coded in Hexagon assembly language.

16.5.1 Absolute

Direct function calls are performed with the CALL instruction, which uses the PC-relative addressing mode to specify the function address.

Table 16-5 Direct function call (absolute)

C	Hexagon assembly language	Relocation
`extern void foo (void *);`	`.extern foo`	
`foo (NULL);`	`r0 = #0`	
	`call foo`	`R_HEX_B22_PCREL`

16.5.2 Position-independent

In external function calls, the function address is not known at build time. In this case the address must be stored in a specific data structure at runtime (Chapter 14).

Assuming that the relevant data structures have been initialized, external functions are normally called through stubs contained in the procedure linkage table.

Table 16-6 Direct function call (position-independent)

C	Hexagon assembly language	Relocation
`extern void foo (void *);`	`.extern foo`	
`foo (NULL);`	`r0 = #0`	
	`call foo@PLT` [a]	`R_HEX_PLT_B22_PCREL`

[a] The @PLT suffix is optional in this symbol reference.

NOTE: Local function calls use the same code sequence for both absolute and position-independent code.

16.6 Indirect function call

This section presents examples showing how indirect function calls can be coded in Hexagon assembly language.

16.6.1 Absolute

Indirect function calls are performed with the CALLR instruction.

Table 16-7 Indirect function call (absolute)

C	Hexagon assembly language	Relocation
`extern void foo (void *);`	`.extern foo`	
`static void bar (void);`	`.local bar`	
`auto void (*fun) (void *);`		
`fun = foo;`	`r1 = ##foo`	`R_HEX_32_6_X + R_HEX_16_X`
	`memw(#fun) = r1`	`R_HEX_GPREL_2`
`fun (bar);`	`r0 = ##bar`	`R_HEX_32_6_X + R_HEX_16_X`
	`callr r1`	

If the calling function does not have access to a small data area (Section 10.1), the function addresses must be loaded from regular memory, which requires a less-efficient code sequence.

Table 16-8 Indirect function call (absolute – no small data)

C	Hexagon assembly language	Relocation
`extern void foo (void *);`	`.extern foo`	
`static void bar (void);`	`.local bar`	
`auto void (*fun) (void *);`		
`fun = foo;`	`r0 = ##fun`	`R_HEX_32_6_X | R_HEX_16_X`
	`r1 = ##foo`	`R_HEX_32_6_X | R_HEX_16_X`
	`memw(r0+#0) = r1`	
`fun (bar);`	`r0 = ##fun`	`R_HEX_32_6_X | R_HEX_16_X`
	`r1 = memw(r0+#0)`	
	`r0 = ##bar`	`R_HEX_32_6_X | R_HEX_16_X`
	`callr r1`	

16.6.2 Position-independent

In position-independent code, indirect function calls are also performed with the CALLR instruction. However, it is necessary to calculate at runtime the absolute address of the function.

Table 16-9 Indirect function call (position-independent)

C	Hexagon assembly language	Relocation

`extern void foo (void *);`
`static void bar (void);`
`auto void (*fun) (void *);`
	`.extern foo`
	`.local bar:`	

`fun = foo;`	`r20 = memw (r24 + ##foo@GOT)` [a]	`R_HEX_GOT_32_6_X` [b] `R_HEX_GOT_11_X`
`fun (bar);`	`r0 = add (pc, ##bar@PCREL)`	`R_HEX_6_PCREL_X`
	`callr r20`	

[a] If GOT is known to be smaller than 64 KB, the immediate extension can be omitted with the consequent changes to the appropriate relocation type.

[b] Relocation created for immediate constant extender.

If the GOT (Section 14.2) is known to be smaller than 64 KB, then a missing small data area requires fewer loads from memory because the callee function address value fits in 16 bits.

Table 16-10 Indirect function call (position-independent, no small data)

C	Hexagon assembly language	Relocation

`extern void foo (void *);`
`static void bar (void);`
`auto void (*fun) (void *);`
	`.extern foo`
	`.local bar`	

`fun = foo;`	`r24 =`
	`add(pc,##_GLOBAL_OFFSET_TABLE_@PCREL)`
		`R_HEX_GOT_16`

	`r1 = add(r24, #foo@GOT)`	
	`r1 = memw(r1)`	
`fun (bar);`	`r0 = add(pc,##bar@PCREL)`	`R_HEX_6_PCREL_X`
	`callr r1`	

NOTE: The absolute address of the GOT is assumed to be stored in R24 (Section 16.1).

NOTE: Because the address of the function is referenced as a data object, it is resolved along with the other data object when an object file is loaded, effectively bypassing the lazy-binding mechanism (Section 14.3).

16.7 Direct branch

This section presents an example showing how direct branches can be coded in Hexagon assembly language.

16.7.1 Absolute and position-independent

Direct branches are performed with the JUMP instruction, which uses the PC-relative addressing mode to specify the branch address.

Table 16-11 Direct branch (absolute and position-independent)

C	Hexagon assembly language	Relocation
`bar:`	`.Lbar:`	
`goto bar;`	`jump .lbar`	`R_HEX_B22_PCREL`

16.8 Indirect branch

This section presents examples showing how indirect branches can be coded in Hexagon assembly language.

16.8.1 Absolute

Indirect branches are performed with the JUMPR instruction. For instance, the following example uses an indirect branch to implement a jump table.

```
r22 = const32 (#.Lbar)
r23 = addasl (r22, r21, #2)
r23 = memw (r23)
jumpr     r23
...
.L1:
...
.L2:
...

.section .rodata, “a”, @progbits

.Lbar:
    .word    .L1
    .word    .L2
...
```

If the code performing the branch does not have access to a small data area (Section 10.1), the branch address must be loaded from regular memory, which requires a less-efficient code sequence.

```
r22 = ##.Lbar
r23 = addasl (r22, r21, #2)
r23 = memw (r23)
jumpr     r23
...
.L1:
...
.L2:
...

.section .rodata, “a”, @progbits

.Lbar:
    .word    .L1
    .word    .L2
...
```

16.8.2 Position-independent

In position-independent code, indirect branches are also performed with the JUMPR instruction. However, it is necessary to calculate at runtime the absolute address of the branch destination.

For instance, the following example loads the branch address from regular memory.

```
r22 = add (pc, ##.Lbar@PCREL)
r23 = addasl (r22, r21, #2)
r23 = memw (r23)
r23 = add (r23, r24)
jumpr    r23
...
.L1:
...
.L2:
...

.section .data, “aw”, @progbits

.Lbar:
    .word    .L1
    .word    .L2
...
```

16.9 Data access

This section presents examples showing how data accesses can be coded in Hexagon assembly language.

16.9.1 Absolute

Data accesses are performed using the Hexagon load and store instructions.

Table 16-12 Data access (absolute)

C	Hexagon assembly language	Relocation

`extern int src;`
`static int dst;`
`auto int *ptr;`
	`.extern src`
	`.lcomm dst, 4`	

`ptr = &dst;`	`r23 = const32 (#dst)`	`R_HEX_GPREL16_2 +` `R_HEX_32` [a]
`*ptr = src;`	`r22 = memw (#src)`	`R_HEX_GPREL16_2`
	`memw (r23) = r22`	

[a] Relocation created in small data area.

If the code performing the data access does not have access to a small data area (Section 10.1), the data address must be loaded from regular memory, which requires a less-efficient code sequence.

Table 16-13 Data access (absolute – no small data)

C	Hexagon assembly language	Relocation

`extern int src;`
`static int dst;`
`auto int *ptr;`
	`.extern src`
	`.lcomm dst, 4`	

`ptr = &dst;`	`r23 = ##dst`	`R_HEX_HI16` `R_HEX_LO16`
`*ptr = src;`	`r21 = memw (##src)`	`R_HEX_HI16` `R_HEX_LO16`
	`memw (r23) = r21`	

16.9.2 Position-independent

In position-independent code, data accesses are also performed with the Hexagon load and store instructions. However, it is necessary to calculate at runtime the absolute address of the data.

Table 16-14 Data access (position-independent – no small data)

C	Hexagon assembly language	Relocation

`extern int src;`
`static int dst;`
`auto int *ptr;`
	`.extern src`
	`.lcomm dst, 4`	

`ptr = &dst;`	`r24 = add(pc, ##_GLOBAL_OFFSET_TABLE_@PCREL)`	`R_HEX_GOTREL_HI16` `R_HEX_GOTREL_LO16`
	`r1 = memw(r24 + ##ptr@GOT)`	
	`memw(r1+#0) = r0`	
`*ptr = src;`	`r1 = memw(r24 + ##src@GOT)`	`R_HEX_GOT_HI16` `R_HEX_GOT_LO16`
	`r16 = memw(r1+#0)`	
	`r0 = add(pc, ##dst@PCREL)`	
	`memw(r0+#0) = r16`	

16.10 Thread-local storage

This section presents examples showing how data accesses to thread-local storage (TLS) can be coded in version-specific Hexagon assembly language.

NOTE: The following examples for Hexagon V4 also apply to Hexagon V5x and V6x.

16.10.1 Absolute

This section presents code examples using the following TLS access methods:

- GD
- LD
- IE
- LE

GD TLS data accesses in V4 use 32-bit constant extenders to make the code sequence more efficient.

Table 16-15 Thread-local storage (V4 absolute – GD access)

C	Hexagon assembly language	Relocation

`extern __thread int src;`	`.extern src`
	`.type src, @tls_object`	

`extern __thread int dst;`	`.section .tbss, "awT", @nobits`
	`.global dst`
	`.type dst, @tls_object`
	`dst: .skip 4`	

`auto int *ptr;`		
	`.text`	
`ptr = &dst;`	`r0 = add (r24, ##dst@GDGOT)` [a]	`R_HEX_GD_GOT_32_6_X` [b] + `R_HEX_GD_GOT_16_X` + `R_HEX_DTPMOD_32` [c] + `R_HEX_DTPREL_32` [c]

	`call dst@GDPLT`
	`r23 = r0`
		`R_HEX_GD_PLT_B22_PCREL`

`*ptr = src;`	`r0 = add (r24, ##src@GDGOT)` [a]	`R_HEX_GD_GOT_32_6_X` [b] + `R_HEX_GD_GOT_16_X` + `R_HEX_DTPMOD_32` [c] + `R_HEX_DTPREL_32` [c]

	`call src@GDPLT`
	`r22 = memw (r0)`
	`memw (r23) = r22`
		`R_HEX_GD_PLT_B22_PCREL`

[a] If GOT is known to be smaller than 64 KB, the immediate extension can be omitted with the consequent changes to the appropriate relocation type.

[b] Relocation created for immediate constant extender.

[c] Relocation created in GOT.

LD TLS data accesses in the V4 processor use 32-bit constant extenders to make the code sequence more efficient.

Table 16-16 Thread-local storage (V4 absolute – LD access)

C	Hexagon assembly language	Relocation

`static __thread int src;`	`.section .tbss, “awT”, @nobits`
	`.type src, @tls_object`
	`src: .skip 4`	

`static __thread int dst;`	`.type src, @tls_object`
	`dst: .skip 4`	

`auto int *ptr;`		
	`.text`	
`ptr = &dst;`	`r0 = add (r24, ##dst@LDGOT)` [a]	`R_HEX_LD_GOT_32_6_X` [b] + `R_HEX_LD_GOT_16_X` + `R_HEX_DTPMOD_32` [c]
	`call dst@LDPLT`	`R_HEX_LD_PLT_B22_PCREL`
	`r23 = add (r0, ##dst@DTPREL)` [d]	`R_HEX_DTPREL_32_6_X` [b] + `R_HEX_DTPREL_16_X`
`*ptr = src;`	`r22 = add (r0, ##src@DTPREL)` [d]	`R_HEX_DTPREL_32_6_X` [b] + `R_HEX_DTPREL_16_X`
	`memw (r23) = r22`	

[a] If GOT is known to be smaller than 64 KB, the immediate extension can be omitted with the consequent changes to the appropriate relocation type.

[b] Relocation created for immediate constant extender.

[c] Relocation created in GOT.

[d] If TLS template is known to be smaller than 64 KB, the immediate extension can be omitted with consequent changes to the appropriate relocation type.

IE TLS data accesses in the V4 processor use indirect-with-register-offset addressing mode and 32-bit constant extenders to make the code sequence more efficient.

Table 16-17 Thread-local storage (V4 absolute – IE access)

C	Hexagon assembly language	Relocation
extern __thread int src;	.extern src	
	.type src, @tls_object	
extern __thread int dst;	.section .tbss, "awT", @nobits	
	.global dst	
	.type dst, @tls_object	
	dst: .skip 4	
auto int *ptr;		
	.text	
ptr = &dst;	r23 = memw (##dst@IE)	R_HEX_IE_32_6_X [a] +
		R_HEX_IE_16_X +
		R_HEX_TPREL_32 [b]
*ptr = src;	r22 = memw (##src@IE)	R_HEX_IE_32_6_X [a]+
		R_HEX_IE_16_X +
		R_HEX_TPREL_32 [b]
	r22 = memw (r25 + r22 << #1)	
	memw (r25 + r23 << #1) = r22	

[a] Relocation created for immediate constant extender.

[b] Relocation created in GOT.

LE TLS data accesses in the V4 processor use 32-bit constant extenders to make the code sequence more efficient.

Table 16-18 Thread-local storage (V4 absolute – LE access)

C	Hexagon assembly language	Relocation
static __thread int src;	.extern src	
	.type src, @tls_object	
static __thread int dst;	.section .tbss, "awT", @nobits	
	.type dst, @tls_object	
	dst: .skip 4	
auto int *ptr;		
	.text	
ptr = &dst;	r23 = add (r25, ##dst@TPREL) [a]	R_HEX_TPREL_32_6_X [b] +
		R_HEX_TPREL_16_X
*ptr = src;	r22 = memw (r25 + ##src@TPREL)	R_HEX_TPREL_32_6_X [b] +
		R_HEX_TPREL_11_X
	memw (r23) = r22	

[a] If TLS area is known to be smaller than 64 KB, the immediate extension can be omitted with consequent changes to the appropriate relocation type.

[b] Relocation created for immediate constant extender.

16.10.2 Position-independent

This section presents code examples using the following TLS access methods:

- IE

Position-independent IE TLS data accesses in the V4 processor use indirect-with-register-offset addressing mode and 32-bit constant extenders to make the code sequence more efficient.

Table 16-19 Thread-local storage (V4 position-independent – IE access)

C	Hexagon assembly language	Relocation

`extern __thread int src;`	`.extern src`
	`.type src, @tls_object`	

`extern __thread int dst;`	`.section .tbss, "awT", @nobits`
	`.global dst`
	`.type dst, @tls_object`
	`dst: .skip 4`	

`auto int *ptr;`		
	`.text`	
`ptr = &dst;`	`r23 = memw (r24 + ##dst@IEGOT)` [a]	`R_HEX_IE_GOT_32_6_X` [b] + `R_HEX_IE_GOT_11_X` + `R_HEX_TPREL_32` [c]
`*ptr = src;`	`r22 = memw (r24 + ##src@IEGOT)` [a]	`R_HEX_IE_GOT_32_6_X` [b]+ `R_HEX_IE_GOT_11_X` + `R_HEX_TPREL_32` [b]

	`r22 = memw (r25 + r22 << #1)`
	`memw (r25 + r23 << #1) = r22`	

[a] If GOT is known to be smaller than 64 KB, the immediate extension can be omitted with consequent changes to the appropriate relocation type.

[b] Relocation created for immediate constant extender.

[c] Relocation created in GOT.