# Apiary: Software Tools for Beehive v2

Thomas L. Rodeheffer
Microsoft Research, Silicon Valley

December 16, 2009

# Contents

# Chapter 1

# Introduction

The Beehive software tools are a collection of programs to assist in software development for Beehive. Available tools include a C compiler, relocating assembler, loader, image maker, and simulator.

## 1.1 Tools

The available software tools are described briefly as follows:

`Bgcc1` The Beehive C compiler. This is a port of the GNU C compiler version 4.3.3. Chapter 2 describes how to use the C compiler.

`Bas` The Beehive relocating assembler. Chapter 3 describes the assembler in detail.

`Bar` The Beehive archiver, which gathers a collection of relocatable object files into a library archive. Section 4.1 describes how to use the archiver.

`Bld` The Beehive loader, which loads a collection of relocatable object files into an executable object file, binding external references. Section 4.2 describes how to use the loader.

`Bimg` The Beehive image maker, which constructs a binary memory image of an executable object file. Section 4.3 describes how to use the image maker.

`Bvmem` The Beehive Virtex mem file maker, which constructs code and data Virtex mem files from an executable object file. Section 4.4 describes how to use the Virtex mem file maker.

`Bsim` The Beehive simulator, which simulates the execution of an executable object file. Chapter 5 describes how to use the simulator.

`Bsimimg` A version of the Beehive simulator that takes its input from a binary memory image.

`Bfiledata` A utility that converts a file of bytes into a C source file that initializes a global to an array of those bytes. Section 4.5 describe how to use the file data utility.

## 1.2 File types

The various file types on which the tools operate are distinguished by convention using standard extensions. The standard extensions and file types are as follows:

`.c` A source file written in C.

`.h` A header (include) file for a C program.

`.as` A source file written in assembly language.

`.s` An intermediate assembler source file produced by compiling a C source file.

`.o` A relocatable object file produced by assembling an assembler source file. For convenience in tool development, a relocatable object file is an XML text file. The format of this file is described in Appendix B.

`.a` An archive of relocatable object files produced by the archiver. An archive file is treated as a library by the loader, which loads as many relocatable object files from it as are needed to satisfy unbound external references. For convenience in tool development, an archive file is an XML text file. The format of this file is described in Appendix B.

`.out` An executable object file. An executable object file is produced by the loader from a collection of relocatable object files and archives. Technically, an executable object file is identical in format to a relocatable object file. However, the expectation is that the executatble object file will have no unbound external references. The start address is the value of the symbol “main”.

`.img` An executable image file. An executable image file is produced by the image maker from an executable object file and it consists of a binary memory image starting at some specified word index in memory. The program is expected to start execution at the first word in the image.

`.mem` A Virtex mem file. A Virtex mem file is produced by the Virtex mem file maker from an executable object file. The program is expected to start execution at code address 0.

`.lst` A listing file. The assembler produces a listing file as one of its outputs during assembly. The listing file is in text format and is meant for human consumption.

`.map` A map file. The loader produces a map file as one of its outputs during assembly. The map file is in text format and is meant for human consumption.

The following chapters describe these tools and file types in detail.

## 1.3 Getting the tools

The Beehive software tools can be obtained from

```
\\msr-svc\files\users\tomr\Beehive\apiary-distv2.tar
```

Make a directory that you will use as the root directory for Beehive software tools. There is no required name for the directory but henceforth we will call it APIARY. It would be a good idea to define an environment variable APIARY that contains the name of the root directory. (In the future this may become mandatory.)

Unzipping the Beehive software tools file into APIARY reveals the following directory structure:

`APIARY/bin` Tool executables and dlls.

`APIARY/include` Include files.

`APIARY/lib` Library files.

`APIARY/src/lib` Source files used to build the library files.

`APIARY/src/hello` Source files of a simple C program.

In order to execute the Beehive software tools, you have to add `APIARY/bin` to your PATH environment variable.

# Chapter 2

# C Compiler - Bgcc1

Bgcc1 is a port of the GCC compiler version 4.3.3 to the Beehive. Appendix C describes the software conventions used by Bgcc1 in its employment of the Beehive architecture.

## 2.1 Making a bootable image

Converting a C program into a bootable image file takes four steps.

1. Compiling the C source code to assembler.
2. Assembling the assembler source code to a relocatable object file.
3. Loading the relocatable object file with startup code and libraries to produce an executable object file.
4. Converting the executable object file to a bootable image file.

Starting with a C source file *source.c*, the recommended commands for performing these four steps are as follows, where *APIARY* stands for the name of the Apiary root directory:

```
Bgcc1 -quiet -std=c99 -fno-builtin -O2 -IAPIARY/include source.c
Bas -x -datarota=2 source.s
Bld -o source.out -codebase=1000 -datafloat APIARY/lib/base.o source.o -LAPIARY/lib -lc -lgcc
Bimg source.out
```

Next these steps are described in more detail.

### 2.1.1 Compiling C to assembler

Bgcc1 compiles a C source file to assembler code. Bgcc1 is invoked using the command line

```
Bgcc1 [options] source.c
```

One source file is expected. The C source code *source.c* is compiled to the assembler source file *source.s* using the same base file name but replacing the `.c` extension with `.s`

Options start with a hypen (`-`). GCC supports a hideously enormous number of options. For a listing, use the command

```
Bgcc1 --help
```

or refer to GCC documentation such as `http://gcc.gnu.org/onlinedocs/gcc/`. Note that some of the documented options pertain to the so-called gcc driver and not to the compiler proper, which is what Bgcc1 is.

The recommended command line to compile an example C source file *source.c* to the assembler file *source.s* is as follows:

Bgcc1 -quiet -std=c99 -fno-builtin -O2 -I*APIARY*/include *source.c*

The arguments are interpreted as follows:

**-quiet** Cause the compiler not to print lots of generally uninteresting compilation statistics.

**-std=c99** Declare that we want the c99 standard dialect of C. Without this, you get an older dialect that prohibits declaring variables other than at the start of blocks, which is painful.

**-fno-builtin** Tell the compiler not to think it understands what printf and friends do.

**-O2** Ask for optimization level 2. At this level, you get nice data flow analysis and register allocation.

**-I*APIARY*/include** Add the standard Apiary include directory to the search path. *APIARY* is the Apiary root installation directory (see Section 1.3), preferably stored in the APIARY environment variable.

**source.c** This is the C source file.

### 2.1.2 Assembling

The assembly source code produced by Bgcc1 has several properties that affect how it must be assembled. (1) Bgcc1 uses symbol names for external references without ever defining them. (2) Bgcc1 assumes that the data segment is byte-addressed.

The recommended command line to assemble the compiler-generated assembler source file *source.s* to the relocatable object file *source.o* is as follows:

Bas -x -datarota=2 *source.s*

The arguments are interpreted as follows:

**-x** Cause the assembler to treat each undefined symbol as an external reference.

**-datarota=2** Specify that the data segment is byte-addressed.

**source.s** This is the assembler source file.

### 2.1.3 Loading

Converting the relocatable object file into an executable object file requires supplying startup code and various libraries. The startup code prepares the C environment and is described in Section 2.2.

The library *libc.a* is a basic C library containing malloc, some string routines, and simple input and output including printf. It also includes some beehive-specific support routines described in Appendix **??**.

The library *libgcc.a* contains compiler runtime support routines for arithmetic, shifting, and for fetching and storing bytes and shorts. This library must always appear last.

Assuming that the intent is eventually to produce a binary memory image that can be loaded by the level 1 Beehive boot loader, the requirements of this boot loader must also be kept in view. These requirements are as follows. (1) The binary memory image must be a continuous sequence of words starting at memory word index 0x1000. (2) The program must start execution at memory word index 0x1000. These requirements can be satisfied by supplying the proper arguments to Bld.

The recommended command line to load the compiler-generated and assembled relocatable object file *source.o* to produce the executable object file *source.out* is as follows:

Bld -o *source.out* -codebase=1000 -datafloat *APIARY*/lib/base.o *source.o* -L*APIARY*/lib -lc -lgcc

The arguments are interpreted as follows:

**-o source.out** Specify that the output file is *source.out*

**-codebase=1000** Specify that the code segment will be relocated to start at memory word index 0x1000. This is the required base load address for the level 1 Beehive boot loader.

**-datafloat** Specify that the data segment will be relocated to start after the end of the code segment.

***APIARY*/lib/base.o** Cause the loader to start off by loading the startup code which is found in the library module base.o. *APIARY* is the Apiary root installation directory (see Section 1.3), preferably stored in the APIARY environment variable.

**source.o** Cause the loader to continue by loading the relocatable object code compiled and assembled from source.c. Additional relocatable object files may be listed at this point.

**-L*APIARY*/lib** Cause the loader to add the directory *APIARY*/lib to the library search path.

**-lc** Cause the loader to find the library archive libc.a on the library search path and load all relocatable object files needed to satisfy external references.

**-lgcc** Cause the loader to find the library archive libgcc.a on the library search path and load all relocatable object files needed to satisfy external references.

### 2.1.4 Making a bootable image

Assuming that the executable object file was loaded with the proper arrangement to become a bootable image file, the actual bootable image file is produced by Bimg. The recommended command line is as follows:

Bimg *source.out*

The arguments are interpreted as follows:

**source.out** This is the executable object file.

The resulting bootable image file is *source.img*

## 2.2 Startup code

The startup code prepares the C execution environment. It contains the executable entry point `main` and has the responsibilities of (1) initializing the assume zero register `zero`, (2) initializing the stack pointer register `sp`, and then (3) calling the C language main subroutine `main` with no parameters. Software register usage is described in Section C.1.

Since many arrangements for booting programs require that the program start execution at the first word loaded, it is also convenient for the startup code to be loaded first so that the executable entry point `main` occurs in the correct place. Such an arrangement permits the C language main subroutine to appear anywhere in memory, Note that C language global symbol names are prefixed with an underscore (_) in assembly code. Thus the C language main subroutine can be distinguished from the executable start address.

The recommended startup code module `base.o` sets up a small stack which should suffice for simple programs. However, an alternate startup module may be provided. The Apiary library includes the following startup code object files:

***APIARY*/lib/base.o** Includes a 200 word stack in the data segment.

***APIARY*/lib/basehs.o** Initializes the stack pointer to `0xfffffffc`, which is the highest word address in data memory.

***APIARY*/lib/basemc.o** Multicore startup code. Includes an array of 16 stacks in the data segment, each 256 words long and aligned on a cache line boundary. Initializes the stack pointer to the top of the stack corresponding to the current core. Note that since global variables are in general not aligned on cache line boundaries they are problematic to use in multicore programming, since flushing one variable may overwrite others with stale data.

# Chapter 3

# Assembler - Bas

Bas is a relocating assembler for Beehive [1] version 2. The instruction set is summarized in Appendix A. Bas reads one or more assembler source files and writes a relocatable object file and, if requested, a listing file. The format of the object file is described in Appendix B.

## 3.1 Running Bas

Bas is invoked using the command line

```
Bas [options] sourcea.s sourceb.s ...
```

Options start with a hypen (`-`). The following options are supported:

**-o *out*** Use *out* as the file name for the relocatable output file. Note there is a space between -o and *out*. The default is to use name of the first input file with its extension replaced with `.o`

**-lst *lst*** Use *lst* as the file name for the listing file. Note there is a space between -lst and *lst*. The default is to omit the listing file.

**-x** Automatically define all otherwise undefined symbols as external references to global symbols.

**-I*dir*** Add the directory *dir* to the list of directories searched for include files. Note there is no space between -I and *dir*.

**-codebase=n** Set the base of the default code segment `.text` to word index $n$, where $n$ is a hexadecimal number. The default is 0. Note that this is a word index, so it is unaffected by address rotation. The loader typically changes the base during relocation, so the base given to Bas really only has significance for the listing file.

**-database=n** Set the base of the default data segment `.data` to word index $n$, where $n$ is a hexadecimal number. The default is 0. Note that this is a word index, so it

is unaffected by address rotation. The loader typically changes the base during relocation, so the base given to Bas really only has significance for the listing file.

**-coderota=n** Set the rotation of code addresses to $n$, where $n$ is a hexadecimal number. The default is 0, which produces a word-addressed architecture for code addresses. The code address rotation controls how current location advances in all code segments.

**-datarota=n** Set the rotation of data segment addresses to $n$, where $n$ is a hexadecimal number. The default is 0, which produces a word-addressed architecture for data addresses. The data address rotation controls how the current location advances in all data segments and in the absolute segment.

Although any extension may be used for the assembler source files, the extension `.as` is recommended for user-written source files. Multiple source files are concatenated to form a single input source program. The following extensions are recommended for the output files:

`.lst` the listing file
`.o` the object file

## 3.2 Source format

The assembler source files consist of comments, label definitions, symbol definitions, instructions, and directives. Generally each source line contains zero or more label definitions and, optionally, an equate definition or an assembler statement. However, multiple logical lines can be placed on the same physical line by separating them with semicolons.

## 3.3 Comments

An end-of-line comment starts with `//` and goes up to the end of the line. A multiline comment starts with `/*` and ends with a matching `*/` and may be nested. Comments are treated as white space.

```
// this is an end-of-line comment
/* this is a multiline comment
    /* which may be nested */ */
```

## 3.4 Identifiers, numbers, and strings

A word is a non-empty sequence of alphabetic characters, digits, ".", and "_". An identifier is a word that does not start with a digit. An identifier that starts with "." is special as explained later. A number is a word that starts with a digit. As in C, if the number starts with "0x" it is interpreted in hexadecimal, otherwise if it starts with "0" it is interpreted in octal, otherwise it is interpreted in decimal.

| | |
|---|---|
| + | positive |
| - | negative |
| ~ | bit complement |
| $ | register number |

Table 3.1: Prefix operators

| | |
|---|---|
| + | addition |
| - | subtraction |
| \| | bit or |
| & | bit and |
| ^ | bit xor |
| * | multiplication |
| / | unsigned division |
| % | unsigned remainder |
| ROR | rotate right |
| ROL | rotate left |
| LSR | logical shift right |
| LSL | logical shift left |
| ASR | arithmetic shift right |
| ASL | arithmetic shift left (same as LSL) |

Table 3.2: Infix operators

A string is enclosed in double quotes (") and has the usual escapes using backslash (\) as in C. A string can span multiple lines. Escaping the newline prevents the newline from being part of the string. A string of up to four characters can be used as a constant in an expression. The first character defines the low order eight bits, the second character (if any) the next eight bits, and so on, with any leftover bits being defined as zero.

## 3.5 Expressions

Words can be combined into expressions using parenthesis, prefix operators, and infix operators. Prefix operators have precedence over infix operators. For simplicity, all infix operators have the same precedence and associate to the left. Table 3.1 shows the prefix operators and Table 3.2 the infix operators. Note that the infix operators ROR, ROL, and so on are reserved words in the grammar. Although the infix operator ASL is provided for completeness, it is the same as LSL.

As explained in Section 3.7, expressions compute values and values have types. In most cases the arguments of operators must be absolute numbers.

## 3.6 Registers

Ordinary registers are specified via register numbers using the dollar sign ($) prefix operator. The ordinary registers are $0, $1, $2, etc. You can also write expressions such as $(3+4) but this is probably not very useful. An identifier may be defined as an ordinary register.

Special registers are specified via the following predefined identifiers:

**pc** the program counter register, read via Ra overload 31
**link** the link register, read via Ra overload 30
**rq** the read queue register, read via Ra overload 29
**wq** the write queue register, written via Rw overload 31

Bas ensures that ordinary registers and special registers are used only in their proper places. For example, Bas checks that $31 is not specified for Ra, which would not work because of Ra overloading.

## 3.7 Values and types

Expressions compute values and values have types.

The simplest type is an *absolute number* such as 0, 1, 2, etc. Strings that are used as constants in an expression are also considered to be absolute numbers. Absolute numbers can be combined in an expression using any of the arithmetic and bit operators.

Another type is a *register number*. A register number is obtained by applying the prefix register number operator ($) to an absolute number. Each of the special registers is also its own type. Register numbers and special registers cannot be further combined in an expression.

A *relocatable offset* is another class of types. Examples of these types are offsets in a segment and offsets from an external reference to a global symbol. Each different basis of relocation gives rise to a unique type. So, for example, offsets in one code segment are one type, offsets in another code segment are a second type, offsets in a data segment are a third type, and offsets from a particular external symbol are yet a fourth type. Two relocatable offsets can be subtracted from one another, producing an absolute number, provided that the offsets are of the same type. An absolute number can be added or subtracted from a relocatable offset with the obvious result.

The final type is the *exprpatch*. An exprpatch is a symbolic expression tree in which the leaves are absolute numbers and relocatable offsets and the operators are addition, subtraction, bitwise inclusive or, bitwise clear, and rotation. The exprpatch is the general type that is handled by the relocating loader. Observe that absolute numbers and relocatable offsets can be promoted trivially into exprpatches.

Bas constructs an exprpatch as its internal representation of how to convey information from “long_ld” and similar instructions to the loader. Another operator available in the exprpatch is “mbz”, which is a loader-checked assertion that certain bits in a symbolic value must be zero. Bas uses the mbz operation to convey information from the “x_lli” instruction to the loader.

Unfortunately, Bas currently does not permit the user to construct an exprpatch as the result of an expression. This deficiency will be fixed when time permits.

## 3.8 Label definitions

A label definition consists of an identifier followed by a colon:

```
identifier: // a label definition
```

The identifier is assigned the address of the current location. Note that the current location can be a relocatable offset in a segment or an absolute value.

Multiple label definitions may appear at the start of a line. Any given identifier can be defined at most once.

## 3.9 Equate definitions

An equate definition consists of an identifier followed by an equals sign (=) followed by an expression followed by the end of the line. Any identifiers used in the expression must be defined earlier in the input source.

```
identifier = expression // an equate definition
```

The identifier is assigned the value of the expression. Note that values come in various types. A value can be an absolute number, a relocatable offset, a register number, or one of the special registers.

## 3.10 Instructions

An instruction consists of an opcode followed by a comma-separated list of arguments:

```
opcode arg,arg,arg,...
```

Each argument is an expression. The opcode defines a semantic operation with a given number of arguments that is to be assembled into a certain number of machine instructions. Most opcodes assemble into one instruction but a few assemble into two instructions. The assembled instructions are emitted into the current segment.

Although lexically an opcode is an identifier, it is not in the same namespace as predefined and user defined identifiers. Opcodes are not reserved words. The various classes of instructions are described next.

### 3.10.1 Basic functions

The Beehive CPU supports eight basic arithmetic and logical functions which are specified via opcodes as follows:

```
add  w,a,b  // w = a + b
sub  w,a,b  // w = a - b
or   w,a,b  // w = a | b
orn  w,a,b  // w = a | ~b
and  w,a,b  // w = a & b
andn w,a,b  // w = a & ~b
xor  w,a,b  // w = a ^ b
xorn w,a,b  // w = a ^ ~b
```

Each of the arguments is an expression that specifies a value as follows:

**w** a register number or a special register **wq** or **link**.
**a** a register number or a special register **pc**, **link**, or **rq**.
**b** a register number or an absolute number `0..0xfff`.

The instruction assembles into a machine instruction using the NOSHIFT op in order to permit the widest range of constants.

### 3.10.2 Basic function with address queue pushes

The Beehive CPU has machine instructions that push the result of any of the eight basic functions onto the address queue in addition to writing it to the destination register. These machine instructions are specified by opcode families derived from each of the basic function opcodes. For simplicity, we show only the opcode family for "add". Analogous families exist for each of the other basic functions.

```
aqr_add  w,a,b  // aqr = w = a + b (memory read)
aqw_add  w,a,b  // aqw = w = a + b (memory write)
```

Each of the arguments is an expression that specifies a value as follows:

**w** a register number or a special register **wq** or **link**.
**a** a register number or a special register **pc**, **link**, or **rq**.
**b** a register number or an absolute number `0..0xfff`.

### 3.10.3 Basic function with shifts

The Beehive CPU has machine instructions that apply an arbitrary shift of any of five types to the result of any of the eight basic functions. These machine instructions are specified by opcode families derived from each of the basic function opcodes. For simplicity, we show only the opcode family for "add". Analogous families exist for each of the other basic functions.

```
add_ror  w,a,b,s  // w = (a + b) rotate right s
add_rol  w,a,b,s  // w = (a + b) rotate left s
add_lsr  w,a,b,s  // w = (a + b) logical shift right s
add_lsl  w,a,b,s  // w = (a + b) logical shift left s
add_asr  w,a,b,s  // w = (a + b) arithmetic shift right s
```

Each of the arguments is an expression that specifies a value as follows:

**w** a register number or a special register **wq** or **link**.
**a** a register number or a special register **pc**, **link**, or **rq**.
**b** a register number or an absolute number `0..0x7f`.
**s** an absolute number 0..31.

Note that the permissible range of absolute numbers in argument b is reduced considerably because of the necessity to specify a shift count.

The Beehive CPU version 2 omits *rotate left* since it is redundant with *rotate right*. Therefore Bas assembles *rotate left* $s$ as *rotate right* $32 - s$.

### 3.10.4 Basic function with jumps

The Beehive CPU has machine instructions that conditionally jump to an address which is the result of any of the eight basic functions. These machine instructions are specified by opcode families derived from each of the basic function opcodes. For simplicity, we show only the opcode family for "add". Analogous families exist for each of the other basic functions.

```
call_add a,b  // link = nextpc; goto (a + b)
j_add    a,b  // goto (a + b)
jz_add   a,b  // if (ZERO) goto (a + b)
jm_add   a,b  // if (MINUS) goto (a + b)
jc_add   a,b  // if (CARRY) goto (a + b)
jnz_add  a,b  // if (!ZERO) goto (a + b)
jnm_add  a,b  // if (!MINUS) goto (a + b)
jnc_add  a,b  // if (!CARRY) goto (a + b)
j0_add   a,b  // class 1 jump 0
j1_add   a,b  // class 1 jump 1
j2_add   a,b  // class 1 jump 2
j3_add   a,b  // class 1 jump 3
j4_add   a,b  // class 1 jump 4
j5_add   a,b  // class 1 jump 5
j6_add   a,b  // class 1 jump 6
j7_add   a,b  // class 1 jump 7
```

Each of the arguments is an expression that specifies a value as follows:

**a** a register number or a special register **pc**, **link**, or **rq**.
**b** a register number or an absolute number `0..0x1ffff`.

### 3.10.5 Class 1 jump instructions

The Beehive CPU class 1 jump instructions are interpreted by special function units and generally they do not actually jump, although they fetch ALU operands in the normal way, which may include pulling a word from the read queue. To assist in specifying class 1 jump instructions that may have an unusual structure, Bas provides special opcodes. The following opcodes put "w" into the low-order four bits of the Rw field, and 0 into the Ra, Rb, Count, Const, and Fun fields:

```
j0w  w  // class 1 jump 0
j1w  w  // class 1 jump 1
j2w  w  // class 1 jump 2
j3w  w  // class 1 jump 3
j4w  w  // class 1 jump 4
j5w  w  // class 1 jump 5
j6w  w  // class 1 jump 6
j7w  w  // class 1 jump 7
```

Each of the arguments is an expression that specifies a value as follows:

**w** an absolute number `0..15`.

The following opcodes put "x" into the Fun field and assemble "a" as ALU argument a and "b" as ALU argument b:

```
j0x  x,a,b  // class 1 jump 0
j1x  x,a,b  // class 1 jump 1
j2x  x,a,b  // class 1 jump 2
j3x  x,a,b  // class 1 jump 3
j4x  x,a,b  // class 1 jump 4
j5x  x,a,b  // class 1 jump 5
j6x  x,a,b  // class 1 jump 6
j7x  x,a,b  // class 1 jump 7
```

Each of the arguments is an expression that specifies a value as follows:

**x** an absolute number `0..7`.
**a** a register number or a special register **pc**, **link**, or **rq**.
**b** a register number or an absolute number `0..0x1ffff`.

### 3.10.6 Synthesized loads

It may be observed that the Beehive CPU lacks instructions that load one register from another or from a constant. However, in many cases the desired effect can be obtained by employing a proper selection of ALU function and arguments. This is particularly effective if some register can be assumed to contain a useful value such as, for example, zero. See the .assume directive for how to tell the assembler about an assumed value. Bas provides the following synthesized load opcodes that assemble into a single machine instruction:

```
ld      w,f    // w = f
aqr_ld  w,f    // aqr = w = f (memory read)
aqw_ld  w,f    // aqw = w = f (memory write)
ror     w,f,s  // w = f rotate right s
rol     w,f,s  // w = f rotate left s
lsr     w,f,s  // w = f logical shift right s
lsl     w,f,s  // w = f logical shift left s
asr     w,f,s  // w = f arithmetic shift right s
```

Each of the arguments is an expression that specifies a value as follows:

**w** a register number or a special register **wq** or **link**.
**f** a register number; a special register **pc**, **link**, or **rq**; the absolute numbers `0` or `0xffffffff`; or any constant offset up to plus or minus `0xfff` (only `0x7f` in the case of the shift opcodes) from an assumed register value.
**s** an absolute number `0..31`.

Note that specifying a register number of $29, $30, or $31 in argument f requires having an assumed zero register in order to get around Ra overloads. Note that special register **pc** always has an assumed value.

The Beehive CPU version 2 omits *rotate left* since it is redundant with *rotate right*. Therefore Bas assembles *rotate left* $s$ as *rotate right* $32 - s$.

### 3.10.7 Synthesized jumps

As in the case of synthesized loads, Bas provides the following synthesized jump opcodes that assemble into a single machine instruction:

```
call f  // link = nextpc; goto f
j    f  // goto f
jz   f  // if (ZERO) goto f
jm   f  // if (MINUS) goto f
jc   f  // if (CARRY) goto f
jnz  f  // if (!ZERO) goto f
jnm  f  // if (!MINUS) goto f
jnc  f  // if (!CARRY) goto f
j0   f  // class 1 jump 0 alu=f
j1   f  // class 1 jump 1 alu=f
j2   f  // class 1 jump 2 alu=f
j3   f  // class 1 jump 3 alu=f
j4   f  // class 1 jump 4 alu=f
j5   f  // class 1 jump 5 alu=f
j6   f  // class 1 jump 6 alu=f
j7   f  // class 1 jump 7 alu=f
```

The argument f is an expression that specifies a value as follows:

**f** a register number; a special register **pc**, **link**, or **rq**; the absolute numbers `0` or `0xffffffff`; or any constant offset up to plus or minus `0x1ffff` from an assumed register value.

Note that specifying a register number of $29, $30, or $31 in argument f requires having an assumed zero register in order to get around Ra overloads. Note that special register **pc** always has an assumed value. This is particularly useful in the case of synthesized jumps.

For the class 1 jumps, these instructions assemble to produce f as the output of the ALU, just like a normal jump would do. This may or may not be what you want.

### 3.10.8 Load link immediate

The Beehive CPU has a "load link immediate" instruction that loads the **link** register with any constant whose low order four bits are zero:

```
lli   i  // link = i
x_lli x  // link = x
```

The argument is an expression that specifies a value as follows:

**i** an absolute number whose low order four bits are zero.
**x** a relocatable offset. The loader will verify that the low order four bits are zero.

The `x_lli` instruction can be used to get the address of a table into the **link** register provided that the table is properly aligned in memory, for example by using an `.align 16` directive.

### 3.10.9 Synthesized long loads

Any 32-bit value can be loaded into a register by using a "load link immediate" instruction to place the high order 28 bits in the **link** register followed by an "or" instruction to combine it with the low order 4 bits. Bas provides the following opcodes that assemble to this sequence:

```
long_ld      w,k  // w = k
aqr_long_ld  w,k  // aqr = w = k (memory read)
aqw_long_ld  w,k  // aqw = w = k (memory write)
```

Each of the arguments is an expression that specifies a value as follows:

**w** a register number or a special register **wq** or **link**.
**k** any absolute number or relocatable offset.

### 3.10.10 Synthesized long jumps

Since the "load link immediate" instruction does not affect the condition codes, it can be used as a prefix to a conditional jump in order to jump conditionally to an arbitrary address. Bas provides the following synthesized long jumps that assemble into a sequence of two machine instructions:

```
long_call k  // link = nextpc; goto k
long_j    k  // goto k
long_jz   k  // if (ZERO) goto k
long_jm   k  // if (MINUS) goto k
long_jc   k  // if (CARRY) goto k
long_jnz  k  // if (!ZERO) goto k
long_jnm  k  // if (!MINUS) goto k
long_jnc  k  // if (!CARRY) goto k
long_j0   k  // class 1 jump 0 alu=k
long_j1   k  // class 1 jump 1 alu=k
long_j2   k  // class 1 jump 2 alu=k
long_j3   k  // class 1 jump 3 alu=k
long_j4   k  // class 1 jump 4 alu=k
long_j5   k  // class 1 jump 5 alu=k
long_j6   k  // class 1 jump 6 alu=k
long_j7   k  // class 1 jump 7 alu=k
```

The argument k is an expression that specifies a value as follows:

**k** any absolute number or relocatable offset.

For the class 1 jumps, these instructions assemble to produce k as the output of the ALU, just like a normal jump would do. This may or may not be what you want.

### 3.10.11 Simulator control

Bas provides the following instruction as a run-time interface to the simulator:

```
simctrl s  // simulator control s
```

The argument s is an expression that specifies a value as follows:

**s** an absolute number 0..31.

This instruction assembles as Rw=0, Ra=0, Rb=0, const=0, count=s, Fun=OR, Op=NOSHIFT. Observe that in the Beehive CPU architecture this is equivalent to

```
or $0,$0,$0
```

because the count field is irrelevant in such an instruction. However, the simulator notices this instruction and takes special actions based on s. See Section 5.5 for a discussion of the simulator controls.

## 3.11 Directives

A directive consists of an opcode possibly followed by some arguments. Although lexically an opcode is an identifier, it is not in the same namespace as predefined and user defined identifiers. In order to make clear which opcodes are instructions and which are directives, directive opcodes start with a period. The various directives are described next.

### 3.11.1 Segment selection directives

Bas implements three kinds of segments: code, data, and absolute. Code segments are intended to contain code and data segments are intended to contain data. An absolute segment is intended to provide for the layout of data without defining its contents. Labels get the kind of the segment in which they are defined. There can be multiple code segments and multiple data segments. There is only one absolute segment. The code address rotation controls how the current location advances in code segments. The data address rotation controls how the current location advances in data segments and in the absolute segment. Assembled instructions and data words can be emitted into code segments and data segments regardless of the kind of the segment. However, nothing can be emitted into the absolute segment.

The general directive for changing the current segment is:

```
.section n,s     // switch to segment n characteristics s
.section n,s,b   // switch to segment n characteristics s option b
```

The arguments are as follows:

**n** an identifier that is the name of the new segment.
**s** a string that defines characteristics of the new segment.
**b** an identifier giving an additional option.

The string *s* is interpreted character-by-character and each character specifies a characteristic as follows:

**a** all labels in this segment should be retained for debugging.
**w** segment is writable.
**x** segment is executable.
**s** segment is small.
**S** segment contains strings.
**T** segment is thread-local storage.

Characteristics other than "x" are ignored by Bas. Bas interprets characteristic "x" to specify a code segment. The absence of "x" specifies a data segment.

The optional argument *b* specifies options as follows:

**@nobits** the segment contains no initialized contents.
**@progbits** the segment may contain content.

The `@nobits` option is used, for example, to specify a `.bss` segment. The option is ignored by Bas.

When changing the current segment, if the new named segment does not already exist it is created. Otherwise, Bas merely switches to the existing segment and extends it.

The following abbreviated directives can be used to change to the default code and data segments:

```
.code  // switch to default code segment .text
.data  // switch to default data segment .data
.bss   // switch to secondary data segment .bss
```

Finally, the following directive changes to the absolute segment:

```
.abs i  // switch to absolute segment location i
```

The argument i is an expression that specifies a value as follows:

**i** any absolute number.

The absolute segment does not have a name. Although words cannot be emitted into the absolute segment, its current location can be advanced. This makes it convenient to define absolute labels in laying out a structure.

### 3.11.2 Advance current location directives

The following directive advances the current location within the current segment:

```
.blkw i  // advance current location by i * step
.blkb i  // advance current location by i bytes
```

The argument i is an expression that specifies a value as follows:

**i** any absolute number.

.blkw advances by a number of words and .blkb by a number of bytes.

The treatment of .blkb depends on the current segment's step. If the step is 4, implying a byte-addressed segment, .blkb advances the current location by i. If the step is 1, implying a word-addressed segment, .blkb advances the current location by (i + 3) / 4, which is the number of words it would take to store i bytes.

### 3.11.3 Align current location directives

The following directives advance the current location within the current segment, if necessary, until it has a specified alignment:

```
.align  i  // advance current location until it is 0 mod i
.alignw i  // advance current location until it is 0 mod (i * step)
```

The argument i is an expression that specifies a value as follows:

**i** any absolute number that is a power of two

Note that the current location need not be on a word boundary when current segment's step is 4. This can result from use of the .abs, .blkb, .byte, .string, or .ascii directives, for example. The .align directive is used to reestablish a desired alignment.

### 3.11.4 Emit words or bytes directives

The following directives emit words or bytes into the current segment:

```
.word  k,k,k,...  // emit words
.long  k,k,k,...  // alias for .word
.byte  k,k,k,...  // emit bytes
.2byte k,k,k,...  // emit bytes in chunks of 2
.3byte k,k,k,...  // emit bytes in chunks of 3
.4byte k,k,k,...  // emit bytes in chunks of 4
.short k,k,k,...  // alias for .2byte
```

Each of the arguments k is an expression that specifies a value as follows:

**k** any absolute number or relocatable offset.

Multiple arguments separated by commas may be specified.

Emitting bytes gets very special treatment from the assembler. The ".byte" directive emits one byte for each argument, the ".2byte" directive emits two bytes for each argument, the ".3byte" directive emits three bytes for each argument, and the ".4byte" directive emits four bytes for each argument. The emitted bytes are taken from the argument value starting with its least significant byte. Argument values that require relocation *are permitted*. Any part of the argument value that is not emitted is just ignored.

If the current segment's step is 4, meaning that it is a byte-addressed segment, each byte is emitted at a consecutively higher byte address just as you might expect. However, if the current segment's step is 1, meaning that it is a word-addressed segment, the assembler groups the sequence of bytes into chunks of four, arranges each chunk into a word value, and emits the chunks into successive words.

Note that ".4byte" is not the same as ".word" because the former emits four bytes per argument regardless of where the word boundary falls, whereas the latter always checks for word alignment.

If you want to know why I had to implement all these crazy directives, gcc uses them when writing debugging information. Emitting byte values that require relocation can generate many relocation patches in the output file.

### 3.11.5 Emit string directives

The following directives emit a string into the current segment:

```
.ascii  z // emit string
.string z // emit string with null terminator
```

The argument z is a string of any length. The characters in the string are emitted in order effectively using .byte directives. In the case of .string an additional zero byte is emitted at the end.

### 3.11.6 Region nesting directives

Bas maintains a current region name as it processes the source input. Any identifier that starts with a "." (except for "." itself) is implicitly prefixed by the current region name. This permits labels and symbols to be abbreviated locally in a region. Regions can be nested.

```
.enter name  // enter region
.leave name  // leave region
```

The argument name is an identifier that is used to name the region. Note that if this identifier starts with "." it will itself be subject to the implicit prefix transformation. To make this work with nested regions, in both .enter and .leave the argument belongs to the enclosing region.

Opcodes are immune to the implicit prefix transformation.

### 3.11.7 Assume register directives

The utility of the opcodes that synthesize instructions is greatly enhanced if some registers can be assumed to contain a known value, for example, zero. This is specified by the following directives:

```
.assume    r,k  // henceforth assume r = k
.noassume  r    // henceforth contents of r is unknown
```

Each of the arguments is an expression that specifies a value as follows:

**r** a register number.
**k** any absolute number or relocatable offset.

The .assume applies to all subsequent source input lines until cancelled by a .noassume.

### 3.11.8 Global symbol directive

A symbol is declared as global using the global symbol directive:

```
.globl n  // declare symbol n as global
```

If the symbol is defined in the current assembly, then its name and definition is made available to the relocating loaded to bind external references. The definition must be an absolute number or a relocatable offset. If the symbol is not defined in the current assembly, then it signifies an external reference to a global symbol. The `-x` option causes all otherwise undefined symbols to be declared as global, unless they are specifically declared as local.

### 3.11.9 Local symbol directive

A symbol is declared as local using the local symbol directive:

```
.local n  // declare symbol n as local
```

The purpose of declaring a symbol as local is to override the presumption of the `-x` option with regard to a common request.

### 3.11.10 Common request directive

A symbol is requested to be defined as the base address of a common area using the common request directive:

```
.comm n,s,a  // request a common area
```

The common request directive requests that the symbol “n” be defined as the base address of a common area of size “s” bytes with alignment “a”. The common area will be of kind “data” with the same address rotation as the data segment. The alignment “a” may be omitted, which case it defaults to the step of the data segment.

If the symbol “n” is declared as local, then this is a local common request, and it will be satisified by allocating space at the end of the data segment. The definition of “n” will be local and no other object file will be able to see it.

Otherwise, if the symbol “n” is declared as global, then this is a global common request. Global common requests of the same symbol made different object files are combined into a single request by taking the maximum of the requested sizes and the maximum of the alignment requirements.

### 3.11.11 Include directive

A file can be incorporated into the source stream using the include directive:

```
.include z  // include file “z” at this point
```

This file name z is a string. Each of the specified directories of include files is searched in order to find the indicated include file. The `-I` option is used to specify an include file directory.

### 3.11.12 Commentary directives

The gcc compiler emits various directives related to debugging. For the present, the following directives are ignored:

```
.size  n,k   // declare size of symbol n to be k
.type  n,... // declare type of symbol n
.file  s     // file name s
.ident s     // compiler identification s
```

## 3.12 Predefined symbols

Bas manages a collection of predefined symbols. Some of these symbols have values that change as assembly progresses (which is not possible for user defined symbols). The predefined symbols are as follows:

**pc** the program counter special register, read via Ra overload 31
**link** the link special register, read via Ra overload 30 and written via Rw overload 30
**rq** the read queue special register, read via Ra overload 29

**wq** the write queue special register, written via Rw overload 31
**.** the current location in the current segment
**code.rota** the code segment address rotation (an absolute number)
**data.rota** the data segment address rotation (an absolute number)
**code.step** address offset between words in the code segment (an absolute number)
**data.step** address offset between words in the data segment (an absolute number)

# Chapter 4

# Utilities

Various utility tools manipulate relocatable object files. The archiver collects relocatable object files into a library archive. The loader binds relocatable object files into an executable object file. The image maker converts an executable object file into a binary memory image. The Virtex mem file maker convers an executable object file into Virtex mem files. There is also a utility to process a file of bytes into C source that can then be compiled into a relocatable object file. These utilities are described next. The format of the object files is described in Appendix B.

## 4.1 Archiver - Bar

Bar combines a number of relocatable object files into a library archive. Bar is invoked using the command line

```
Bar command archive.a modulea.o moduleb.o ...
```

Bar adds the modules to the archive, creating the archive first if necessary. The command is a string of characters, each of which is interpreted separately. The following command characters are supported:

**r** ("Replace") Add new modules to the end of the archive, deleting any existing ones with the same names.

**q** (So-called "quick") Add new modules to the end of the archive, without deleting any existing ones that might have the same names.

**c** ("Create") Expect to create the archive. The archive is always created if it does not already exist, but a warning is issued in such a case unless this command is specified.

**v** ("Verbose") Give additional commentary on the actions taken.

Exactly one of "r" or "q" must be given. Any number of relocatable object files may be provided as input. You can also provide an archive as input, in which case all of the relocatable object files it contains are added to the archive being constructed.

The command line format conforms to that of the gnu archiver so that the gnu tool chains can use it.

## 4.2 Loader - Bld

Bld is a relocating loader. Bld reads a number of relocatable object files and archive files, resolves external symbol references, concatenates same-named segments, and outputs an executable object file. The format of the object file is described in Appendix B.

Bld is invoked using the command line

```
Bld [options] module.o library.a ...
```

Any number of relocatable object files and library archive files may be provided. The files are incorporated into the final executable in the order in which they appear on the command line. If the file is a relocatable object file, it is incorporated without further ado. If the file is a library archive file, its constituent object files are scanned to determine if any satisfies an external symbol reference and, if so, that object file is incorporated. If any object file is incorporated from a library archive, the archive is rescanned to see if additional object files need to be incorporated.

Options start with a hypen (`-`). The following options control relocation and are processed in the order they appear on the command line:

**-codebase=*n*** Specify that code segments will be relocated to start at memory word index *n* (in hex). If there are multiple code segments, they are relocated in succession to start at the memory word index after the previously relocated code segment, in the order in which they are encountered as the input files are processed. Note that the memory word index is independent of the segement address rotation.

**-database=*n*** Specify that data segments will be relocated to start at memory word index *n* (in hex). If there are multiple data segments, they are relocated in succession to start at the memory word index after the previously relocated data segment, in the order in which they are encountered as the input files are processed. Note that the memory word index is independent of the segement address rotation.

**-codefloat** Specify that the code segments will be relocated to start at a memory word index after the previously specified kind of segment (if any).

**-datafloat** Specify that the data segments will be relocated to start at a memory word index after the previously specified kind of segment (if any).

The following options specify the name of various output files:

**-o *a.out*** Specify that the output executable object file will be named *a.out*

**-map *a.map*** Create a map file *a.map* listing the base word index and size of segments and the definitions of global symbols.

The following options control the loading of libraries. They can occur anywhere in the command line and are processed in order with other files that are loaded.

**-L*dir*** Add *dir* to the library search path.

**-l*xxx*** Find the library archive file *libxxx.a* on the library search path and load it as described above.

The following options control how *common* symbols are processed. A common symbol is created with a `.comm` assembler directive or with an uninitialized global non-extern variable declaration in C. Common symbols are weird in that multiple object files may declare them. The multiple declarations are combined by taking the largest length and most restrictive alignment requirement.

**-commbss** Specify that common symbols should be allocated in the `.bss` segment. This is the default.

**-nocommbss** Specify that common symbols should be allocated by creating a special data segment for each one. The segment is named by prefixing "`comm-`" to the name of the common symbol. This is the way the loader used to work.

The following options control how debugging segments are processed. When GCC is given the `-g` option it generates debugging information into data segments whose names start with "`.debug_`". Such segments are called debugging segments.

**-debugseg** Specify that debugging segments should be changed to have kind "debug", which causes them to be relocated separately from code and data segments and, more significantly, *not* to appear in the memory image created by Bimg or Bsim. This is the default.

**-nodebugseg** Specify that debugging segments should be left as data segments. This is the way Bld used to work.

## 4.3 Image maker - Bimg

Bimg is the image maker. Bimg reads an executable object file, constructs a memory image by applying all specified patches, and outputs the result as a binary memory image file. The binary image file contains a consecutive range of memory words, starting with the word at the lowest memory word index loaded by the executable object file and continuing through the highest memory word index loaded by the executable object file.

Bimg is invoked using the command line

```
Bimg [options] a.out
```

One executable object file *a.out* is expected. Options start with a hypen (`-`). The following options are provided:

**-img *a.img*** Specify that the output binary memory image file will be named *a.img* instead of the default, which is to take the name of the input executable object file and replace the extension with `.img`

## 4.4 Virtex mem file maker - Bvmem

Bvmem makes Virtex mem files that can be used to initialize the cache memory of a Beehive design. Bvmem reads an executable object file, constructs separate code and data memory images by applying all specified patches, and outputs the results in Virtex mem file format. Each resulting Virtex mem file contains a consecutive range of memory words starting with word index zero and continuing through the last word loaded by the object file into that memory image. The format of these mem files is described in the Data2MEM Users Guide [2].

Note that the Beehive has separate code and data caches. Each cache is initialized to contents that appear to have been fetched from physical memory word indexes zero through `0x3ff`. However, since the caches are separate, the initialized contents of the code and data caches are independent. The loader can relocate both the code segment and the data segment to start at word index zero, which is the proper arrangement for constructing Virtex mem files.

Bvmem creates two Virtex mem files, one for code and one for data. Given an input file *a.out* by default the resulting code Virtex mem file is named *acode.mem* and the resulting data Virtex mem file is named *adata.mem*. The extension `.mem` is required by the Virtex tools.

Bvmem is invoked using the command line

```
Bvmem [options] a.out
```

One executable object file *a.out* is expected. Options start with a hypen (`-`). The following options are provided:

**-mem *b.mem*** Specify that the Virtex mem files will be named *bcode.mem* and *bdata.mem* instead of the default, which is to take the name of the input executable object file and replace the extension with `code.mem` and `data.mem`

## 4.5 File of bytes processor - Bfiledata

Bfiledata takes a file of bytes *file.dat* and produces a C source file *file.c* that defines and initializes two global symbols, *file* and *file_cnt*. A header file *file.h* is also produced that contains external definitions for these symbols. The global symbol *file* is defined as an array of bytes and is initialized to the contents of the file of bytes. The global symbol *file_cnt* is defined as an int and is initialized to the number of bytes in the array. Bfiledata is invoked using the command line

```
Bfiledata [options] file.dat
```

Options start with a hypen (`-`). The following options are provided:

**-c *b.c*** Specify the name of the output C source file. The default is *file.c*

**-h *b.h*** Specify the name of the output C header file. The default is *file.h*

**-gbl *b*** Specify the name of the global that is initialized to the array of bytes. The default is *file*

**-cnt *b_cnt*** Specify the name of the global that is initialized to the size of the array. The default is *file_cnt*

# Chapter 5

# Simulator - Bsim

Bsim is a simulator for Beehive [1]. Bsim reads an object file, initializes a simulated memory image, and then simulates Beehive instructions beginning at the address of `main`. The format of the object file is described in Appendix B. The simulator implements the full physical memory space and a selectable number of normal cores, each with a full instruction set, instruction and data caches, debug unit, and most of the coprocessors.

## 5.1 Running Bsim

Bsim is invoked using the command line

```
Bsim [options] program.out
```

Options start with a hypen (`-`). The following options are supported:

**-trace** Turn on instruction trace mode before starting the simulation. Instruction trace mode can also be turned on or off during simulation by using the simulator control instructions described in Section 3.10.11.

**-debug** Activate the interactive debugger on core 1. The interactive debugger is described in Section 5.6.

**-ncore=**$n$ Set the number of normal cores to $n$. The default is 1 normal core. Note that all cores start execution at the same start address, either zero if there are any cache preload requests, or otherwise the global symbol `main`. Special startup code must be used to allocate a separate properly-aligned stack to each core.

**-megastepmax=**$n$ Set the maximum number of cycles that the simulator will execute to $n$ million. Note that this number is in millions. It takes the simulator about a second per core to simulate a million cycles. The default is 0 which means no limit.

**-cachestat** Print out cache statistics for each core and the end of simulation. This is the default.

**-nocachestat** Do not print out cache statistics for each core and the end of simulation.

**-icache=**$n$ ***file.mem*** Add icache core $n$ *file.mem* to the list of cache preload requests.

**-icache *file.mem*** Add icache core 0 *file.mem* to the list of cache preload requests.

**-dcache=**$n$ ***file.mem*** Add dcache core $n$ *file.mem* to the list of cache preload requests.

**-dcache *file.mem*** Add dcache core 0 *file.mem* to the list of cache preload requests.

## 5.2 Cache preload requests

The simulator supports preloading of instruction and data caches. This is requested by the -icache and -dcache options. There may be more than one request. The requests are processed in order, after the requested number of cores has been created and physical memory has been initialized.

Note that if there are any cache preload requests, the simulator will start execution of all cores at location zero. This corresponds to a reset in the hardware.

A cache preload request consists of (1) a specification of icache or dcache, (2) a core number, and (3) a file. If the core number is zero it is interpreted as applying to all cores, otherwise the preload request applies only to the indicated core, if present. The file is expected to be in vmem format such as created by Bvmem.

Preloading a cache initializes the contents of the cache as if the specified data had been fetched from addresses 0 through 0xfff. The cache lines are marked as valid and not dirty. The data is taken from the first 1024 words of the mem file. If the mem file contains fewer words, it is padded with zeros.

As an example, the following command preloads core 1 with “master” code and all other cores with “slave” code:

```
Bsim -icache slave.mem -icache=1 master.mem program.out
```

## 5.3 Physical memory

The simulator implements the full physical memory space of 0x80000000 words. However, actually trying to use the entire memory space will likely cause the simulator to exhaust its resources.

The simulator implements both the data cache and the instruction cache. Unless they have been preloaded, both caches are initially entirely invalid.

Data accesses are performed more rapidly than in the hardware, especially with regard to taking cache misses. The simulator is not cycle exact with respect to data accesses. An attempt to access a nonexistent memory address produces an error message.

The simulator supports all address rotation options. The code and data rotations are specified in the object file.

| supported | coprocessor |
|---|---|
| yes | 0: ASLI interface coprocessor (see Appendix A.6.2) |
| no | 1: multiply coprocessor |
| no | 2: miscellaneous output signal coprocessor |
| yes | 3: data cache controller (see Appendix A.6.3) |
| yes | 4: message controller (see Appendix A.6.4) |
| yes | 5: lock controller (see Appendix A.6.5) |

Table 5.1: Simulated coprocessors

## 5.4 Simulated coprocessors

Table 5.1 summarizes the simulation support for the various Beehive coprocessors. At the start of simulation, the cycle counter is set to zero, all lines in the data cache are invalid, all receive message queues are empty, and no core holds any lock.

The simulator connects the ASLI interface register to the console. If the simulator is not run from a console (for example, when run from inside an emacs shell), then input is not possible due to deficiencies in Windows. In such a case output will happen normally but it will appear that the receiver never has a byte ready to read.

The simulator counts cycles starting with zero at the beginning of the simulation. A taken jump adds an extra cycle to account for the post jump nullify. Even though the simulated memory system is fast, there may also be stalls due to memory access.

## 5.5 Simulator controls

The simulator takes special notice of any instruction which has const=0 and Fun=OR. After interpreting such an instruction, it interprets the count field (which is unused in this instruction by the Beehive CPU architecture) as a special control. See the simctrl instruction in Section 3.10.11 for how to create such an instruction in the assembler. The controls are:

**0** no operation
**1** exit simulator (normal termination)
**2** start tracing instructions onto the console output
**3** stop tracing instructions
**4** dump register file onto the console output
**5** exit simulator (abnormal termination)
**6–31** reserved

## 5.6 Interactive debugger

The simulator supports an interactive debugger attached to core 1. The debugger is activated using the `-debug` option. When active, the debugger breaks before the first simulated instruction and accepts commands from the console input. Simulation of all

| | |
|---|---|
| `<cr>` | (empty line) run until break |
| `i` | implicit break on each instruction |
| `s` | implicit break on new subroutine |
| `g` | run without implicit breaks |
| `dm` *`addr count`* | dump *count* words of data memory |
| `cm` *`addr count`* | dump *count* words of code memory |
| `r` | dump registers |
| `trace y` | trace each instruction |
| `trace n` | do not trace each instruction |
| `q` | quit |
| `h` | print help message |

Table 5.2: Debugger commands

cores is frozen while the debugger is accepting commands. The debugger commands are summarized in Table 5.2. Each debugger command occupies an input line.

The debugger inspects each instruction (on core 1) before it is executed. The debugger can trace each instruction before execution or it can trace the instruction only when it breaks. Tracing means that the debugger prints out the current program counter and pending instruction in symbolic form. For example, the initial break typically produces a trace output such as

```
main+00000000:  $27 = $27 ANDN $27
```

This means that the current program counter is at offset 0 from global symbol “main” and the pending instruction is an ANDN that stores zero into register 27.

Note that the instruction printed in an instruction trace comes from the physical memory, as opposed to the code cache on core 1. This may be fixed in the future.

The debugger gets symbol definitions from the input files. Only global symbols are considered. Symbols are classified as code symbols or data symbols depending on which kind of segment they are defined in. The debugger assumes that a subroutine extends from one code symbol to the closest next one defined at a higher address.

The debugger can perform an implicit break on each instruction or on each change of subroutine. The execution of an exit-simulator instruction (see Section 5.5 also causes a break. Currently there is no provision for setting break points.

The debugger can dump words from code memory or from data memory. Note that these printouts come from physical memory rather than from the code and data caches of core 1. This may be fixed in the future. The address to dump must be specified in hexadecimal. This may also be fixed in the future. The address is interpreted according to the relevant address rotation.

# Appendix A

# Beehive architecture

The Beehive architecture is based on a 32-bit word. It has a register file containing 32 registers, a two-input ALU followed by a full barrel shifter, and an unusual queued interface to a memory controller for access to data memory and memory mapped IO. Instruction space and data space may be considered as separate during execution of pre-initialized cache contents. (The Beehive hardware manual [1] should be consulted for further details.) In addition to the register file, there is a program counter register, a link register (for subroutine linkage and constant assembly), and a condition code register. All instructions have the same format, as shown in Figure A.1.

## A.1 ALU function

Almost all instructions select two arguments, A and B, for the ALU, which performs a function determined by the Function field:

**0** A + B
**1** A - B
**2** A & B
**3** A & ~B
**4** A | B
**5** A | ~B
**6** A ^ B
**7** A ^ ~B

### A.1.1 ALU argument A

Argument A is specified by the Ra field. In most cases, Ra selects a register from the register file. However, certain values of Ra are overloaded. The Ra overloads are:

**29** the read queue (takes one word; stalls until read queue is nonempty)
**30** the link register
**31** the program counter register (address of the current instruction)

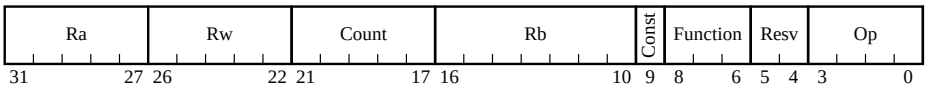



Figure A.1: Beehive instruction format

### A.1.2 ALU argument B

When Const = 0, argument B is specified by the Rb field, which selects a register from the register file. Rb $>$ 31 is reserved.

When Const = 1, argument B is generated as a constant assembled from the instruction in a mode determined by the Op field. There are three modes: RbConst, CountRbConst, and RwCountRbConst.

**RbConst** the constant is the Rb field (7 bits).
**CountRbConst** the constant is the concatenation of the Count and Rb fields (12 bits).
**RwCountRbConst** the constant is the concatenation of Rw[3:0], Count, and Rb fields (16 bits). Note that in Beehive CPU version 2 the high order bit of the Rw field is not included in the constant.

In all cases the constant is padded on the left with 0s to fill out the 32-bit word.

## A.2 Major Operation

The Op field determines the constant mode, the shift mode, and various major effects of the instruction, as follows:

**0** RbConst, logical shift right by Count bits, write result in Rw
**1** RbConst, logical shift left by Count bits, write result in Rw
**2** RbConst, rotate right by Count bits, write result in Rw
**3** Load link immediate (see below)
**4** RbConst, arithmetic shift right by Count bits, write result in Rw
**5** CountRbConst, no shift, write result in Rw
**6** CountRbConst, no shift, write result in Rw and into address queue as a write command
**7** CountRbConst, no shift, write result in Rw and into address queue as a read command
**8** RwCountRbConst, no shift, jump operation (see below)
**9** RwCountRbConst, no shift, jump operation (see below)
**10** RwCountRbConst, no shift, jump operation (see below)
**11** RwCountRbConst, no shift, jump operation (see below)
**12** RwCountRbConst, no shift, jump operation (see below)
**13** RwCountRbConst, no shift, jump operation (see below)
**14** RwCountRbConst, no shift, jump operation (see below)
**15** RwCountRbConst, no shift, jump operation (see below)

| Rw[4] | Op | jump operation |
|---|---|---|
| 0 | 8 | call: link = nextpc, jump always |
| 0 | 9 | jump if minus |
| 0 | 10 | jump if zero |
| 0 | 11 | jump if carry |
| 0 | 12 | jump always |
| 0 | 13 | jump if not minus |
| 0 | 14 | jump if not zero |
| 0 | 15 | jump if not carry |
| 1 | 8 | class 1 jump 0 |
| 1 | 9 | class 1 jump 1 |
| 1 | 10 | class 1 jump 2 |
| 1 | 11 | class 1 jump 3 |
| 1 | 12 | class 1 jump 4 |
| 1 | 13 | class 1 jump 5 |
| 1 | 14 | class 1 jump 6 |
| 1 | 15 | class 1 jump 7 |

Table A.1: Jump operations

Many of the operations write the result of the shifter into register Rw in the register file. In these cases, if Rw = 31 the result is also written into the write queue or if Rw = 30 the result is also written into the link register.

The load link immediate operation copies the instruction into the link register, replacing the low order 4 bits with zero. All other effects (condition code update, queue reads and writes, and register file writes) are suppressed.

The jump operations suppress condition code update, queue writes, and register file writes. (But notably a jump does not suppress queue reads, so it is possible to take a subroutine return address from the read queue and jump to it.) In Beehive CPU version 2, the high order bit of the Rw field (Rw[4]) is not used for the constant and is instead combined with the op field to specify the particular jump operation. Table A.1 shows the specifications.

The jumps are divided into class 0 jumps and class 1 jumps. Class 0 jumps are the ordinary jumps, including call. Class 1 jumps are intended for interpretation by special function units and may do special things. At present, the only defined special function unit is the debug unit (see Appendix A.5.1) which interprets class 1 jump 7.

## A.3 Condition codes

Condition codes are updated at the end of an instruction (unless suppressed) and therefore always reflect the result of a previous instruction. Exactly those operations that write the result of the shifter into Rw are those that update the condition codes. The result of the shifter determines ZERO and MINUS. The result of the ALU function determines CARRY, which is undefined except for ADD and SUB.

## A.4 Reserved

When Const = 0, ALU argument B is specified by the Rb field, which selects a register from the register file. Rb $>$ 31 is reserved.

Except for the load link immediate instruction, the Resv field is reserved and must always contain 0.

Class 1 jumps 0 through 6 are not at present defined and must not be used.

## A.5 Special function units

The class 1 jumps 0 through 7 are interpreted by corresponding special function units 0 through 7. A class 1 jump instruction resembles a normal jump instruction in that the A and B arguments are fetched, which may include pulling a word from the read queue, and an ALU result is computed without modifying the condition codes. However, the special function unit may examine the instruction and cause special things to happen to the Beehive CPU.

At present, the only special function unit 7 (the debug unit) is defined.

### A.5.1 Debug unit

The debug unit is designed to permit core 1 (called the master core) to control and debug the other Beehive CPU cores (called the slave cores). Each Beehive CPU core has a debug unit, although the debug unit on core 1 is of little use.

The debug unit maintains two state bits, *running* and *wantstop*, and two word registers, *savedPC* and *savedLink*. When a slave core takes a breakpoint or is stopped, the PC is saved in savedPC, the link is saved in savedLink, running and wantstop are cleared, and the PC is set to zero. It is assumed that appropriate code appears starting at location zero that will save the CPU state in memory and then wait for a command from the master core.

The Beehive CPU state is complicated, consisting of register values, the condition codes, and the address, read, and write queues. In Beehive CPU version 2, there is no provision to save and restore the condition codes, the address queue, or the write queue. Consequently, breakpoints can legally be inserted only at points where the address and write queues are empty and the condition codes are immaterial.

The debug unit interprets class 1 jump 7 instructions. The debug unit disregards the result of the ALU and examines the low-order three bits of the register b field in the instruction to determine what special things to do, as follows:

**rb=0** This instruction has no effect, except possibly for pulling a word from the read queue.

**rb=1** The link is loaded with the data address of a “savearea” in which it is intended that the Beehive CPU state for this core be saved. The savearea data address for core N is 0x4000 + 512 * N.

**rb=2** The link is loaded with the contents of the debug unit’s savedPC register.

**rb=3** The link is loaded with the contents of the debug unit's savedLink register.

**rb=4** The link is loaded with 1 if the read queue is not empty, and with 0 otherwise.

**rb=5** The link is loaded with 1 if the debug unit's running bit is set, and with 0 otherwise.

**rb=6** The savedPC register is loaded with PC+1, the savedLink register is loaded with link, the running bit is cleared, the wantstop bit is cleared, and the PC is loaded with 0. The instruction at location 0 should be a nop, because due to pipeline issues the Beehive CPU does not guarantee that it will actually be executed.

**rb=7** Not defined.

The debug unit also interprets control messages received by the inter-core message controller. As described in Appendix , control messages have zero words of payload and are interpreted immediately upon arrival, rather than being enqueued on the message queue. The debug unit's control messages are as follows:

**start** This control message is sent by the master core (src=1) with type=0 and it causes the debug unit to set the running bit. It is assumed that the location zero code in the slave core is looping waiting for the running bit to be set, whereupon it will restore the CPU state from the savearea and resume normal execution.

**stop** This control message is sent by the master core (src=1) with type=1. If running is set, the debug unit will set the wantstop bit, otherwise there will be no effect. As described below, the wantstop bit causes the debug unit to watch program execution and interrupt it at a point where the address and write queues are empty and the condition codes are irrelevant.

**kill** This control message is sent by the master core (src=1) with type=2. If running is set, the debug unit will immediately reset the address and write queues and force an interruption of program execution, otherwise there will be no effect.

When the debug unit interrupts program execution, it clears running and wantstop, loads savedPC and savedLink from PC and link, and loads PC with zero. Note that the savedPC is the PC of the first instruction not executed.

When wantstop is set, the debug unit examines the stream of instructions executed by the CPU looking for a legal place to interrupt. This happens when the address and write queues are empty and the CPU is about to execute an instruction that sets the condition codes. Instead of executing this instruction, the CPU is interrupted.

## A.6 Memory controller

The processor communicates with the memory controller via three queues: the address queue, the write queue, and the read queue.

The memory controller processes commands in order from the address queue. A write command requires also a word from the write queue, which is then stored in

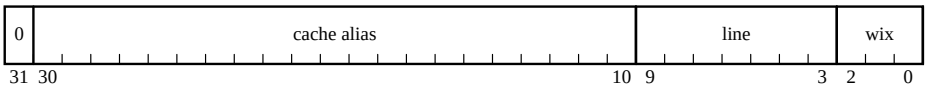



Figure A.2: Memory space address format

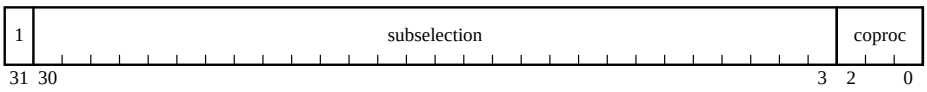



Figure A.3: Coprocessor space address format

the specified data address. A read command causes the word to be fetched from the specified data address and then placed in the read queue.

The processor accesses the write queue via an Rw overload. Each time an instruction specifies an unsuppressed write of Rw=31, the result of the shifter is placed onto the write queue. If necessary, the processor stalls until the write queue is non-full.

The processor accesses the read queue via an Ra overload. Each time an instruction specifies an unsuppressed read of Ra=29, the value is taken from the read queue. If necessary, the processor stalls until the read queue is non-empty.

Access to the address queue is specified via the major operation. The output of the shifter is rotated right by the data segment's address rotation and the result is placed onto the address queue along with the indication of whether it is a read command or a write command. Note that the address queue always contains word addresses, regardless of the data segment's address rotation.

In the Beehive CPU version 2, there is *no hardware check* to prevent the address queue from overflowing. If this happens the behavior is undefined. Hence it is the software's responsibility to avoid such a situation.

Note that some requests on the address queue may take a long time to complete. The worst case is probably a complete data cache flush when the data cache is entirely dirty. The only way the software can assure that the address queue is draining is by issuing a read request and waiting for the read data to come back on the read queue.

### A.6.1 Address queue value

Each value on the address queue refers either to memory space (when bit 31 is 0) or coprocessor space (when bit 31 is 1). Note that the shifter result that the CPU has to generate in order to create an address queue value depends on the data segment's address rotation.

Figure A.2 shows the format of a memory space address queue value. The low order three bits give the word index in a cache line. The next seven bits give the cache line number in the data cache. The next 21 bits distinguish different cache aliases. Finally, bit 31 must be 0.

Figure A.3 shows the format of a coprocessor space address queue value. The low order three bits select the specific coprocessor. The next 28 bits are provided to the

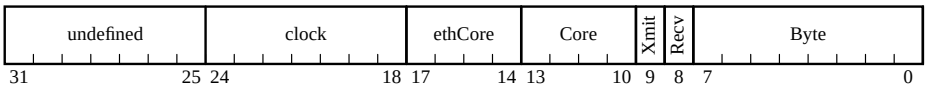



Figure A.4: ASLI interface register format

coprocessor for subselection. The coprocessor can use the subsection bits in any way it wants. Finally, bit 31 must be 1.

The coprocessors numbers are defined as follows:

**0** ASLI interface
**1** multiply coprocessor
**2** miscellaneous output signal controller
**3** data cache controller
**4** inter-core message controller
**5** lock controller
**6** undefined
**7** undefined

### A.6.2 ASLI interface

Coprocessor 0 is the ASLI interface. It contains two registers: an ASLI interface register, subselected when address queue value bit 3 is 0, and a cycle counter register, subselected when address queue value bit 3 is 1. These registers are described next.

Figure A.4 shows the format of the ASLI interface register.

Reading the ASLI interface register gives the following status:

**clock** the system clock speed in MHz
**ethCore** the core number of the EtherNet core
**Core** the core number of this core
**Xmit** 1 = the transmitter is ready for another byte
**Recv** 1 = the receiver has a byte ready to read
**Byte** the byte the receiver has ready, if any

The system clock speed for the BEE3 is 125 MHz and for the ML509 is 100 MHz. The simulator claims a clock speed of 2 MHz, which is roughly accurate. The core number of a normal core is in the range $1 \ldots \mathrm{ethCore} - 1$.

Writing the ASLI interface register has the following effects:

**Xmit** 1 = provide a byte to the transmitter, assuming it was ready
**Recv** 1 = acknowledge the byte from the receiver, assuming it was ready
**Byte** the byte provided to the transmitter, if any

Figure A.5 shows the format of the cycle counter register. Reading the cycle counter register gives the number of instruction cycles (counting both executed instructions and stalls) that have elapsed since some arbitrary initial point. Writing the cycle counter register has no effect.

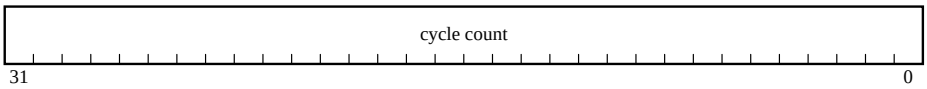
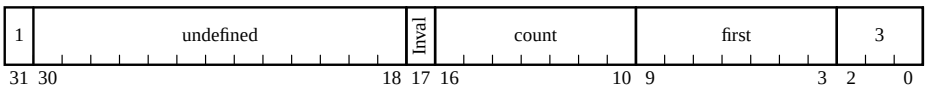



Figure A.5: Cycle counter register format




**Inval** 1 = invalidate, 0 = clean
**count** one less than the number of consecutive cache lines to process
**first** the number of the first cache line to process

Figure A.6: Data cache controller address queue write value format

### A.6.3 Data cache controller

Coprocessor 3 is the data cache controller. This coprocessor is unusual in that it is accessed using an address queue write with no words written on the write queue.

Figure A.6 shows the format of the data cache controller address queue write value. Note that bit 31 must be 1 to select coprocessor space and the low order three bits must contain 3 to select the data cache controller. Note that the shifter result that the CPU has to generate in order to create such an address queue value depends on the data segment's address rotation.

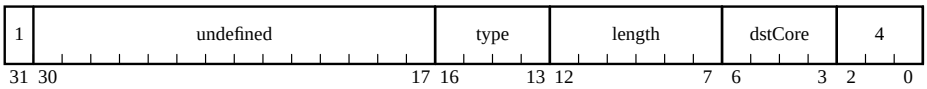
### A.6.4 Inter-core message controller

Coprocessor 4 is the inter-core message controller. This coprocessor is unusual in that the number of words written to the write queue (when sending a message) or read from the read queue (when receiving a message) depends on the contents of the message header.

Messages consist of a header and 0 to 63 words of payload. Note that zero words of payload is permitted, but it is interpreted as a control message by the receiving core's message controller and will not be enqueued on the message queue for software reception. Control messages are described in the discussion of the debug unit in Appendix A.5.1.

To send a message, you first write the payload words to the write queue. Then you write the header as an address queue write value whose format is shown in Figure A.7. Note that bit 31 must be 1 to select coprocessor space and the low order three bits must contain 4 to select the message controller. Note that the shifter result that the CPU has to generate in order to create such an address queue value depends on the data segment's address rotation.

To receive a message, you must poll by writing a receive request as an address




**type** the software message type, uninterpreted by hardware
**length** the number of words in the message payload
**dstCore** the destination core for the message

Figure A.7: Message controller address queue write value format to send a message

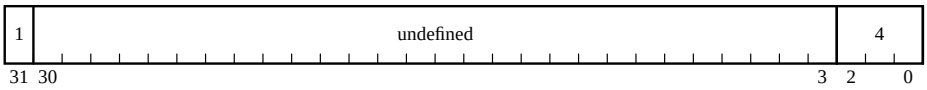



Figure A.8: Message controller address queue read value format to poll for a message

queue read value whose format is shown in Figure A.8. Note that bit 31 must be 1 to select coprocessor space and the low order three bits must contain 4 to select the message controller. Note that the shifter result that the CPU has to generate in order to create such an address queue value depends on the data segment's address rotation.

Then you read a status word from the read queue. The format of the status word is shown in Figure A.9. If the status word is zero, there is no message to be received at this time. Otherwise, the source core number will be non-zero, the message payload length will be non-zero, and the message payload words have been enqueued onto the read queue immediately after the status word. You must read the payload words from the read queue.

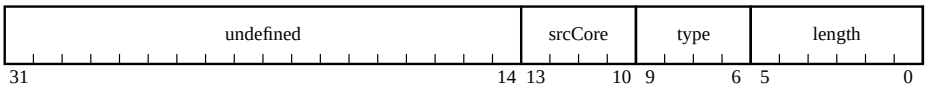
Note that the hardware provides a receive message queue at each core but no flow-control. The receive message queue is 1024 words long, which is large enough to hold a maximum length message (1 header word + 63 payload words) from each core. *Results are undefined* if a receive message queue overflows. Software must enforce a flow-control discipline so that this does not happen.

### A.6.5 Lock controller

Coprocessor 5 is the lock controller. This coprocessor is unusual in that no words are written to the write queue when loading an address queue write to release a lock.

A lock is conditionally acquired by writing an address queue read value and a lock is released by writing an address queue write value. In both cases, the address queue value has the same format, which is shown in Figure A.10. Note that bit 31 must be 1 to select coprocessor space and the low order three bits must contain 5 to select the lock controller. Note that the shifter result that the CPU has to generate in order to create such an address queue value depends on the data segment's address rotation.

When you write an address queue read value to conditionally acquire a lock, the lock controller enqueues a word onto the read queue that indicates the result of the




**srcCore** the core number that sent the message
**type** the software message type, uninterpreted by hardware
**length** the number of words in the message payload

Figure A.9: Message controller status word format


Figure A.10: Lock controller address queue value format

attempt. The value is zero to indicate a failed attempt, and non-zero to indicate a successful attempt. Attempting to acquire a lock which you already hold is always successful.

When you write an address queue write value to release a lock that the core currently holds, the lock is released. There is no effect if the core does not hold the lock. In any event, no words are removed from the write queue.

## A.7 Instruction fetch

Instructions are fetched as follows. The content of the pc is rotated right by the code segment's address rotation and the result is used as the word address of the instruction to fetch. Instructions cannot be fetched from coprocessor space.

# Appendix B

# Object file format

For convenience in software tool development, object files are represented as XML format text files. A relocatable object file consists of an `<object>` element. An archive file consists of an `<archive>` element. An executable object file is identical in format to a relocatable object file, except that it is presumed that no unbound external symbol references remain and that the global symbol "main" is defined to specify the program entry point.

Next we describe the elements in the document model in more detail.

## B.1 Archive element

The `<archive>` element represents an archive of relocatable object files. It contains a number of `<object>` subelements each of which represents a relocatable object file.

## B.2 Object element

The `<object>` element represents a relocatable object file. It contains a number of `<segment>`, `<local>`, `<globl>`, `<extrn>`, and `<comm>` subelements. Each segment subelement represents a segment within the relocatable object file.

The local, globl, extrn, and comm subelements relate to symbols within the relocatable object file. Each local subelement represents a local symbol defined within the object file. Local symbols are not used for binding external symbol references and need not be unique. Each globl subelement represents a global symbol defined withing the object file. Each extrn subelement represents an external symbol reference. Each comm subelement represents a global common request.

The `<object>` element has the following attributes:

**file** Name of the assembler source file that generated this relocatable object file.

**coderota** Address rotation for the code segment (in hex).

**datarota** Address rotation for the data segment (in hex).

## B.3 Segment element

The  element represents a relocatable memory segment. It contains a sequence of <w> (word), <z> (zero), <p> (patch), and <cp> (choice patch), subelements that specify the memory contents and relocation patches of the segment.

The word and zero subelements must be processed in sequence. The word subelement specifies the absolute value of the next word in the segment. The zero subelement specifies that the next several words in the segment have an absolute value of zero. If the segment ends with a number of zero words, there need not be a final zero element to specify them, since the isize segment attribute tells how many words are contained in the segment.

Once the absolute values of all words in the segment have been determined by the word and zero elements, the patch elements can be processed to apply relocation patches. Each patch element specifies the relative word index within the segment to which it applies. Multiple patches can be applied to the same memory word. The choice patch element is a generalization of the patch element and specifies a number of alternative patches to be attempted on a specified memory word.

The  element has the following attributes:

**name** Name of the segment.

**kind** Kind of the segment. There are only two possible kinds, “code” and “data”.

**ibase** Base memory word index for the segment (in hex).

**isize** Length of the segment in words (in hex).

**rota** Address rotation of the segment (in hex).

**alignbic** Mask specifying which bits in the base memory word index must be zero for proper alignment of this segment (in hex).

## B.4 Word element

The <w> element represents a memory word in a segment. It contains no subelements and has the following attributes:

**v** Absolute value of the word (in hex).

## B.5 Zero element

The <z> element represents a sequence of zero words in a segment. It contains no subelements and has the following attributes:

**c** Number of zero words (in hex).

## B.6 Patch element

The <p> element represents a *patch* in a segment. A patch specifies a value

$$( ( ref + off ) \,\&\, \sim bic ) \text{ ROL } rol$$

that is computed and then OR'ed into a specified word within the segment. The $ref$ is either a global symbol reference or the base address of a segment in the current relocatable object file. A global symbol reference can be a globl symbol defined within the current object file or an extrn symbol requested by the current object file.

The <p> element contains no subelements and has the following attributes:

**i** Relative word index within the segment to which this patch applies.

**sym** Name of a global symbol. Excludes the seg attribute.

**seg** Name of a segment within this relocatable object file. Excludes the sym attribute.

**off** Offset from the global symbol or segment base (in hex). The default is "0".

**bic** Mask of bits to clear after computing the relocated offset (in hex). The default is "0".

**rol** Number of bit positions to rotate left after masking (in hex). The default is "0".

## B.7 Expression patch element

The <ep> element represents an *expression patch* in a segment. An expression patch specifies a word index within the segment that is to be patched and a number of symbolic expression patch alternatives. Each alternative can either succeed or fail. The first successful alternative specifies a value that is OR'ed into the specified word within the segment. It is an error if none of the alternatives are successful. Each alternative is represented by a expression subelement contained within the <ep> element.

The <ep> element has the following attributes:

**i** Relative word index (in hex) within the segment to which this choice patch applies.

**msg** Commentary to be included in the error message if none of the alternatives are successful. The default is the empty string.

There are four types of expression element: <val>, <refsym>, <refseg>, and <bin>. The first three types are leaves and the last type represents a binary operation.

The <val> expression element specifies an absolute value. It has one attribute:

**v** Absolute value (in hex).

The <refsym> expression element specifies a reference to a global symbol. It has one attribute:

**name** The name of the symbol.

The <refseg> expression element specifies a reference to a segment. It has two attributes:

**name** The name of the segment.

**off** The offset (in hex) from the start of the segment. The default is 0.

The <bin> expression element specifies a binary operation. It contains two subelements and it has one attribute:

**op** The binary operation to be performed in the values of the two expression subelements.

The possible operations are add, sub, ior (bitwise inclusive or), bic (bitwise clear), rol (rotate left), and mbz (must be zero). The bic operation computes the bitwise and of the first argument with the complement of the second of the second argument. The mbz operation checks that the bitwise and of the first and second arguments is zero, and if so the result is the first argument. If not, the expression fails.

## B.8 Extrn element

The <extrn> element represents an external symbol reference within a relocatable object file. It contains no subelements and has the following attributes:

**name** Name of a global symbol.

## B.9 Globl element

The <globl> element represents a global symbol definition within a relocatable object file. A global symbol can be defined as (1) an absolute value, (2) an offset from another global symbol, or (3) an offset from the base of a segment in the current object file.

The <globl> element contains no subelements and has the following attributes:

**name** Name of the global symbol being defined.

**sym** Name of a global symbol. Excludes the seg attribute.

**seg** Name of a segment within this relocatable object file. Excludes the sym attribute.

**off** Offset from the global symbol or segment base (in hex). If neither sym nor seg attribute appears, then “offet” is the absolute value of the definition. The default is “0”.

## B.10 Local element

The <local> element represents a local symbol definition within a relocatable object file. A local symbol can be defined as (1) an absolute value, (2) an offset from a global symbol, or (3) an offset from the base of a segment in the current object file.

The <local> element contains no subelements and has the following attributes:

**name** Name of the local symbol being defined.

**sym** Name of a global symbol. Excludes the seg attribute.

**seg** Name of a segment within this relocatable object file. Excludes the sym attribute.

**off** Offset from the global symbol or segment base (in hex). If neither sym nor seg attribute appears, then "offet" is the absolute value of the definition. The default is "0".

## B.11 Comm element

The <comm> element represents a global common request within a relocatable object file. It gives the name of a global symbol which is requested to be defined as the base address of a common area of a specified minimum size, allocated in a segment of a specified kind with a specified address rotation. If this symbol is not otherwise defined, then the loader is requested to create such a common area and define the symbol as its base address. Multiple global common requests of the same symbol name can be combined by taking the maximum of the sizes and combining the alignment requirements.

The <comm> element contains no subelements and has the following attributes:

**name** Name of the common area being requested.

**kind** Kind of segment in which the requested common area should be allocated.

**isize** Size of the requested common area in words (in hex).

**rota** Address rotation of the segment in which the requested common area should be allocated (in hex).

**alignbic** Alignment requirement of the requested common area, expressed as a mask of the word index bits that must be zero (in hex).

# Appendix C

# Software conventions

Here we describe the software conventions used by compilers for the Beehive architecture.

## C.1 Register usage

The Beehive CPU has 32 general purpose registers, although a few of the higher-numbered registers are overloaded in the instruction architecture and are therefore not as "general purpose" as the others. Furthermore, in version 2 of the CPU register 0 is a fixed zero register. Table C.1 summarizes the Apiary register usage conventions. The columns are described in more detail as follows.

**Name.** This column gives the name of the register as used in compiler-generated assembly code.

**Beehive overload.** A few of the higher-numbered registers have overloaded meanings when used as Ra or Rw in a Beehive instruction. This column describes the overloaded meaning, if any.

**Apiary purpose.** This column gives the purpose of the register as used during execution of compiler-generated code.

- **Fixed zero.** The register is fixed at the value zero.
- **General.** The register can be used for any purpose, such as storing a local variable or a compiler-generated temporary.
- **Frame pointer.** The register is used to store the frame pointer. If the current subroutine does not use a frame pointer, this is the same as a general register.
- **Temporary.** The register is used in a canned instruction sequence that is considered as a single instruction by the compiler. For example, the gcc compiler assumes that any primitive data type can be transferred between a general register and memory without requiring an additional temporary register to

| reg | name | Beehive overload | Apiary purpose | return value | parameter passing | callee save |
|---|---|---|---|---|---|---|
| 0 | zero | no | fixed zero | | | n/a |
| 1 | r1 | no | general | 1st word | | no |
| 2 | r2 | no | general | 2nd word | | no |
| 3 | r3 | no | general | | 1st word | no |
| 4 | r4 | no | general | | 2nd word | no |
| 5 | r5 | no | general | | 3rd word | no |
| 6 | r6 | no | general | | 4th word | no |
| 7 | r7 | no | general | | 5th word | no |
| 8 | r8 | no | general | | 6th word | no |
| 9 | r9 | no | general | | | yes |
| 10 | r10 | no | general | | | yes |
| 11 | r11 | no | general | | | yes |
| 12 | r12 | no | general | | | yes |
| 13 | r13 | no | general | | | yes |
| 14 | r14 | no | general | | | yes |
| 15 | r15 | no | general | | | yes |
| 16 | r16 | no | general | | | yes |
| 17 | r17 | no | general | | | yes |
| 18 | r18 | no | general | | | yes |
| 19 | r19 | no | general | | | yes |
| 20 | r20 | no | general | | | yes |
| 21 | r21 | no | general | | | yes |
| 22 | r22 | no | general | | | yes |
| 23 | fp | no | frame pointer | | | yes |
| 24 | t1 | no | temporary | | | no |
| 25 | t2 | no | temporary | | | no |
| 26 | t3 | no | temporary | | | no |
| 27 | p1 | no | platform | | | no |
| 28 | sp | no | stack pointer | | | yes |
| 29 | vb | Ra=RQ | void bval | | | no |
| 30 | r30 | Ra,Rw=LINK | none | | | no |
| 31 | r31 | Ra=PC, Rw=WQ | none | | | no |

Table C.1: Apiary register conventions

be allocated. This assumption comes into play when the compiler has to spill registers because it has run out. Unfortunately, the Beehive has no instructions to load or store bytes. The only way to accomplish it requires a sequence of instructions using several temporary registers. This is what the “temporary” registers are for. Since temporary registers are never used otherwise by compiler-generated code, they are also freely available for use by assembly-language support routines.

**Platform.** The register is reserved for use by the platform, such as, for example, as a pointer to the thread control block. Patform registers are not used by the compiler nor by any of the compiler runtime support routines.

**Stack pointer.** The register is used as the stack pointer.

**Void bval.** This register has two uses. First, some instructions are needed just for their effects on the condition codes or on the address queue but these instructions nonetheless must specify a destination register. The void bval register is used for this purpose. Second, the void bval register is used as part of a canned instruction sequence to construct a value which is subsequently used as the Rb argument of an instruction.

**None.** The register has no purpose.

**Return value.** The return value registers are used in passing the return value from a subroutine to its caller. Different things happen depending on whether the return value is a primitive type or pointer, or a struct, as discussed in Section C.3.1.

**Parameter passing.** A number of registers are available for passing parameters to a subroutine. There is considerable complexity here, since some parameters may be passed in registers and some on the stack, as discussed in Section C.3.4.

**Callee save.** Some registers are preserved across a subroutine call. If the callee has a reason to use the register, it can save and then later restore the register's contents. Other registers need not be preserved across a subroutine call.

## C.2 Memory layout

The Beehive architecture has a rather unusual memory layout. The memory space consists of $2^{32}$ words of 32 bits each. These words are indexed by word numbers $0 \ldots 2^{32} - 1$.

Memory space is divided into a top half and a bottom half. The bottom half of the memory space is occupied by physical memory. Hence there are $2^{31}$ words or 8 gigabytes of physical memory. The top half of memory space is occupied by I/O devices. The Beehive hardware manual should be consulted to learn about the I/O devices.

Instruction references use word numbers. This is reflected in the contents of the program counter, jump calculations, and subroutine return addresses. Hence the program counter can address any word in physical memory. It is considered improper for the program counter to address I/O space.

Data references, on the other hand, are different. In the Beehive architecture, data memory access is accomplished via queues that transfer addresses and data between the Beehive CPU and the memory controller. When a Beehive CPU instruction specifies that the ALU result is to be enqueued onto the address queue for transfer to the memory controller, the value that actually gets enqueued is the ALU result rotated right by two bit positions. The memory controller then treats the value it gets from the address queue as a word number.

What this means is that for the bottom $2^{30}$ words of memory space, the Beehive CPU addresses data words as if it were using byte addresses. The addresses are precisely those values in the range $[0 \ldots 2^{32} - 1]$ that are zero mod four. Note that the Beehive memory system has no concept of byte access or of unaligned word access. The Beehive memory system fetches and stores words referenced by word number, with the funny tweak that the word number is determined by rotating the CPU data address right by two bit positions. Still, it looks to software as if the memory system uses byte addressing.

Byte addressing is important to Apiary because GCC and MSIL have an unbreakable assumption that their target architecture is a byte addressed machine. So, Apiary uses the bottom $2^{30}$ words of memory space for data and pretends that it has such a machine.

Most data types occupy some multiple of words. Apiary is careful to allocate instances of these data types on word boundaries. Apiary gives the address directly to the Beehive memory system when such values are fetched and stored. For a one-byte data type, Apiary assumes that the address might refer to any byte. Fetching or storing such a value requires an instruction sequence to access the relevant memory word and insert or extract the byte. A two-byte data type is similar, except that Apiary assumes that the address is aligned on a two-byte boundary. Table C.2 describes the Apiary implementation of primitive C types.

Even though Apiary uses only the bottom $2^{30}$ words of memory space for data, software can still access the entire memory space using an unaligned address that has been cast into a pointer to `int`. You had better know exactly what you are doing if you do this.

Since Apiary uses only the bottom $2^{30}$ words of memory space for data, the next $2^{30}$ words of physical memory can be used for instructions, effectively giving an architecture with split I/D space. Neither GCC nor MSIL have any problem with using word numbers to reference instructions.

## C.3 C subroutine linkage

This section describes the Apiary subroutine linkage convention for C. We assume that all subroutines have a prototype in scope. All parameters except for arrays are passed by value. Recall that in C an array is actually passed as a pointer to its first element.

### C.3.1 Return value

A subroutine that returns a primitive type or pointer uses the return value registers to pass the return value back to the caller. Primitive types are one or two words long. Short integers and characters are coerced to an integer before being returned, so these cases never come up.

A subroutine declared as returning a struct actually works as follows. The caller allocates space for the return value and passes its address via the “return value” register r1. Regardless of the declaration of the struct, the allocated space starts on a word

`char` An unsigned integer type that occupies one byte. Coerced to `int` when passed as a parameter or as a return value. When allocated as a variable, assumed to be aligned on a word boundary. When allocated in a struct or when dereferencing a pointer, the address is assumed to have any alignment.

`signed char` Same as `char` except signed.

`short` A signed integer type that occupies two bytes. Coerced to `int` when passed as a parameter or as a return value. When allocated as a variable, assumed to be aligned on a word boundary. When allocated in a struct or when dereferencing a pointer, the address is assumed to be aligned on a two-byte boundary.

`unsigned short` Same as `short` except unsigned.

`int` A signed integer type that occupies one word. When allocated as a variable, assumed to be aligned on a word boundary. When allocated in a struct or when dereferencing a pointer, the address is assumed to be aligned on a word boundary.

`unsigned int` Same as `int` except unsigned.

`long` Identical to `int`.

`unsigned long` Same as `long` except unsigned.

`long long` A signed integer type that occupies two words, least significant word first. Same alignment assumptions as `int`.

`unsigned long long` Same as `long long` except unsigned.

`float` A floating point type that occupies one word. Same alignment assumptions as `int`. Operations not yet implemented.

`double` A floating point type that occupies two words. Same alignment assumptions as `int`. Operations not yet implemented.

Table C.2: Apiary implementation of primitive C types.

boundary and contains an integral number of words. The subroutine stores its return value in the indicated space and returns the address in the return value register r1.

### C.3.2 Layout of the parameter block

The parameters of a subroutine call are mapped consecutively onto a region of storage called the parameter block. Figure C.1 shows an example subroutine prototype and its corresponding parameter block. The type `Struct4wd` is assumed to be a struct that occupies four words. The relative address of each word in the parameter block is indicated on the left. Note that addresses decrease from top to bottom, as consistent

```
int subr (
  int a,
  int * p,
  long long c,
  Struct4wd q)
```

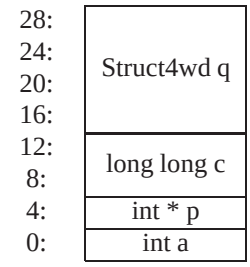



Figure C.1: Example C subroutine prototype and its parameter block.

with a stack that grows downward.

### C.3.3 Integral number of words

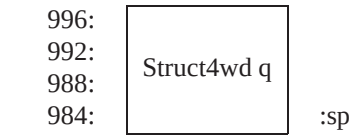
As each parameter is mapped to the parameter block, it is expanded if necessary to an integral number of words. Parameters of primitive integer type, such as `char` and `short`, that are shorter than a word are extended to a full word. The extension is signed or unsigned depending on whether the primitive type is signed or unsigned. The only other kind of type that would have to be extended is a struct. In this case, padding bytes of undefined content are added to the end as necessary to reach a word boundary.

### C.3.4 Passing the parameter block

For efficiency, all or some initial part of the parameter block may be passed in registers. The considerations are somewhat complex. The initial part of the parameter block that is passed in registers is called the *register part* and the remaining part that is passed on the stack is called the *stack part*.

First, the C language supports subroutines that have a variable number of arguments. Such a subroutine is called a *varargs subroutine* and the list of parameters in its prototype ends with three dots, which specifies that an arbitrary number of additional, anonymous parameters may be passed. For a varargs subroutine, none of the parameters may be passed in registers. The entire parameter block is passed on the stack.

Otherwise, some leading parameters in the parameter block may be passed in registers. The decision proceeds parameter by parameter. When once a parameter is encountered for which the decision is made not pass it in registers, that parameter and all




| reg | contents |
|---|---|
| LINK | return address |
| sp | 984 |
| r6 | 2nd word of long long c |
| r5 | 1st word of long long c |
| r4 | int * p |
| r3 | int a |

Figure C.2: Stack and register contents on entry to the subroutine of Figure C.1.

following parameters are passed on the stack.

The decision for a given parameter proceeds as follows. First, if the parameter occupies more than two words, it cannot be passed in registers. Second, if the parameter occupies more words than remain parameter passing registers available to hold it, it cannot be passed in registers.

In the case of a two-word parameter being passed in registers, the lower numbered register gets the first word of the parameter.

The stack part of the parameter block is always aligned on a word boundary. Since the Beehive CPU does not support unaligned word access, it would be terminally foolish to do otherwise.

### C.3.5 Calling the subroutine

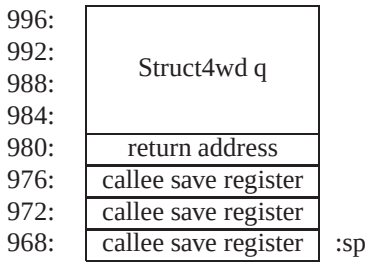
Once it is determined how the parameter block divides into a register part and a stack part, the calling program arranges the register part into parameter registers, arranges the stack part onto the stack, and then jumps to the entry point of the subroutine.

On entry to the subroutine, the stack pointer (sp) contains the address of the first word in the stack part of the parameter block. Figure C.2 illustrates the situation for the subroutine of Figure C.1. Note that the parameter registers are r3 through r8, but r7 and r8 have not been used because the next parameter is four words long, which is both longer than two words and also too long to fit in the remaining parameter registers.

### C.3.6 Subroutine entry

GCC compiler subroutines obey the following conventions on subroutine entry.

First, the return address is in the LINK register, where it is subject to being clobberred by any CALL or LLI instruction. So the first thing to do is push it onto the stack.




| reg | contents |
|---|---|
| sp | 968 |
| r6 | 2nd word of long long c |
| r5 | 1st word of long long c |
| r4 | int * p |
| r3 | int a |

Figure C.3: Stack and register contents after entry to the subroutine of Figure C.1. Three callee save registers are assumed.

Second, any callee save registers that are modified inside the subroutine are pushed onto the stack. Registers are pushed in reverse numerical order.

Figure C.3 shows the stack and register contents after this standard entry sequence has been performed for the method of Figure C.1. This illustration assumes that there are three callee save registers that need to be saved.

### C.3.7 Subroutine return

When a subroutine prepares to return, it first arranges its return value, if any. If the return value is a primitive type or pointer, the return value is left in the return registers. In this case, the return value type is extended to an integral number of words just like in the case of parameters, as discussed in Section C.3.3.

Otherwise, the return value is a struct. In this case, a pointer to a location in which to store the return value has been passed in the “return value” register. The subroutine copies its return value to this location. The address is left in the return value register.

The struct return value location is always aligned on a word boundary. Since the Beehive CPU does not support unaligned word access, it would be terminally foolish to do otherwise.

Any callee save registers that were modified by the subroutine must be restored to the values they had on entry. This includes the stack pointer and frame pointer. Finally, the subroutine jumps to the return address.

## C.4 Instruction schemas

Generally, the Beehive is a RISC architecture in which each instruction specifies two source registers, computes an ALU function, and stores the result in a destination register. This can be summarized by the instruction schema:

$$Rw = ALU(Ra, Rb)$$

In addition, the source register $Rb$ in a Beehive instruction can be replaced with a constant. This is summarized by the instruction schema:

$$Rw = ALU(Ra, CONST)$$

Although a Beehive instruction supports only a limited range of constants, an arbitrary constant can be constructed and left in a register using two preparatory instructions. In fact, due to register overloading, there are two general registers that can be specified for $Rw$ and $Rb$ that are not available for $Ra$, and using one of these is ideal for the construction of arbitrary constants in this manner. So an arbitrary $CONST$ operand can be effectuated, if necessary, via a prefix instruction sequence. For simplicity the following discussion omits details regarding constants.

In the Beehive architecture, memory access is accomplished via queues that transfer addresses and data between the Beehive CPU and the memory controller. There are three queues. The *address queue* transfers addresses from the CPU to the memory controller. The *read queue* transfers fetched data words from the memory controller to the CPU. The *write queue* transfers data words to store from the CPU to the memory controller.

The use of these queues can be considered an extension to the basic instruction schema, as discussed next.

### C.4.1 Fetching from memory

Fetching a word from memory requires two CPU instructions. The first CPU instruction computes the address and enqueues it onto the address queue. In effect, this instruction emulates some of the addressing modes that are found in CISC architectures. The instruction can compute the sum or difference of a source register and a small constant, thus emulating an offset addressing mode. The instruction can compute the sum or difference of two registers, thus emulating an indexing addressing mode. Since the instruction must specify a destination register to receive the ALU result, preincrement and predecrement addressing modes can also be emulated. Furthermore, by adding prefix instructions to the address generation instruction, other addressing modes such as absolute addressing and arbitrary offset addressing can be emulated.

The memory controller dequeues the address, fetches the data word—flushing and loading a cache line if necessary—and enqueues the data word onto the read queue. The second CPU instruction then dequeues this word from the read queue using an overloaded source register $Ra$ specification. Synchronization between the CPU and the memory controller is accomplished by having the CPU instruction stall if necessary until the read queue is non-empty.

Since the second CPU instruction can specify any three-register operation, we can think of the Beehive architecture as supporting the instruction schema:

$$Rw = ALU(FETCH, Rb)$$

The $FETCH$ operand is effectuated via a prefix instruction sequence that computes the address and enqueues it onto the address queue.

### C.4.2 Storing into memory

Storing a word into memory also requires two CPU instructions. One instruction computes an address and enqueues it onto the address queue in a manner exactly analogous to that for a memory fetch. Another instruction computes the data word to store and enqueues it onto the write queue.

When both the address queue and the write queue are non-empty, the memory controller dequeues the address and data word and performs the store, flushing and loading a cache line if necessary. The Beehive architecture permits the two instructions—one which enqueues the address and one which enqueues the data—to appear in either order.

At present, Apiary software adopts the convention of enqueuing the address first and then enqueuing the data to store. This convention has several advantages. First, it is conceptually simpler, since storing and fetching resemble each other in that in both cases the address is computed via prefix instructions. Second, if the condition codes need to be examined for the data that is stored, they are available at the conclusion of the instruction sequence. Third, for an instruction that performs both fetch and store, the computation of the store address is hidden by the fetch latency.

A CPU instruction enqueues a data word onto the write queue using an overloaded destination register specification. Since this instruction can specify any three-register operation, we can think of the Beehive architecture as supporting the instruction schema:

$$STORE = ALU(Ra, Rb)$$

The $STORE$ operand is effectuated via a prefix instruction sequence that computes the address and enqueues it onto the address queue.

### C.4.3 General schema

Since both read queue overloading and write queue overloading can be specified in the same Beehive CPU instruction, we can think of the Beehive architecture as supporting the instruction schema:

$$STORE = ALU(FETCH, Rb)$$

The $FETCH$ prefix must preceed the $STORE$ prefix or else the final instruction will deadlock.

Recall that the source register $Rb$ in a Beehive instruction can be replaced with a constant, giving us the the general instruction schema:

$$STORE = ALU(FETCH, CONST)$$

```
*(int *)0x01001200 = *(int *)0x01004700 ^ 0xfeedface;

    aqr_long_ld  vb,0x01004700    // fetch prefix
    aqw_long_ld  vb,0x01001200    // store prefix
    long_ld      vb,0xfeedface    // const prefix
    xor          wq,rq,vb
```

Figure C.4: Example C code and generated assembly code for the general schema.

Furthermore, recall that an arbitrary $CONST$ operand may have to be effectuated via a prefix instruction sequence. To implement the general schema, the compiler first emits the $FETCH$ prefix, then the $STORE$ prefix, then the $CONST$ prefix, and finally the Beehive instruction.

Figure C.4 shows an example of C code and generated assembly code for the general schema. The fetch prefix constructs the memory fetch address via a two instruction sequence using the link register. The fetch address is enqueued onto the address queue. Since the address is not needed otherwise, the destination register is specified as `vb`. The store prefix constructs the memory store address via a similar two instruction sequence. The const prefix constructs the arbitrary constant using a similar two instruction sequence, this time leaving the constant in the `vb` register where it can be used by the final instruction. Then, the final instruction dequeues the fetched word from the read queue, computes the ALU operation with the constant, and enqueues the result in the write queue.

Although this general schema was originally employed in the Beehive GCC compiler, I noticed that the const prefix did not help with code density. In any case in which the const prefix would be needed, it would take the same number of instructions to require the compiler to allocate a register and assemble the constant into that register, assuming available registers. Furthermore, forcing the compiler to load the constant into a register exposes the constant to common subexpression elimination. So now the GCC compiler does not use the const prefix.

# Bibliography

[1] C. Thacker. Beehive: A many-core computer for FPGAs, Oct. 2009. Unpublished.

[2] Xilinx. Data2MEM users guide, Apr. 2009. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/data2mem.pdf`.

# Index