**Basm: An Assembler for Beehive**

Tom Rodeheffer, MSR Silicon Valley

Basm is a simple, non-relocating assembler for the instruction set described in “Beehive: A Many-Core computer for BEE3” of May 7, 2009. The instruction set is summarized in section 4. Basm reads one or more assembler source files and writes a listing file, a code segment file, and a data segment file. The segment files are in Virtex VMEM format.

# Installing Basm

Get the zip file Basm.zip from <http://sharepoint/sites/apiary/Shared%20Documents> and unpack it. The zip file contains a .NET executable Basm.exe along with a Beehive shell program btest.s and the output listing and segment files that result from executing the command

Basm –code1 –data4 btest.s

Put Basm.exe into a directory on your search path. Basm is invoked using the command line as explained in the next section.

# Running Basm

Basm is invoked using the command line

Basm [options] sourcea.s sourceb.s ...

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

-o out Use out as the base file name for the output files, rather than the default, which is to use the first source file name.

-code1 Set the word size in the code segment to 1. This produces a word-addressed architecture for instruction addresses. The address offset from one instruction to the next is 1. This is the default.

-code4 Set the word size in the code segment to 4. This produces a byte-addressed architecture for instruction addresses. The address offset from one instruction to the next is 4.

-data1 Set the word size in the data and absolute segments to 1. This produces a word-addressed architecture for data addresses. The address offset from one data word to the next is 1. This is the default.

-data4 Set the word size in the data and absolute segments to 4. This produces a byte-addressed architecture for data addresses. The address offset from one data word to the next is 4.

-sim Run the simulator after a successful assembly. See section 4 for a discussion of the simulator.

Although any extension may be used for the assembler source files, the extension .s is recommended. Multiple source files are concatenated to form a single input source program, which provides rudimentary multi-module support. The output files have the following extensions:

.lst the listing file

code.mem the code segment file

data.mem the data segment file

Note that the code and data segment files append “code” and “data” to the base part of the file name. In this way each of the segment files has a “.mem” extension, which is required by the Virtex tools.

# Source file format

The assembler source files consist of comments, label definitions, symbol definitions, instructions, and directives.

## Comments

An end-of-line comment starts with // and goes 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

/\* multiline comments may be nested \*/ \*/

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.

## 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.

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.

## 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.

The prefix operators are

+ positive

- negative

~ bit complement

$ register number

The infix operators are

+ 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 (identical to logical shift left)

Note that the infix operators ROR, ROL, etc. are reserved words in the grammar.

## 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

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

## 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.

Another type is an offset in the code segment. Two code segment offsets can be subtracted to produce an absolute number. An absolute number can be added to or subtracted from a code segment offset, producing a code segment offset. There is no other way to use a code segment offset in an expression.

The final type is an offset in the data segment. These offsets are analogous to code segment offsets.

The restrictions on segment offsets exist in anticipation of the future development of a relocating assembler and linker. It is anticipated that code and data segments would have to be relocated by the linker.

## Label definitions

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

identifier: // this is a label definition

The identifier is assigned the address of the current location. Note that the current location can be in the code segment, in the data segment, or an absolute value (in the “absolute” segment). The segment type is part of the address.

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

## 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 // this is 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, an offset in the code segment, an offset in the data segment, a register number, or one of the special registers.

## 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.

### 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 the special register “wq”.

a a register number or a special register “pc”, “link”, or “rq”.

b a register number or an absolute number in the range 0..0xfff.

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

### 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 the special register “wq”.

a a register number or a special register “pc”, “link”, or “rq”.

b a register number or an absolute number in the range 0..0xfff.

### 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 the special register “wq”.

a a register number or a special register “pc”, “link”, or “rq”.

b a register number or an absolute number in the range 0..0x7f.

s an absolute number in the range 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.

### 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.

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)

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 in the range 0..0x1ffff.

Note that the jump instructions also function as subroutine calls, because these instructions also load the return address into the link register. This can be confusing when reading assembly code, because it can be difficult to see which of the jumps are actually subroutine calls. Basm provides the following aliases to make the intent more clear:

call\_add a,b // call (a + b)

callz\_add a,b // if (ZERO) call (a + b)

callm\_add a,b // if (MINUS) call (a + b)

callc\_add a,b // if (CARRY) call (a + b)

callnz\_add a,b // if (!ZERO) call (a + b)

callnm\_add a,b // if (!MINUS) call (a + b)

callnc\_add a,b // if (!CARRY) call (a + b)

### 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. Basm 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 the special register “wq”.

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 in the range 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.

### Synthesized jumps

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

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

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.

As before, Basm provides the following aliases to make the intent more clear when calling a subroutine:

call f // call f

callz f // if (ZERO) call f

callm f // if (MINUS) call f

callc f // if (CARRY) call f

callnz f // if (!ZERO) call f

callnm f // if (!MINUS) call f

callnc f // if (!CARRY) call f

### Load link immediate

The Beehive CPU has a “load link immediate” instruction that loads the high order 28 bits of the “link” register with a 28-bit constant and the low order 4 bits with zero:

lli i // link = i

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

i an absolute number whose low order four bits are zero.

### 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. Basm 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 the special register “wq”.

k any absolute number, any address in the code segment, or any address in the data segment.

### 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. Basm provides the following synthesized long jumps that assemble into a sequence of two machine instructions:

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

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

k any absolute number, any address in the code segment, or any address in the data segment.

As before, Basm provides the following aliases to make the intent more clear when calling a subroutine:

long\_call k // call k

long\_callz k // if (ZERO) call k

long\_callm k // if (MINUS) call k

long\_callc k // if (CARRY) call k

long\_callnz k // if (!ZERO) call k

long\_callnm k // if (!MINUS) call k

long\_callnc k // if (!CARRY) call k

### Simulator control

Basm 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 in the range 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 4 for a discussion of the simulator controls.

## 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.

### Segment selection directives

Basm implements three segments: a code segment, a data segment, and an absolute segment. Labels get the type of the segment in which they are defined. Each segment maintains a current location that advances independently. Assembled instructions and words can be emitted into the code and data segments.

The following directives change the current segment:

.code // switch to code segment

.data // switch to data 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.

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.

### Advance current location directives

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

.blkw i // advance current location by i \* wordsize

.blkb i // advance current location by i

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. This only makes a difference when *wordsize* is 4. The effects are identical when *wordsize* is 1. Note that *wordsize* is a property of the current segment, since the word size in the code segment can be different from the word size in the data or absolute segment.

### Align current location directive

The following directive advances 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

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 word size 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.

### Emit words or bytes directives

The following directive emits words into the current segment:

.word k,k,k,… // emit words

.byte k,k,k,… // emit bytes

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

k any absolute number, any address in the code segment, or any address in the data segment.

Multiple arguments separated by commas may be specified. Effects are identical when the current segment’s word size is 1.

### 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.

### Region nesting directives

Basm 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.

### 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, any address in the code segment, or any address in the data segment.

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

### Commentary directives

The gcc compiler emits directives related to global symbols and additional attributes in the symbol table. 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

.globl n // declare symbol n as global

.long n,... // alias for .word

## Predefined symbols

Basm 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

rq the read queue special register, read via Ra overload 29

wq the write queue special register, written via Rw overload 31

code offset 0x0 in the code segment

data offset 0x0 in the data segment

. the current location in the current segment

code. the current location in the code segment

data. the current location in the data segment

codewordsize address offset between words in the code segment (an absolute number)

datawordsize address offset between words in the data segment (an absolute number)

codewordror log base 2 of codewordsize (an absolute number)

datawordror log base 2 of datawordsize (an absolute number)

# 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 are separate.

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 follows:



## ALU function

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

1. A + B
2. A – B
3. A & B
4. A & ~B
5. A | B
6. A | ~B
7. A ^ B
8. A ^ ~B

### 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:

1. the read queue (takes one word; stalls until read queue is nonempty)
2. the link register
3. the program counter register (address of the current instruction)

### 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 the Rw, Count, and Rb fields (17 bits).

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

## Major Operation

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

1. RbConst, logical shift right by Count bits, write result in Rw
2. RbConst, logical shift left by Count bits, write result in Rw
3. RbConst, rotate right by Count bits, write result in Rw
4. RbConst, rotate left by Count bits, write result in Rw
5. RbConst, arithmetic shift right by Count bits, write result in Rw
6. CountRbConst, no shift, write result in Rw
7. CountRbConst, no shift, write result in Rw and into address queue as a write command
8. CountRbConst, no shift, write result in Rw and into address queue as a read command
9. Load link immediate (see below)
10. RwCountRbConst, no shift, jump if MINUS
11. RwCountRbConst, no shift, jump if ZERO
12. RwCountRbConst, no shift, jump if CARRY
13. RwCountRbConst, no shift, jump ALWAYS
14. RwCountRbConst, no shift, jump if NOT MINUS
15. RwCountRbConst, no shift, jump if NOT ZERO
16. RwCountRbConst, no shift, jump if NOT CARRY

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.

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.) If the condition codes are as specified, the address following the current instruction is loaded into the link register and the program counter register is loaded from the result of the shifter.

## 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.

## 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.

The Resv field is reserved and must always contain 0.

## 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 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 log base 2 of the data segment’s word size and the result is placed onto the address queue along with the indication of whether it is a read command or a write command. If necessary, the processor stalls until the address queue is non-full. Note that the address queue always contains word addresses, regardless of the data segment’s word size.

## Instruction fetch

Instructions are fetched as follows. The content of the pc is rotated right by log base 2 of the code segment’s word size and the result is used as the word address of the instruction to fetch.

# Simulator

The simulator implements 0x10000 words of code memory and 0x10000 words of data memory. The simulator supports all word size options. Memory accesses are performed instantly, with requests being taken from the address and write queues and results placed on the read queue with no simulated delay. An attempt to access a nonexistent memory address produces an error message. An attempt to read from an empty read queue also produces an error message.

## Memory mapped ASLI interface

In addition to the data memory, the simulated memory controller has a memory mapped ASLI interface at address queue value 0x80000000 (independent of word size) that connects with the console. Note that the shifter result that the CPU has to generate in order to create this address queue value depends on the data segment’s word size.

The ASLI interface uses a word with the format:



Reading the word gives the following status:

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

Writing the word 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

The simulator connects the ASLI interface 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.

## 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 2.8.10 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…31 reserved