Code Generation for the Beehive ISA

Thomas L. Rodeheffer
Microsoft Research, Silicon Valley

August 12, 2010

Developing computer architecture research platforms is difficult because it requires both the hardware platform and the software toolchain. In this paper, we describe the code-generation task of the software toolchain for the Beehive platform. Beehive is a many-core computer composed of simple 32-bit RISC connected using a token ring to a memory controller and Ethernet controller. We discuss the ISA, ABI, and idiosyncracies of the RISC core and its impact and the compiler. We also discuss the differences in porting GCC and LLVM and their resulting performance on small benchmarks in terms of code size and execution time.

1 Introduction

Beehive [18] is an experimental many-core computer implemented on a single FPGA. The system includes a number of RISC cores, each with a data cache and an instruction cache. The cores are connected to each other and to main memory using a token ring interconnect. A high-speed I/O channel is provided by an additional station on the ring that implements a 1 Gb/s Ethernet controller.

The Beehive design was initially implemented on the BEE3 (Berkeley Emulation Engine, version 3) [2]. The BEE3 implementation supports 8GB of main memory and the RISC cores run at 125 MHz.

Since the BEE3 is an expensive piece of hardware, a second version of the design, with a different memory controller, was implemented on the Xilinx ML509 (XUPv5) development board [20]. This version supports 2GB of main memory and the RISC cores run at 100 MHz. Figure 1 shows a system diagram.

The Beehive is intended to enable experiments in hardware architecture, especially in the memory system and in inter-processor coordination. This of course requires writing code to see how such experiments pan out. To write any significant amount of code, a compiler is absolutely necessary.

Although the Beehive system has many interesting architectural features, in this paper we focus on the Beehive instruction set architecture (ISA), which we take to include the CPU core and its load and store operations on its data cache. Assuming that the ISA remains invariant, we can reuse the same compiler for all experiments. So it is of interest to learn how to generate code for the Beehive ISA.

The remainder of this paper is organized as follows. Section 2 describes the Beehive ISA. Section 3 describes ways in which the Beehive ISA is perhaps surprisingly deficient and what software accomodations are necessary. Section 4 discusses the structure of modern target-independent compilers and what they need in order to generate code for a new target. Section 5 presents a comparative case study of retargeting to the Beehive two modern compilers, GCC and LLVM. Finally, Section 6 concludes.

2 The Beehive ISA

The Beehive ISA is in many respects a fairly conventional RISC architecture, with a 32-bit datapath, a register file containing 32 registers, and a three-operand instruction format. There are ways of performing arithmetic, accessing memory, affecting control flow, and assembling arbitrary constants. Figure 2 shows a schematic diagram of the core components that implement the Beehive ISA.

For simplicity, almost all Beehive instructions have the same format, which is illustrated in Figure 3. Details

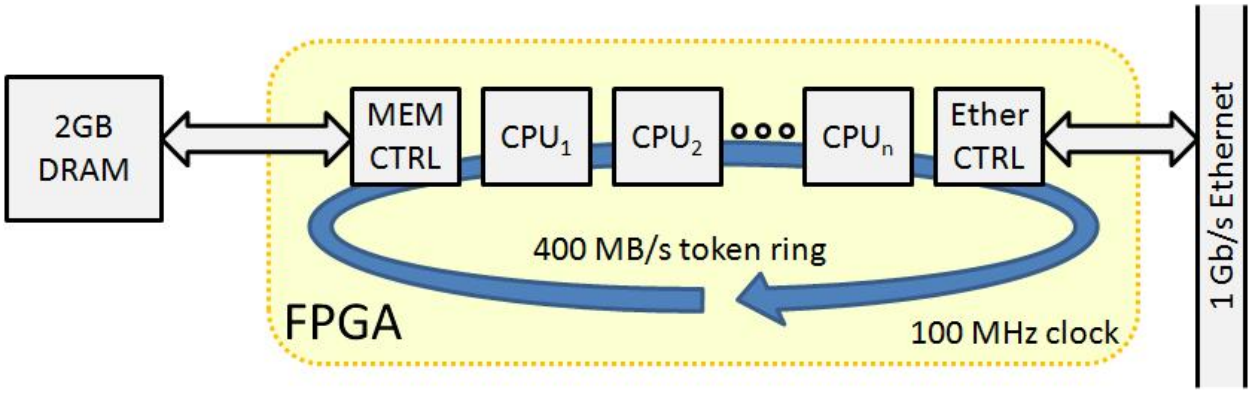

Figure 1: Beehive system diagram (ML509 implementation).

ADD	`A + B`	SUB	`A - B`
AND	`A & B`	ANDN	`A &~B`
OR	`A | B`	ORN	`A |~B`
XOR	`A ^ B`	XORN	`A ^~B`

Table 1: Beehive ALU functions.

LSL	shift left
LSR	logical shift right (zero fill)
ASR	arithmetic shift right (sign fill)
ROR	circular rotate right

Table 2: Beehive shift types.

about instruction formats and encodings may be found in the Beehive hardware documentation [18]. These details are of essential importance when generating binary code, but are irrelevant for understanding the problems of code generation, so we omit discussion of them.

2.1 Arithmetic

Most Beehive instructions read two registers, *ra* and *rb*, from the register file, combine the values with an ALU function, and then write the result back to a third register, *rw*. The eight simple ALU functions available are listed in Table 1. The Beehive ALU can add, subtract, and perform a number of bitwise logical operations.

A barrel shifter lies on the main datapath following the ALU. An instruction may select *no shift* or it may specify that the output of the ALU be shifted by any constant amount before being written back to the register file. The four types of shift available are listed in Table 2. In the circular right shift, bits shifted out from the least significant bit (lsb) re-enter at the most significant bit (msb). A circular rotate left shift of n bits can be accomplished by specifying a circular right shift of $(32 - n) \bmod 32$.

An instruction may specify a small positive integer constant in place of its second register input, *rb*. As opposed to architectures such as MIPS [8], in which the constant is sign-extended for arithmetic operations and zero-extended for logical operations, in Beehive, the constant is always zero-extended. This turns out not to be a limitation on Beehive. Because the Beehive ALU functions come in positive-negative or positive-complement pairs, it is always possible to select an alternate ALU function to achieve the effect of a negative or complement constant.

For an instruction that specifies a shift, the constant is limited to the range $[0, \ldots, 2^7 - 1]$. If the instruction specifies *no shift*, the shift count field is exploited to increase the available constant range to $[0, \ldots, 2^{12} - 1]$.

To make it convenient to load a constant into a register, register 0 in the register file is hardwired to always read as zero. This also makes it possible to discard a computed result without writing it into an otherwise useful register in the register file.

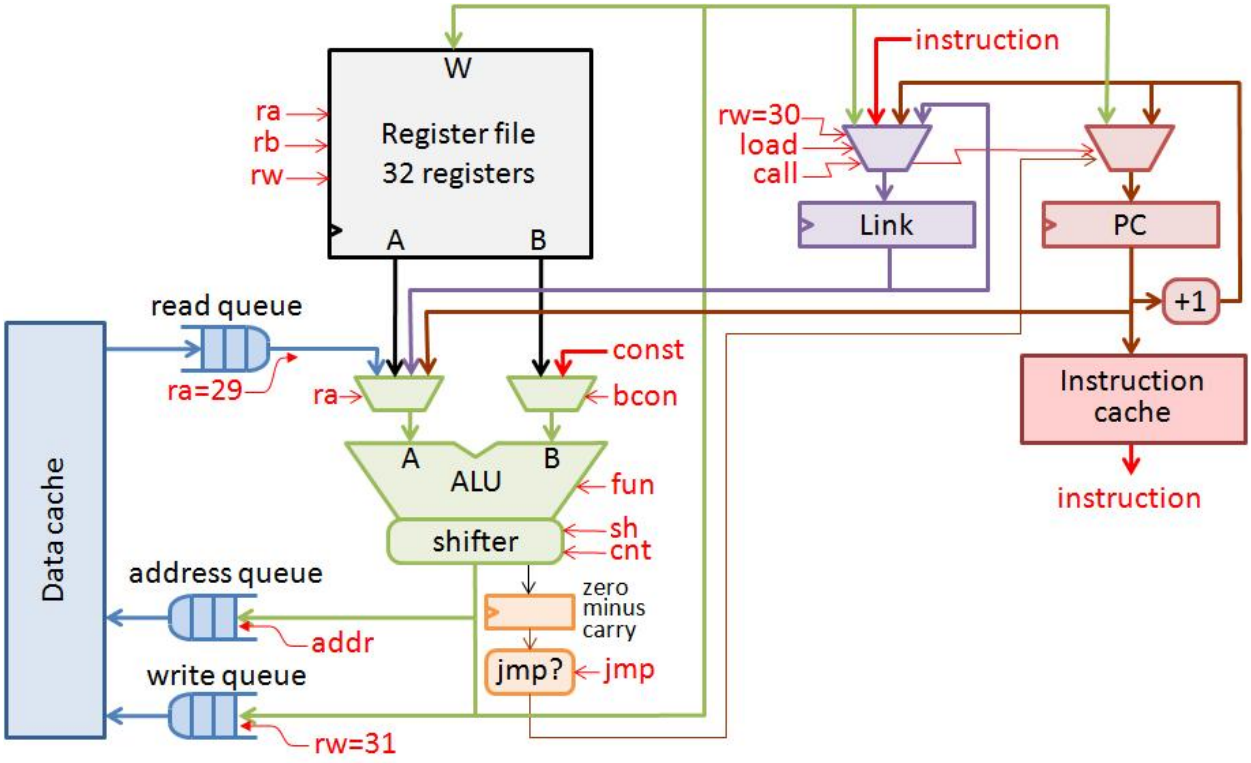

Figure 2: Core components that implement the Beehive ISA.

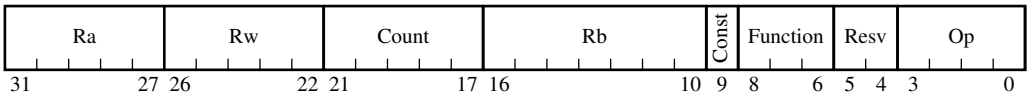

Figure 3: Beehive instruction format

2.2 Memory

In conventional RISC architectures, a memory read specifies both the address in memory to be read and the register into which the returned data is to be placed. Similarly, a memory write specifies the source of the data (a register), and the memory location into which the data is to be written.

Things are different in Beehive. The Beehive decouples address calculation from source or destination selection. To issue a read, an instruction sends the output of the ALU to the *address queue* as a read request. A non-empty address queue causes the read to occur, and the resulting data is placed in the *read queue*. The output of the read queue may be extracted by selecting *ra* = 29. We write this as *RQ*. If the data has not yet returned from the data cache, the processor stalls.

Writes are handled similarly. An instruction sends the output of the ALU to the address queue as a write request. Separately, the data to be written is sent to the *write queue* by selecting *rw* = 31. We write this as *WQ*.

The Beehive core has only one address queue and its data cache manager handles requests strictly in order. Hence the program establishes the exact sequence of operations and it can relate data on the read and write queues to the corresponding requests on the address queue.

It is convenient to describe sending read and write requests to the address queue as sending to *AQR* and *AQW*, respectively. Since it is not expected that shifting would be particularly useful as the last step in address computation, the instruction specifies these actions using variants of *no shift* and the available constant range is $[0, \ldots, 2^{12} - 1]$. Like ordinary arithmetic instructions, the result of the ALU is written to register *rw* in the register file. If the result of the address calculation is not otherwise needed, it can be discarded by specifying *rw* = 0. Specifying *rw* = *ra* can be used to implement pre-increment or pre-decrement addressing modes.

Figure 4 shows an example of an instruction sequence on Beehive using queued memory access. This three-instruction sequence reads a word from a stack frame location, adds to it the contents of a register, and writes the result back to a second stack frame location.[1] The results

```
AQR := sp + 12
WQ  := RQ + r3
AQW := sp + 20
```

Figure 4: Beehive instruction sequence showing queued memory access.

of the address computation instructions are not needed back in the register file, so they are written to register 0, which throws them away. This detail is omitted from the example.

The idea of queued memory access first appeared in this form in the design of the WM architecture family [19]. This architecture was unfortunately never implemented.

2.3 Control flow

To affect control flow, the Beehive has an unconditional jump, conditional jumps, and a jump-and-link. Collectively these are all jumps and they all work the same way.

To perform a jump, the Beehive instruction specifies a variant of *no shift* that sends the output of the ALU to the program counter. A hardware stall is necessary to nullify the effect of decoding the next instruction in sequence, but this is invisible to the programmer. Since jump target addresses are unlikely to be of any other use, jumps do not send the ALU result anywhere other than to the program counter and the *rw* field is exploited to increase the available constant range to $[0, \ldots, 2^{16} - 1]$.[2]

The program counter may be read by selecting *ra* = 31. We write this as *PC*. This reads the instruction address of the currently executing instruction. To accomplish a relative branch an instruction can add or subtract a small positive integer constant from the current program counter. The other ALU functions are not likely to be useful in this context.

In addition to unconditional jumps, the Beehive provides conditional jumps based on based on the three con-

[1] For a memory write operation, the Beehive ISA permits the instruction that sends the write address to *AQW* and the instruction that sends the write data to *WQ* to appear in either order. For simplicity of exposition in this example we show the write data being sent to *WQ* first. However, in general it is better to send the write address to *AQW* first, because (1) this provides an instruction to help fill the latency of fetching the read data from the data cache and (2) at the end of the sequence the condition codes reflect the value written to memory.

[2] The alert reader may wonder where the fifth bit of *rw* went. It went to augment the ISA with special purpose debugging instructions. For the gory details, consult the hardware manual.

zero	ALU/Shifter result was zero
minus	ALU/Shifter result msb was one
carry	carry out from ADD or SUB

Table 3: Beehive condition codes.

JZ	jump if zero	JNZ	jump if not zero
JM	jump if minus	JNM	jump if not minus
JC	jump if carry	JNC	jump if not carry

Table 4: Beehive conditional jumps.

dition codes listed in Table 3. The condition codes reflect the result of the ALU/Shifter from the *previous instruction*. The carry is undefined for ALU functions other than ADD or SUB. The condition codes are left unchanged by jump instructions and by the load-link-immediate instruction discussed later. All other instructions set the condition codes.

Beehive conditional jumps test the true or false value of any single condition code. Table 4 lists the six conditional jumps. Execution is based on the result of the previous instruction that set the condition codes. Generally this is the immediately previous instruction, but it need not be, if the intervening instructions were exclusively jumps or load-link-immediates.

The final kind of jump is the jump-and-link, commonly known as "call", which is intended for calling a subroutine. In addition to sending the ALU output to the program counter, the jump-and-link instruction sends the incremented former program counter to the *link register*. This is the return address for the subroutine.

The link register can be read by selecting *ra* = 30. The link register can be loaded directly from the output of the ALU/Shifter by selecting *rw* = 30. Finally, the link register can be loaded via the load-link-immediate instruction, as described next. For clarity, we write accesses to the link register as *LINK*.

2.4 Arbitrary constants

The load-link-immediate instruction is intended to help load an arbitrary constant. This instruction does nothing else except load the link register with a constant value. The high order 28 bits of the constant value can be specified arbitrarily. The low order 4 bits of the constant value are zero. Basically, what the load-link-immediate instruction does is send itself to the link register.

```
WQ  := A
WQ  := B
AQW := 6    // start multiplier
LO  := RQ   // low product word
HI  := RQ   // high product word
```

Figure 5: Beehive instruction sequence to compute $A * B$ as signed integers, with 64-bit result.

Using the load-link-immediate instruction a Beehive program can load an arbitrary 32-bit constant into a register in two instructions, the first a load-link-immediate and the second an OR of the *LINK* with the low order 4 bits as a constant. Since the second instruction could also send the result to *AQR* or *AQW*, an arbitrary address of a memory operation can be specified in two instructions. Likewise, a program can jump or call to an arbitrary address in two instructions. Since the load-link-immediate does not affect the condition codes, conditional jumps can also operate in this way.

2.5 Multiplier coprocessor

The Beehive has a number of coprocessors that are accessed through the memory queues using reserved addresses. From the point of view of the instruction set architecture, the most interesting coprocessor is a 32-bit multiplier.

The multiplier is used as follows. First, two words are sent to the write queue. These are the two signed 32-bit integers to be multiplied. Then, a special address write value is sent to the address queue. Finally, the 64-bit product is obtained by reading two words from the read queue. The low order product word comes out first. If necessary, the processor will stall until the product words are available. Figure 5 shows a code sequence.

If only a 32-bit result is desired, the high product word must still be read from RQ but it can be discarded by writing into register zero. The low order word of the product is the same regardless of whether A and B are considered as signed or unsigned integers.

There is a five cycle delay between the time the special address write value is sent to the address queue and the

```
WQ  := A
WQ  := B
AQW := 6         // start multiplier
AH  := A ASR 31  // compute AH
AF  := AH & B    // -AH * B
BH  := B ASR 31  // compute BH
BF  := BH & A    // -BH * A
F   := AF + BF   // fixup
LO  := RQ        // low product word
HI  := RQ + F    // high product word
```

Figure 6: Beehive instruction sequence to compute $A * B$ as unsigned integers, with 64-bit result.

first product word can be read from the read queue without stalling. These five cycles can be exploited to compute the required adjustment to the high order product word to convert the multiplier's signed multiply into an unsigned multiply.

Let PH be the high order word of the product of A and B considered as unsigned integers. Since the multiplier interprets the arguments A and B as signed integers, it sign-extends A and B into high-order words AH and BH respectively, and for its high order result word it computes

$$PH + AH * B + BH * A \bmod 2^{32}$$

The Beehive can compute AH in one instruction using an arithmetic shift right of A by 31 bits. Because AH is either 0 or $2^{32} - 1$, it turns out that

$$-AH * B = AH\&B \bmod 2^{32}$$

Hence computing $-AH * B$ takes one more instruction. A similar two instruction sequence suffices to compute $-BH * A$. The fifth instruction adds the two values to produce the required fixup. The fixup can be applied to the high order product word as it is read from the read queue. Figure 6 shows the entire code sequence.

3 ISA handicaps

The Beehive ISA is missing a lot of things one normally finds in a computer architecture. Most glaringly, the Beehive has no support for interrupts. This is a result of a strong desire to keep the Beehive design small and simple, so that it can be understood and modified easily and so that many cores can be implemented on a single FPGA.

Arranging to handle interrupts would add a lot of complexity to the design, especially considering the queued memory access. With many cores available, dedicating one core to handle real-time events is an acceptable overhead.

Skipping over the lack of floating point and other advanced functions, such as integer divide, the Beehive ISA also has numerous deficiencies in regards to small things that would be useful in application code sequences. We call these deficiencies *handicaps*.

For example, the Beehive ISA has no datapath access to the condition codes. To get a data value from a condition code, a Beehive program must perform a conditional jump to select between instructions that assume one value or the other.

For another example, the Beehive ISA has no provision to subtract from a constant other than zero. (Zero can be specified using *ra* = 0.) This is in contrast to the ARM [14], which has a reverse subtract instruction for exactly this purpose. To subtract from a constant other than zero, a Beehive program must first load the constant into a register.

Those handicaps are fairly minor. Table 5 lists some significant handicaps in the Beehive ISA and the best accomodation for working around them. We discuss these significant handicaps in the following subsections.

3.1 No signed integer comparison

In common with all modern architectures, the Beehive ISA uses two's complement arithmetic, in which a negative integer n in the range $[-2^{31}, \ldots, -1]$ is represented as the 32-bit unsigned value $2^{32} + n$. Addition is performed on these unsigned values modulo 2^{32} and the 32-bit result is a correct representation of the sum modulo 2^{32}.

As illustrated in Figure 7, subtracting $A - B$ is performed by complementing B, which computes $2^{32} - 1 - B$, and then adding that with $A + 1$. The additional 1 is a "carry in" that is easily handled by the logic circuitry that performs addition. In common with most architectures, the Beehive ISA actually computes a 33-bit result

Handicap	Accomodation
No signed integer comparison	Value space rotation (3 instructions extra)
No multiword add/sub support	Code sequence for 64-bit (4 instructions) Subword arithmetic in the general case (8 instructions per word)
Aligned word memory access only	Code sequence (11 instructions for byte store in the worst case)
No programmable shift count	Code table (5 instructions)

Table 5: Significant Beehive ISA handicaps and their accomodation.

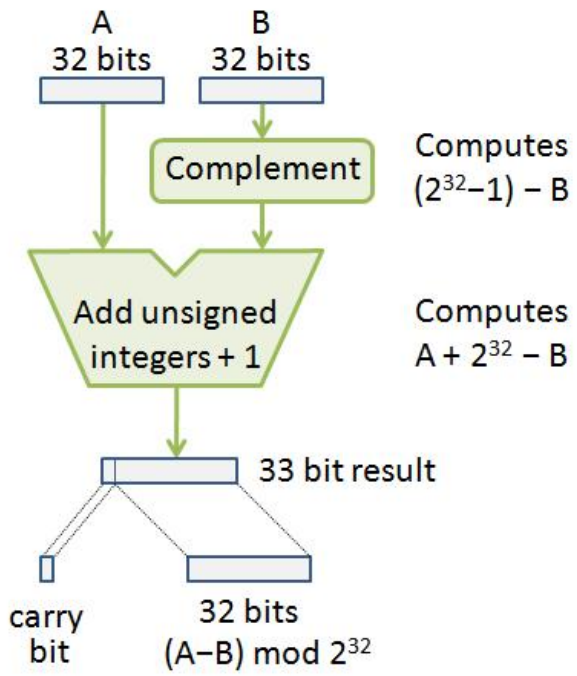

Figure 7: Two's complement subtraction on the Beehive.

$2^{32} + A - B$. The low order 32 bits are the true difference modulo 2^{32} and the high order bit is called the "carry".[3]

Now, suppose you have two 32-bit integers A and B you want to compare. Equality testing is easy: subtract and see if the result is zero. The Beehive ISA has a conditional jump on ALU result zero that does exactly what you want.

Order testing is more difficult. All you really need is a way to test for $A < B$. The other order tests can be reduced to $A < B$ by swapping the arguments, inverting the sense of the test, or both.

Seeing that the Beehive ISA has a conditional jump on ALU result minus (msb = 1), you might think that you could test for $A < B$ by computing $A - B$ and seeing if the result is minus. This seems reasonable, and in fact the Xerox PARC BCPL compiler [1] used exactly this method to compute $A < B$. However, (and the Xerox PARC implementers knew this) it is not correct. The problem is overflow. If the true result of $A - B$ is outside the range of representable values, you will get the wrong answer. Another way to look at it is to fix a choice for B and consider what you get by computing $A - B$ for all possible values of A. Clearly half of the computed values will be minus (msb = 1) but it cannot be possible that $A < B$ is true half of the time no matter what the choice was for B.

Examining Figure 7, notice that the 33-bit result of computing $A - B$ on the Beehive is $2^{32} + A - B$. Hence, if A and B are considered as unsigned integers, the result of the comparison $A < B$ can be determined on the Beehive by computing $A - B$ and testing that the carry is not set. Figure 8 shows the code sequence. This is is a correct method for comparing unsigned integers.

Comparing signed integers is more difficult. The problem is that negative signed integers (msb = 1) look like big unsigned integers. In fact, the carry, minus, and zero condition codes you get from computing $A - B$ are not sufficient to determine correctly the result of the comparison

[3] Some architectures invert the sense of the carry bit in subtraction, which is equivalent, as long as you know what is happening. We describe the way the Beehive does it.

```
A - B
if not carry goto L
```

Figure 8: Beehive instruction sequence to compute $A < B$ as unsigned integers.

```
T := 1 LSL 31
C := A XOR T
D := B XOR T
C - D
if not carry goto L
```

Figure 9: Beehive instruction sequence to compute $A < B$ as signed integers using value space rotation.

$A < B$ when A and B are considered as signed integers.

The best approach on the Beehive is to map the signed integers A and B to corresponding unsigned integers C and D whose unsigned comparison $C < D$ gives the correct result for the signed comparison $A < B$. The mapping that works is to add 2^{31} to both A and B. In terms of the bit representations, what this does is flip the msb. We call this approach *value space rotation*.[4]

Implementing value space rotation on the Beehive costs three instructions in addition to the subtract and conditional jump instructions needed to evaluate the unsigned comparison. Figure 9 shows an example code sequence. A, B, T, C, and D are all assumed to be registers. If there are several signed integer comparisons nearby, an intelligent compiler could save an instruction by reusing the constant in T.

3.2 No multiword add/sub support

Although 32-bit integers are common in modern machines, in some cases 32 bits is just not enough. Applications dealing with time often need to express more than 32 bits of precision. Some compilers lay out data structures using bit indexes, and this creates problems even on a machine with a 32-bit address space when the data structure happens to be extremely large. And given the capacity of disks, a data file can easily be larger than 2^{32} bytes. So it is important to consider how to support arithmetic on quantities larger than a single 32-bit word.

[4]The author thanks Jean-Philippe Martin for a revealing discussion about the benefit of value space rotation.

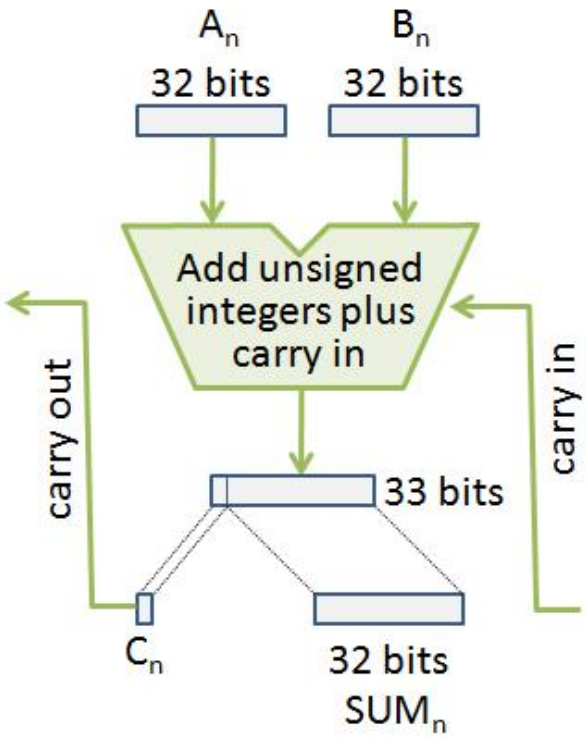

Figure 10: How to perform multiword addition.

On a simple architecture such as the Beehive, it is not surprising to find floating point operations and even integer division performed by library routines. But one would hope that addition and subtraction would be fairly efficient. As discussed in Section 3.1, subtraction is easy if you have addition, so we will focus on performing multiword addition.

Multiword addition requires forwarding any carry from the addition of a lower order word into the addition of the next higher order word. Figure 10 illustrates this process. Many machines have an "add-with-carry" instruction that does precisely this operation.

Lacking an "add-with-carry" instruction, computing $A_n + B_n + C_{n-1}$ requires two additions, either of which may result in a carry, depending on the input values. Knuth solved this problem in MIX using a "sticky" over-

```
      // inputs: A B carry
    if carry goto L1 // carry in...
    SUM := A + B     // only add
    goto E           // done...
L1: P := A + B       // first add
    if carry goto L2 // had carry...
    SUM := P + 1     // second add
    goto E           // done...
L2: SUM := P + 1     // second add
    0 - 0            // set carry
E:
      // outputs: SUM carry
```

Figure 11: Add-with-carry using conditional jumps.

flow indicator [9]. But the Beehive ISA does not have a sticky overflow indicator.

There are two approaches to solving this problem on the Beehive: using conditional jumps or using subword arithmetic.

In the conditional jump approach, each add operation is followed by a conditional jump whose purpose is to extract the result of the carry for that add. Then the program proceeds in different ways depending on whether there was a carry or not. Some optimizations are possible, because if there is no incoming carry, only one add is necessary, and if the first add has a carry, then the second one cannot. The Beehive code sequence to implement an add-with-carry operation is shown in Figure 11. If several of these code sequences appear in succession, as would be the case in performing multiword addition, there are additional opportunities for optimizing the conditional jumps between code sequences.

In the subword arithmetic approach, the Beehive hardware carry bit is ignored and the algorithmic carry is maintained as a value in a register. This approach is also suitable for coding in a higher-level language where concepts such as hardware carry bits are not available. The basic idea is that if the sum of A and B is compared bit-for-bit against the xor of A and B, wherever the bits are different there must have been a carry into that bit position. This observation can be exploited to compute the carry out by summing the msb of A, the msb of B, and the carry into the msb. The Beehive's post-ALU shifter is exceptionally useful in this regard. The resulting Beehive code sequence looks like the example shown in Figure 12. If several of these code sequences appear in succession, as would be the case in performing multiword addition, there are minor opportunities for optimizing the first and last sequences because of the constant carry input and dead carry output.

```
  // inputs: A B C
P   := A + B
SUM := P + C             // sum
X   := A ^ B             // xor
CH  := (SUM ^ X) LSR 31 // msb cin
AH  := A LSR 31          // msb A
BH  := B LSR 31          // msb B
PH  := AH + BH
C   := (PH + CH) LSR 1   // carry out
  // outputs: SUM C
```

Figure 12: Add-with-carry using subword arithmetic.

```
  // inputs: A0 B0 A1 B1
SUM1 := A1 + B1      // high first
SUM0 := A0 + B0      // low second
if not carry goto E // done...
SUM1 := SUM1 + 1     // fixup high
E:
    // outputs: SUM0 SUM1
```

Figure 13: Beehive 64-bit addition.

Computing "add-with-carry" is necessary for the general case of multiword addition, but if all we want is 64-bit addition, the Beehive has a reasonably efficient code sequence. The trick is to add the high order word first. Then, if there is a carry when adding the low order word, the high order result word can be adjusted. Figure 13 shows the resulting code sequence. A similar four-instruction sequence works for 64-bit subtraction.

3.3 Aligned word memory access only

Most modern instruction set architectures are byte-addressed, meaning that data addresses issued by the program increment by 1 for each successive "byte" in memory. Of course, this leaves open the question of what is a "byte". For example, GCC assumes a byte-addressed

machine, but with some effort you can alter its idea of a byte from 8 bits to 16 bits. However, typically a byte is identified with an octet, which is defined as an 8-bit quantity. Many low-level C programs, for example, assume that bytes (and characters) are 8-bit quantities. The intermediate language MSIL, although largely machine independent, assumes an octet-addressed machine.

In this paper, we identify a byte with an octet. Since byte-addressing is often assumed, we have to figure out how to implement it on the Beehive.

It turns out that the Beehive data cache manager deals exclusively in 32-bit words. The Beehive CPU can request that a word be fetched or stored by sending an address as a read or write request to the address queue.

To make it appear as though the Beehive has byte-addressing, the bits in the address are relabeled as they go through the address queue. The low order two bits (which the Beehive CPU could imagine discriminated between bytes 0, 1, 2, and 3 of a 32-bit data word) are rotated to the most significant position in the address and then the address is treated as a word address in memory.[5]

Therefore, if a Beehive program only fetches and stores words from addresses that are aligned on word boundaries, it can pretend that the Beehive memory system is byte addressed. Since many existing architectures forbid or strongly discourage accessing unaligned words, this is not a novel restriction. Compilers are used to aligning data structures on word boundaries.

Of course, we still have to implement some way for a Beehive program to read and write bytes and halfwords. This is the problem of subword memory access. The accomodation is to use a code sequence. Reading can be accomplished by fetching the relevant word and rotating and masking as necessary to extract the subword. Writing can be accomplished by a fetch-modify-store sequence. These code sequences are unfortunately fairly expensive. Figure 14 illustrates the code sequence needed to store a byte. Execution of 11 instructions is needed in the worst case. Some improvement in code size is possible by relegating most of the code sequence to a library routine.

[5]This relabeling does not occur on instruction addresses. Instruction addresses are word addresses. Since relative jump instructions compute the target address by adding or subtracting a constant from the current program counter, this permits them to have the maximum possible range. As a consequence, although data addresses and instruction addresses refer to the same memory, they have different representations.

```
      // inputs: A D
    AQR := A & ~3       // start fetch
    AQW := A & ~3       // start store
    T := (A & 3) LSL 3 // byte index * 8
    T := T + 1          // adjust
    goto PC + T         // indexed jump
B0: ...         // case for byte 0

B1: M := 255  // case for byte 1
    D := (D & M) LSL 8 // [1] mask data
    M := M LSL 8        // [2] position
    TMP := RQ & ~M      // [3] fetch word
    WQ := TMP | D       // [4] store word
    goto E              // [5] done
    nop                 // [6] pad
    nop                 // [7] pad

B2: ...         // case for byte 2

B3: ...         // case for byte 3

E:
```

Figure 14: Beehive byte store.

```
     // inputs: D C
   C := C & 31         // mask count
   if zero goto E      // done...
L: D := D LSL 1        // shift one bit
   C := C - 1          // decrement
   if not zero goto L // repeat..
E:
     // outputs: D
```

Figure 15: Shift via iterative loop.

3.4 No programmable shift count

Although a Beehive instruction can shift the output of the ALU by any constant amount, there is no provision for a programmable shift count. Hence, we need to make up this decifiency with a code sequence.

The simplest approach is to use an iterative loop, as shown in Figure 15. An average execution of 50 instructions is required. This approach is simple, but slow, and it fails to exploit the Beehive's barrel shifter.

```
    // inputs: D C
   T := (C & 31) LSL 1  // count * 2
   LINK := CODETABLE    // library rtn
   call LINK + T        // indexed call
    // outputs: D


// in library
CODETABLE:  // must be 16 word aligned
   D := D LSL 0  // [0] case for shift 0
   goto LINK     // [1] return
   ...
   D := D LSL n  // [0] case for shift n
   goto LINK     // [1] return
   ...
   D := D LSL 31 // [0] case for shift 31
   goto LINK     // [1] return
```

Figure 16: Shift via code table.

A much better approach is to perform an indexed jump on the shift count to an instruction that executes a shift of the desired amount. On the Beehive, this can be implemented efficiently as an indexed call to an out-of-line code table of short subroutines. Figure 16 illustrates the resulting Beehive code. As opposed to 50 instructions for the iterative loop solution, the codetable approach requires executing just 5 instructions.

4 Steps to code generation

Before generating code for the Beehive, many decisions have to be made concerning memory layout, register usage, and calling conventions. Collectively, these decisions are called the *application binary interface* (ABI). The ABI describes how programs will execute on the Beehive. Table 6 summarizes the decisions that we made for the Beehive.

Once the ABI has been designed, a suitable compiler can be found and retargeted to emit Beehive code. Figure 17 illustrates the typical structure of a modern, target-independent compiler. The parser converts source code to an internal form, many stages of analysis and transformation are applied, target instruction patterns are matched and preliminary code generated assuming an unlimited

Memory layout	
Stack direction	grows down, (sp) occupied
Stack alignment	word
Big/little endian	little endian
Multiword integer	little word first
Register usage	
Stack pointer	reg 28
Reserved regs	regs 24..27, 29, 30, 31
Calling sequence	
Caller saved regs	regs 11..23
Parameters	regs 3..10, as fit, then on stack
If varargs	all parameters on stack
Parameter order	lower reg or stack addr first
Where is sp on entry	at lowest addr stack parameter
Where is return addr	in the LINK register
Return value	regs 1..2, if fits, else via sret ptr
Sret ptr, if needed	reg 1

Table 6: Beehive application binary interface (ABI).

number of virtual target registers, instructions are scheduled to optimized pipeline delays, registers are allocated to map virtual registers to physical registers, and finally the resulting target code is emitted. Characteristics of the target inform all of the activities, especially instruction recognition. This is just a general outline; specific compilers differ considerably in detail.

The target-specific components are generally composed of a declarative part and an imperative part. The declarative part contains memory layout definitions, register definitions, instruction definitions, and pipeline definitions. It is generally written in a compiler-specific descriptive language that the compiler build process incorporates into tables and other data structures used by the compiler. The imperative part is written in the compiler's implementation language and generally deals with function prolog and epilog sequences, the calling convention, memory address patterns, custom matching, and custom transformations. Often snippets of imperative code are present in the declarative part.

When creating target-specific components for the Beehive ISA, we discovered a serious interaction between the Beehive and the compiler's register allocator. Recall that

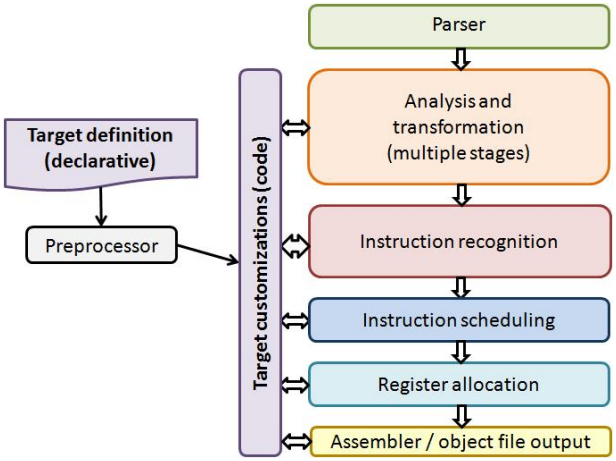

Figure 17: Modern compiler structure.

the register allocator's job is to map the unlimited virtual registers onto the target's physical registers. During this process, the register allocator may need to spill registers to stack locations, reload registers from stack locations, and copy one register to another. The register allocator assumes it can insert instructions to do this between any two instructions without otherwise altering the CPU state.

Unfortunately, this is not possible on the Beehive. All data movement instructions on the Beehive affect the condition codes. So inserting data movement between an instruction that sets the condition codes and the conditional jump that depends on them produces the wrong result. And trying to insert a register spill or restore into the middle of a queued memory access sequence will be disasterous.

The solution is to redefine the ISA as seen by the compiler to hide details of condition codes and memory access. Basically, we make the compiler deal largely in macro instructions which we later expand into individual Beehive instructions.

For memory access, we pretend that the Beehive has three-operand ALU instructions that take a register or memory location as the destination operand, a register or memory location as the first source operand, and a register or immediate value as the second source operand. A memory operand expands to a prefix instruction that sends the address to AQR or AQW, as appropriate, and the final instruction uses RQ or WQ as the corresponding register. This makes the Beehive look like a very conventional CISC architecture.

For the condition codes, we pretend that the Beehive has a combined subtract-and-conditional-jump instruction that takes two operands and a jump target. All of the integer comparisons can be reduced to this macro. Signed integer order comparisons of course need value space rotation first.

Once we have implemented macros, it becomes convenient to exploit the feature to extend the Beehive instruction set in other ways. We also added macros to implement subword memory accesses, programmed shifts, double-word (64-bit) add and subtract, jump tables (the result of compiling a switch statement), and add-with-carry using a reserved register to hold the carry.

5 Case study

We retargeted two modern compilers to generate code for the Beehive: GCC 4.3.3 and LLVM 2.7. Both compilers have publically available source code, incorporate modern optimization technology, and are desiged for retargeting.

5.1 Overview

GCC [3] has been developed through more than 20 years of improvements, rewrites, and redesigns. The target part is fairly modular and many working examples exist. The documentation, especially about writing a target port, is extensive and even includes a monograph [13] and a book [15], although these materials refer to old versions of the compiler. GCC grew up in the days of CISC machines and has very flexible instruction operand patterns. GCC is written in C with many macros.

LLVM [11, 10] was designed as a platform for compiler optimization. The entire compiler is modular and passes can be added as desired. There are some working target examples. The documentation is unfortunately somewhat sparse, dealing mainly at the level of how to get started and omitting all discussion of why things are the way they are. LLVM is aimed at RISC machines and is somewhat clumsy dealing with a CISC architecture, requiring a separate instruction pattern for each combination of operand forms. LLVM is written in C++ with many interfaces, templates, and extensive use of std:: classes.

5.2 Build process

Although they were developed on linux machines, both GCC and LLVM can be built on Windows, under MinGW [12]. Windows of course uses a different line termination convention than linux. After much sweat, we discovered that the GCC target module has one file (`target.opt`) that must not contain carriage returns, or else an awk script used in the GCC build process blows up. Otherwise the line termination sequence was immaterial.

GCC 4.3.3 requires the GMP and MPFR libraries in order to build. Eventually we found acceptable copies of these libraries. [4, 5].

LLVM 2.7 uses a customized GCC front end for the parser. We used a pre-built copy of the customized GCC front end and did not try to build it. We did not try to use Clang parser which the LLVM project is currently developing.

Both GCC and LLVM use a configuration script to tailor the build process to your environment. Among other things, this script compiles lots of little test programs to see what works in your environment. Fortunately, you only have to run this once, each time you change your configuration options.

In GCC, many target options are specified in a target header file that gets included everywhere. Any change to this file, or any addition or subtraction of instruction patterns, results in a complete recompile and global build. Also, making a mistake in one of the macro definitions can cause a compilation error to come out of seemingly anywhere, and it takes a while to track the problem back to the macro definition.

In LLVM, the target is implemented by creating a set of subclasses and registering them with the main compiler. Any change to the target requires only recompiling the target module and relinking the compiler.

It should be noted that the GCC build process also builds the runtime library `libgcc` needed by code generated for the target. This library is assumed by the LLVM compiler.

Table 7 lists the elapsed times it took to configure and build debugging versions of GCC and LLVM on our machine. The debugging versions contain more code and symbols than release versions and hence take longer to compile and link, but when experimenting with the compilers we wanted all the debugging help we could get. The most time-consumimg phase of the GCC build process is the global rebuild time, which is triggered, for example, whenever making any change to the target header file.

	GCC 4.3.3	**LLVM 2.7**
config	33m	5m
global build	70m	72m
target build	-	3m

Elapsed times on a 2.8 GHz Intel® Pentium® 4 with 2 GB memory.

Table 7: Comparison of build times.

5.3 Writing the target module

The recommended procedure for creating a new target module in both GCC and LLVM is to start with an existing target and edit. GCC has many options in the target header file that customize GCC to a specific target. In LLVM, on the other hand, target-specific code is responsible for much low-level customization.

For example, GCC examines instruction patters to determine how to spill, restore, and copy registers. GCC can automatically resolve stack slot references into address arithmetic. GCC can construct call and return sequences based on target options and target-specific code that assigns locations for parameters. And GCC can recognize many common addressing modes. All of these tasks must be accomplished by target-specific code in LLVM.

Both GCC and LLVM assume a basic repertoire of single-word integer operations and have a set of backup plans to deal with missing operations and unsupported data types. The compilers can substitute some operations for others or call a library routine. For example, bitwise double-word integer operations can be assembled as parallel single-word operations.

For the Beehive, we defined instruction patterns for single-word integer operations corresponding to the Beehive ALU functions. Both GCC and LLCM were happy to call library routines for integer multiply and divide.

Since the Beehive has a reasonably efficient code sequence for double-word integer addition and subtraction, we also defined instruction patterns for those operations.

Actually, for GCC all we needed was an instruction pattern. For LLVM, we needed a target lowering customization that was easy to implement, but there was absolutely no documentation telling how to do so.

It turned out that LLVM has a backup plan for double-precision integer addition that uses two single-precision add-with-carry operations. This would be fine if the Beehive had an add-with-carry instruction, but of course it does not. So we had to implement an add-with-carry macro for the Beehive for LLVM. Since we cannot expose the hardware carry bit without the possibility of the register allocator getting in the middle and inserting a register copy, we used the subword method and employed one of the reserved registers to hold the algorithmic carry.

Even after we implemented custom lowering of double-precision integer addition we found a case in which LLVM would still employ its backup plan involving add-with-carry. This case is a double-precision shift left by 1 bit position. LLVM transforms this into a double-precision integer add with self, which arguably would be better assuming the machine had double-precision operations. Unfortunately, this transformation happens after LLVM has finished considering custom lowerings.

Since subword access on the Beehive requires elaborate code sequences, we initially configured the address modes in the GCC target definition so that the only acceptable mode for bytes and halfwords was for the address to be in a register. This would make it easier to write the code sequences, since they would not have to consider offset, indexed, pre-increment, and pre-decrement modes. Unfortunately, we discovered that GCC assumes that the address modes for byte accesses are at least as general as for any other mode. In some cases, to test for the legality of an address transformation, GCC pretends that a transformed memory address formula is in byte mode and then asks the target module if the formula is acceptable. If byte addresses do not accept complicated modes, the original transformation will be rejected, resulting in poor code quality. So for GCC, we had to make our subword memory access code sequences deal with all of the Beehive address modes. This was not a problem with LLVM, because the instruction patterns for loading and storing each data type are completely independent.

Since we wanted to exploit the barrel shifter on the Beehive, we originally just defined instruction patterns for shifting by constant amounts and figured that the compilers would use their backup plans (i.e., calling a library routine) to deal with programmed shift amounts. This eventually revealed a bug in GCC. When compiling a switch statement, GCC has a large repertoire of methods at its disposal. One method is as follows. If the set of case labels is compact and several labels refer to the same branch, you can represent the labels as a bit mask and interrogate by shifting according to the switch value. GCC checks to see that the target has a shift instruction, and if so proceeds to elaborate this method. Well, the Beehive has a shift, but only for constant amounts. When GCC called its instruction recognizer to materialize the necessary shift instruction, nothing suitable could be found, and back came a null pointer. There was no check for this and the eventual result was a compiler memory smash.[6] So we had also to define an instruction pattern for programmed shifts. We made this pattern emit a macro.

Since the Beehive can support pre-increment and pre-decrement addressing modes, we originally included this in the GCC target definition. Eventually, we came across an example where GCC was pushing the address of a local variable onto the stack as an argument to a function call, and as a result of frame pointer elimination the macro instruction it emitted was:

```
*--sp := sp + offset
```

This instruction contains both a pre-decrement reference to the stack pointer (to accomplish the push) and a naked reference to the stack pointer (to compute the address of the local variable).[7] In the internal representation used by GCC, the semantics of this instruction should be that the pre-decrement applies *after* the value of the stack pointer is used in the addition. This is what the frame pointer elimination is expecting.

However, and the GCC documentation is very explicit on this point, it is not permitted to use a pre-decrement reference and a naked reference to the same register in a single instruction. The reason is that targets implement this differently, and the easiest thing was just to rule it out, rather than try to get all targets to follow the GCC semantics. In fact, after the Beehive macro expansion, the pre-decrement will apply before the value of the stack pointer is used in the addition.

[6] Richard Black provided this compiler test case.

[7] Richard Black provided this compiler test case as well.

To fix this, we removed pre-increment and pre-decrement addressing modes from the GCC target definition. This mainly affects the construction of call frames and GCC has alternate strategies using offset addressing. The result was that execution times increased by about one percent, which indicates that pre-increment and pre-decrement addressing modes are not really all that useful anyway. Since we had had this experience with GCC, when we came to LLVM we never defined these addressing modes in the first place.

5.4 Getting it to work

Both GCC and LLVM feature a frequent use of asserts to verify that the internals of the compiler are working as expected, and these asserts multiply when you compile a debug version. GCC has some command line flags that control extra printout, for example, of the instruction patterns matched. LLVM has a structure in which debugging output can be turned on invidually via the command line for each module. Or you can turn on debugging output globally and be awash in information. LLVM also features graphical output of its main data structure at several stages of code generation, which is useful in seeing what is happening.

Both GCC and LLVM expect that you will debug them using GDB or a similar debugger. GCC includes routines to print instances of its important data structures. LLVM includes instance methods for printing its important data structures.

We first implemented and debugged the GCC-based compiler, which took about 4 months. Then we implemented and debugged the LLVM-based compiler, which took about 1 month. The experience of retargeting the GCC compiler certainly helped with the LLVM effort. However, we also omitted a few features from the LLVM target.

We did not implement memory operands as part of ALU instructions in LLVM, due to how tedious it is to deal with flexible operand formats in LLVM. Instead, LLVM treats the Beehive in this respect as a RISC machine, moving data between memory and registers in instructions separate from the instructions that compute with data. We also did not implement custom lowering for jump tables. All of these are implemented in the GCC version of the compiler.

	lines of code	
	GCC 4.3.3 Beehive	**LLVM 2.7 Beehive**
declarative files	2143	2115
header files	1315	1233
source files	7865	7932
total	11323	11280

Table 8: Size of target modules.

Perm A tightly recursive permutation program.
Towers The canonical Towers of Hanoi problem.
Queens The 8-queens chess problem solved 50 times.
Intmm Two 2-D integer matrices multiplied together.
Puzzle A compute bound program.
Quick An array sorted using quicksort.
Bubble An array sorted using bubblesort.

Table 9: Selected programs in the Stanford Small Benchmark Suite.

Table 8 shows the number of lines of target-specific code for each compiler. Declarative files include machine instruction patterns and other definitions. They are written in a compiler-specific language that is preprocessed into code and tables that are compiled into the target module.

5.5 Emitted code quality

To evaluate the relative code quality between the GCC-based Beehive compiler and the LLVM-based Beehive compiler, we compiled and ran seven programs from the Stanford Small Benchmark Suite [7]. This suite is written in C and was assembled as a way of evaluating the early RISC machines. It is not really suited for modern advanced architectures and programming practices, but it can serve as a quick and easy yardstick for a simple architecture such as Beehive. Since the Beehive lacks floating point support, we omitted programs that use floating point. Table 9 lists a brief description of the seven programs we selected.

We compiled each program using the default compiler settings for optimization level 2 (−O2). At this setting, both compilers will omit frame pointers and perform

Program	code size (bytes) GCC 4.3.3 Beehive	LLVM 2.7 Beehive	LLVM 2.7 normalized (GCC=1)
Perm	504	692	1.37
Towers	1140	1420	1.25
Queens	956	1112	1.16
Intmm	452	728	1.61
Puzzle	2412	2532	1.05
Quick	924	1324	1.43
Bubble	536	648	1.21

Table 10: Emitted code size.

Program	cycle count GCC 4.3.3 Beehive	LLVM 2.7 Beehive	LLVM 2.7 normalized (GCC=1)
Perm	3596920	4047024	1.13
Towers	3398692	3324415	0.98
Queens	2028381	2043573	1.01
Intmm	4415054	5141907	1.16
Puzzle	18307420	17355448	0.95
Quick	3965216	2520901	0.64
Bubble	3382892	2870698	0.85

Table 11: Simulated execution time. Average of three consecutive runs.

some automatic inlining of called procedures.

Table 10 shows the code size in bytes generated for each test program by each compiler. Library routines are not included. Clearly, the test programs are all quite small and easily fit in the Beehive CPU's 4KB instruction cache. On average, the code produced by the LLVM compiler is about 20% larger than the code produced by the GCC compiler. This is largely explained by the fact that the LLVM compiler never combines memory accesses with ALU operations, whereas the GCC compiler does this frequently.

We ran the programs under an instruction-level simulator that gives accurate cycle counts for instruction execution. The simulator also makes a rough guess at the time required for flush and fill operations between the caches and main memory. (These operations use the token ring and take about 40 cycles in the ML509 Beehive system design, assuming no contention.) We ran each program three times, getting nearly identical cycle counts each time. The results are listed in Table 11.

We then ran the programs on an ML509 implementation of Beehive. We ran each program three times and took the average cycle count. The results are listed in Table 12. The actual execution times are reasonably close to the simulated execution times, showing that the simulator is fairly accurate, at least for single core programs.

Figure 18 shows a graph comparing the actual execution times for each program normalized to 1 for the GCC-based compiler.

It can be seen that, with a couple of notable exceptions, the code produced by the LLVM-based compiler performs

Program	cycle count GCC 4.3.3 Beehive	LLVM 2.7 Beehive	LLVM 2.7 normalized (GCC=1)
Perm	3640509	4047207	1.11
Towers	3464232	3325360	0.96
Queens	2036062	2153759	1.06
Intmm	4578507	5334151	1.17
Puzzle	18217788	18021134	0.99
Quick	3986108	2421114	0.61
Bubble	3633443	2872363	0.79

Table 12: Actual execution time. Average of three consecutive runs.

more or less the same as the code produced by the GCC-based compiler. This result was unexpected, since the LLVM-based compiler does not take any advantage of the CISC nature of the Beehive instruction set which enables combining a memory fetch and store with an ALU operation. The GCC-based compiler, on the other hand, exploits these operations frequently.

Next we examine the outliers to see why one compiler or the other performed better.

5.5.1 Inspection of Intmm

The worst case for LLVM was Intmm. Intmm is a program that multiplies two 40-by-40 integer matrices. Figure 19 shows the code of the inner loop. For clarity, declarations are included and constant identifiers have been

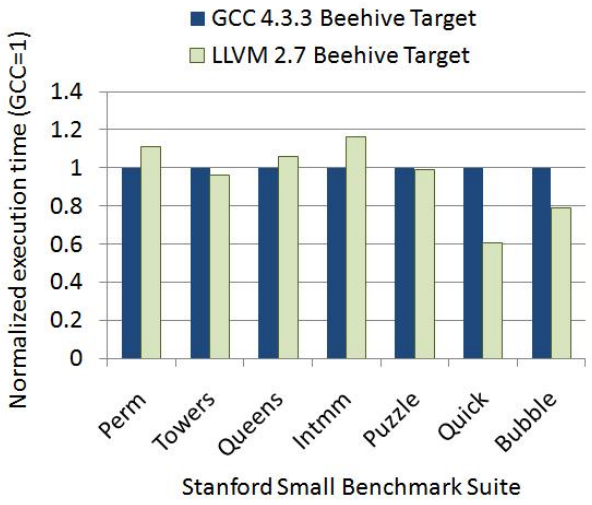

Figure 18: Comparison of actual execution time.

```
int *result;
int a[41][41], b[41][41], row, column;

*result = 0;
for(int i = 1; i <= 40; i++)
  *result = *result + a[row][i] * b[i][column];
```

Figure 19: Inner loop of Intmm.

replaced with their constant values.

Computing the address of an element in a two-dimensional matrix requires multiplying the row and column indexes by the row and column strides. In a loop like this, which takes successive elements from the same row or same column, the multiplications can be strength-reduced to incremental additions.

It turns out that the LLVM-based compiler failed to strength-reduce the subscript calculation for matrix b, thus causing the LLVM-based code to perform two multiplications in the inner loop, rather than just one. Since integer multiplication on the Beehive is relatively expensive, the additional multiplication accounts for the difference in execution time. The GCC-based compiler strength-reduced all the subscript calculations.

```
for (int i = 1; i <= 5000; i++) {
  seed = (seed * 1309 + 13849) & 65535;
  sortlist[i] = seed - (seed/100000)*100000 - 50000;
}
```

Figure 20: Inner loop of Quick init phase.

5.5.2 Inspection of Quick and Bubble

GCC considerably underperformed LLVM on Quick and Bubble. Quick is a program that initializes a 5000-element vector with random integers and then sorts it recursively using a version of the quicksort algorithm. It turned out that the GCC-generated code performed only about 1% slower than the LLVM-generated code in the sorting phase. Almost the entire difference in performance was due to the initialization phase, at which the GCC-generated code was five times slower than the LLVM-generated code.

Figure 20 shows the code of the initialization phase inner loop. For clarity, constant identifiers have been replaced with their constant values. The original source code uses a hand-coded linear congruence random number generator function, but both GCC and LLVM inline this function so, for clarity, it has been inlined in the figure as well.

The interesting part of the code is the expression

$$\text{seed} - (\text{seed}/100000)*100000$$

The original source code would appear to be a little confused about what it is doing. Observe that *seed* must be a positive integer in the range $[0 \ldots 2^{16} - 1]$, as established by the previous statement. So we can conclude that the result of seed/100000 will always be zero and thus most of this expression is pointless.

The GCC-based compiler transforms the expression into computing the remainder, seed % 100000, which is clever. Unfortunately, the Beehive does not have any hardware support for computing remainder and so a slow library call is needed.

The LLVM-based compiler transforms the division into a reciprocal multiplication, using the Beehive's hardware multiplier with a magic constant, which is very clever [6]. Then a second multiplication scales the quotient by 100000 and the result is subtracted from *seed*.

LLVM wins because the Beehive's hardware multiplier is about ten times faster than the software division loop called by GCC to calculate the remainder. But it seems like an accidental win.

Bubble has the same initialization phase and the same analysis explains the performance difference in that program as well.

6 Conclusions

This paper has described the Beehive ISA, how to generate code for it, and the results of retargeting two modern compilers, GCC and LLVM. Although the Beehive has a primitive instruction set, the instructions it does have can be effectively exploited by modern compilers through the use of instruction macros to hide some of the very rough edges.

The number of target-specific lines written for GCC and for LLVM were approximately the same. It was perhaps easier to retarget LLVM than GCC, in spite of more examples and more documentation being available for GCC. However, we omitted trying to express ALU memory operands in the LLVM effort. Surprisingly, this omission does not seem to have impacted performance of the emitted code.

Future machine architecture designs would benefit from considering how to support signed integer comparison and add-with-carry. In a machine that uses condition codes, it would also help to have support for address calculation and data movement that did not affect the condition codes.

A very slight extension to the Beehive ISA would be sufficient to support signed integer comparison and add-with-carry. Add-with-carry requires adding this specific instruction. For signed integer comparison, there are two approaches: (1) adding a fourth condition code to note signed overflow along with a pair of conditional jumps or (2) adding a value-space-rotation version of subtract. The first approach is typical of many architectures such as the ARM [14]. The second approach is more unusual but might be easy enough if an add-with-carry instruction is being added at the same time.

Using macros to conceal the details of condition codes and memory accesses was necessary enable the target-independent register allocators of GCC and LLVM to work with the Beehive ISA. However, using these macros has the disadvantage of producing redundant tests and poorly-scheduled address calculations. A peephole optimizer ought to be able to improve these code sequences.

In future work, the GCC-based compiler and the LLVM-based compiler could be compared on larger programs, such as those in the SPEC CPU benchmarks [16, 17]. The LLVM compiler could be improved to express ALU memory operands. And the effects of various peephole optimizations could be investigated.

Acknowledgements

Chuck Thacker designed and implemented the Beehive system and answered many questions on the exact definition of the Beehive instruction set architecture. Andrew Birrell and Richard Black employed the GCC compiler for real programs and thereby discovered test cases of many compiler bugs. John Davis provided a copy of the Stanford Small Benchmark suite and suggested several improvements in this paper. Finally, thanks to the many designers and implementers of GCC and LLVM who labored for years to produce these retargetable compiler systems.

References

[1] J. E. Curry et al. BCPL reference manual. Computer Sciences Laboratory, Xerox Palo Alto Research Center, Sept. 1979. A copy can be found at `http://www.fh-jena.de/~kleine/history/languages/xerox-parc-bcpldoc.pdf`.

[2] J. D. Davis, C. P. Thacker, , and C. Chang. BEE3: Revitalizing computer architecture research. Technical Report MSR-TR-2009-45, Microsoft Research, Apr. 2009.

[3] Free Software Foundation. GCC, the GNU compiler collection. `http://gcc.gnu.org/`.

[4] Free Software Foundation. The GNU MP bignum library version 4.3.1. `ftp://ftp.gnu.org/gnu/gmp/gmp-4.3.1.tar.gz`.

[5] Free Software Foundation. The GNU MPFR version 2.4.1. http://www.mpfr.org/mpfr-2.4.1/.

[6] T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. *SIGPLAN Not.*, 29(6):61–72, 1994.

[7] J. Hennessy and P. Nye. Stanford small benchmark suite, 1989. Unpublished. The author obtained a copy from John D. Davis. A copy can be found in the archives of the Stanford Hydra project at http://www-hydra.stanford.edu/fast/FAST_SW_Archive.zip.

[8] G. Kane. *MIPS RISC Architecture*. Prentice Hall, 1989.

[9] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, 1969.

[10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.

[11] llvm.org. The LLVM compiler infrastructure. University of Illinois at Urbana-Champaign. http://llvm.org/.

[12] mingw.org. MinGW: Minimalist GNU for Windows. http://www.mingw.org/.

[13] H.-P. Nilsson. Porting GCC for dunces, 2000. ftp://ftp.axis.se/pub/users/hp/pgccfd/pgccfd.pdf.

[14] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley, 2001.

[15] R. Stallman. *Using and Porting the GNU Compiler Collection (GCC)*. Free Software Foundation, Aug. 2000.

[16] Standard Performance Evaluation Corporation. SPEC CPU2000. http://www.spec.org/cpu2000/.

[17] Standard Performance Evaluation Corporation. SPEC CPU2006. http://www.spec.org/cpu2006/.

[18] C. Thacker. Beehive: A many-core computer for FPGAs, Jan. 2010. Unpublished.

[19] W. A. Wulf. The WM computer architectures principles of operation. Technical Report TR-90-02, Computer Science Department, University of Virginia, Jan. 1990.

[20] Xilinx, Inc. XUPV5-LX110T development system. http://www.xilinx.com/univ/xupv5-lx110t.htm.