Instruction Encoding Schemes that Reduce Code Size on a VLIW Processor

Eric J. Stotzer¹ and Ernst L. Leiss²

¹ Texas Instruments, Stafford, Texas, USA
estotzer@ti.com
² Department of Computer Science, University of Houston, Texas, USA
coscel@cs.uh.edu

Abstract. Code size is a primary concern in the embedded computing community. Minimizing physical memory requirements reduces total system cost and improves performance and power efficiency. VLIW processors rely on the compiler to statically encode the ILP in the program before its execution, and because of this, code size is larger relative to other processors. In this paper we describe the co-design of compiler optimizations and processor architecture features that have progressively reduced code size across two generations of a VLIW processor.

Keywords: Instruction level parallelism, code compression, VLIW, ILP

1 Introduction

Because very long instruction word (VLIW) processors do not have hardware to dynamically find implicit instruction level parallelism (ILP) at run time, they rely on the compiler to statically encode the ILP in the program before its execution [18]. Because ILP must be explicitly expressed in the program code, VLIW compiler optimizations often replicate instructions, increasing code size. VLIW processors combine multiple instructions into an execute packet. All instructions in an execute packet are issued in parallel. Code compiled for a VLIW will often include many null operation (NOP) instructions, which occur because there is not enough ILP to completely fill an execute packet with useful instructions. NOPs are dummy instructions that have no effect except to advance the hardware pipeline state.

While code size is a secondary concern in the computing community overall, it can be significant in the embedded community. Minimizing the amount of physical memory reduces total system cost. Reducing code size improves system performance by allowing space for more code in on-chip memory and program caches. Code size reduction improves power efficiency, because it reduces the energy required to fetch instructions from memory [3, 11].

The C6X processor [22] is a family of high performance embedded processors. In addition to providing an overview of the C6X processor, this paper presents the co-design of architecture and compiler features that reduce code size across two generations of the C6X processor family produced by Texas Instruments and used in embedded high performance applications.
2 The C6X Processor Family

Figure 1 is a block diagram of the C6X processor. The first generation C6X (C6X-1) processors are the TMS320C62 (C62) and TMS320C67 (C67). The C6X-1 is a fully pipelined VLIW processor, which allows eight new instructions to be issued per cycle. All instructions can be optionally guarded by a static predicate. If an instruction’s predicate operand evaluates to false, then the results of the instruction are anulled. The C62 provides a foundation of integer instructions. It has 32 static general-purpose registers, partitioned into two register files. A small subset of the registers may be used as predicates. Load instructions have four delay slots, multiplies have one delay slot, and branches have five delay slots. Other instructions have no delay slots. The C67 adds floating point instructions.

The second generation C6X (C6X-2) processor is the TMS320C64 (C64), which builds on the C62 by removing scheduling restrictions on existing instructions and providing additional instructions for SIMD packed-data processing. The C6X-2 processors increase the size of register file by providing an additional 32 static general-purpose registers.

The C6X processors are supported by an optimizing compiler [21]. The structure and operations of a compiler are well documented [4, 5, 8, 16]. The compiler implements important optimization phases such as function inlining, loop nest optimization, data dependence analysis, software pipelining, and many more. The compiler is absolutely critical for exploiting ILP.

2.1 Encoding Wide Instructions

Each instruction on the C6X-1 processors is 32-bit. Instructions are fetched eight at a time from program memory in bundles called fetch packets. Fetch packets are
aligned on 256-bit (8-word) boundaries. The C6X-1 processors can execute from one to eight instructions in parallel. Parallel instructions are bundled together into an *execute packet*. As fetch packets are read from program memory, the instruction dispatch logic extracts execute packets from the fetch packets. All of the instructions in an execute packet execute in parallel. Each instruction in an execute packet must use a different functional unit.

The execute packet boundary is determined by a bit in each instruction called the *parallel-bit (or p-bit)*. The p-bit (bit 0) controls whether the next instruction executes in parallel. The p-bits are scanned from lower to higher addresses. If the p-bit of instruction $i$ is 1, then instruction $i + 1$ is part of the same execute packet as instruction $i$. If the p-bit of instruction $i$ is 0, then instruction $i + 1$ is part of the next execute packet.

Figure 2 shows three p-bit patterns for fetch packets, which result in the following execution sequences for the eight instructions: fully serial, fully parallel, and partially serial. The least significant bits (LSBs) of the program memory address shows how the fetch packets are laid out in memory. Each instruction with a p-bit set to 0 marks the end of an execute packet. The fully serial fetch packet has eight execute packets each made up of one instruction. The fully parallel fetch packet has one execute packet made up of eight instructions, which will execute in parallel. The partially serial fetch packet has four execute packets.

<table>
<thead>
<tr>
<th>LSBs of the byte address</th>
<th>Fully Serial Fetch Packet</th>
<th>Fully Parallel Fetch Packet</th>
<th>Partial Serial Fetch Packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000₂</td>
<td>Instruct A ; 0</td>
<td>Instruct A ; 1</td>
<td>Instruct A ; 0</td>
</tr>
<tr>
<td>00100₂</td>
<td>Instruct B ; 0</td>
<td>Instruct B ; 1</td>
<td>Instruct B ; 0</td>
</tr>
<tr>
<td>01000₂</td>
<td>Instruct C ; 0</td>
<td>Instruct C ; 1</td>
<td>Instruct C ; 1</td>
</tr>
<tr>
<td>01100₂</td>
<td>Instruct D ; 0</td>
<td>Instruct D ; 1</td>
<td>Instruct D ; 1</td>
</tr>
<tr>
<td>10000₂</td>
<td>Instruct E ; 0</td>
<td>Instruct E ; 1</td>
<td>Instruct E ; 0</td>
</tr>
<tr>
<td>10100₂</td>
<td>Instruct F ; 0</td>
<td>Instruct F ; 1</td>
<td>Instruct F ; 1</td>
</tr>
<tr>
<td>11000₂</td>
<td>Instruct G ; 0</td>
<td>Instruct G ; 1</td>
<td>Instruct G ; 1</td>
</tr>
<tr>
<td>11100₂</td>
<td>Instruct H ; 0</td>
<td>Instruct H ; 1</td>
<td>Instruct H ; 0</td>
</tr>
</tbody>
</table>

Fig. 2. Instruction fetch packet layout showing p-bits

On the C6X-1 processors, execute packets cannot span a fetch packet boundary. Therefore, the last p-bit in a fetch packet is always set to 0, and each fetch packet starts a new execute packet. Execute packet are *padded* with explicit parallel NOP instructions to prevent subsequent execute packets from spanning a fetch packet boundary. On the C6X processor, NOP instructions may execute on any of the eight functional units. Figure 3 shows how parallel NOP instructions
are used to align spanning execute packets. Instructions in { } are part of the same execute packet and, therefore, will execute in parallel.

```plaintext
1 * Example of spanning execute packets
2 fp0: ; start fetch packet 0
3 {i1} ; ep0
4 {i2, i3, i4} ; ep1
5 {i5, i6} ; ep2
6 {i7, i8, } ; ep3 spans fp boundary
7 fp1: ; start fetch packet 1
8 {i9}
9 ...
10 fpn:

12 * Example after inserting padding NOPs
13 fp0: ; start fetch packet 0
14 {i1} ; ep0
15 {i2, i3, i4} ; ep1
16 {i5, i6, Nop, Nop} ; ep2
17 fp1: ; start fetch packet 1
18 {i7, i8, i9} ; ep3
19 ...
20 fpn:
```

**Fig. 3.** Example of NOP padding to prevent a spanning execute packet

Except for a few special case instructions such as the NOP, each instruction has a predicate encoded in the first four bits. Figure 4 is a generalization of the C6X 32-bit three operand instruction encoding format. The predicate register is encoded in the condition (creg) field, and the z-bit encodes the true or not-true sense of the predicate. The dst, src2, and src1 fields encode operands. The x-bit encodes whether src2 is read from the opposite cluster’s register file. The op field encodes the operation and functional unit, and the s-bit specifies the cluster that the instruction executes on.

**Fig. 4.** Typical 32-bit instruction encoding format

NOP instructions occur frequently in VLIW programs and as a result increase code size. Often NOP instructions are executed for multiple sequential cycles. The C6X processors include a multi-cycle NOP for encoding a sequence of NOP
instructions. Figure 5 shows how four sequential NOP instructions are encoded as one multi-cycle NOP.

3 NOP Compression

While the fetch-execute packet encoding scheme and multi-cycle NOP instruction improved the code size of the C6X-1 processors relative to previous VLIWs, embedded applications required further code size reductions. To this end, it was proposed that the C6X-2 processors allow execute packets to span fetch packet boundaries with some minimal restrictions, thus reducing code size by eliminating the need for padding NOP instructions. Further, we proposed new instructions, similar to the multi-cycle NOP, that remove pipeline NOP instructions [20, 9].

Because VLIW processors have exposed pipelines, NOP instructions are inserted to compensate for instruction latencies. A latency is the number of cycles it takes for the effect of an instruction to complete. Instruction scheduling is used to fill the latency or delay slots with other useful instructions. Assuming that other instructions are unavailable for execution during the instruction latency, explicit pipeline NOPs are inserted after the instruction issues to maintain correct program execution. On the C6X processor, the load (LD) and branch (B) instructions have five and six cycle latencies, respectively. In Figure 5, the delay slots of load and branch instructions are filled with NOP instructions.

```
1 ; load with sequential nops
2   LD *a0,a5 ; 5-cycle latency
3   NOP
4   NOP
5   NOP
6   NOP
7   ADD a5,a5,a6 ; <- a5 value arrives
8
9 ; load with multi-cycle nop
10  LD *a0,a5 ; 5-cycle latency
11  NOP 4 ; multi-cycle NOP
12  ADD a5,a5,a6 ; <- a5 value arrives
13
14 ; branch with multi-cycle nop
15  B Label ; 6-cycle latency
16  NOP 5 ; multi-cycle NOP
17 ; Branch occurs
```

Fig. 5. Example of pipeline NOP instructions in the delay slots of the load (LD) and branch (B) instructions
Eric J. Stotzer and Ernst L. Leiss

The breakdown of pipeline and padding NOP instructions occurring in a set of embedded applications is shown in Figure 6. These data show that 8.8% and 6.1% of all instructions are pipeline and padding NOPs, respectively. The control-oriented applications had more pipeline NOP instructions, and the loop-oriented applications had more padding NOP instructions. Loop-oriented code with high degrees of ILP contains more padding NOP instructions, because execute packets tend to be larger in loop code, thus increasing the likelihood of spanning execute packets. The opposite occurs in control-oriented code with lower degrees of ILP, because execute packets are smaller and, therefore, pack more efficiently into fetch packets. Because it has less ILP and is characterized by short test-and-branch sequences, pipeline NOPs occur more often in control-oriented code.

<table>
<thead>
<tr>
<th>Application</th>
<th>Pipe NOPs</th>
<th>Pad NOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hard-Disk</td>
<td>16.4%</td>
<td>2.9%</td>
</tr>
<tr>
<td>GSM5.1</td>
<td>5.5%</td>
<td>8.8%</td>
</tr>
<tr>
<td>ADSL</td>
<td>11.6%</td>
<td>4.0%</td>
</tr>
<tr>
<td>G.732.1</td>
<td>5.4%</td>
<td>7.8%</td>
</tr>
<tr>
<td>EVRC</td>
<td>5.5%</td>
<td>6.3%</td>
</tr>
<tr>
<td>Imaging</td>
<td>5.1%</td>
<td>9.7%</td>
</tr>
<tr>
<td>JPEG</td>
<td>7.1%</td>
<td>9.4%</td>
</tr>
<tr>
<td>Reed-Solomon</td>
<td>7.3%</td>
<td>6.6%</td>
</tr>
<tr>
<td>G.723.1 float</td>
<td>7.9%</td>
<td>6.5%</td>
</tr>
<tr>
<td>Zlib</td>
<td>10.6%</td>
<td>4.1%</td>
</tr>
<tr>
<td>DSP loops</td>
<td>2.1%</td>
<td>4.0%</td>
</tr>
<tr>
<td>G.729</td>
<td>3.9%</td>
<td>8.3%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>8.8%</strong></td>
<td><strong>6.1%</strong></td>
</tr>
</tbody>
</table>

**Fig. 6.** Percentage of pipeline NOPs, and padding NOPs in a set of embedded applications compiled for the C6X-1 processors

We proposed a new instruction format that implements a variable delay operation [9, 12], that in effect encodes subsequent NOP instructions as an operand. For example, because of their long latency, branch instructions are often followed by a multi-cycle NOP instruction. The Branch with NOP (BNOP) instruction encodes the subsequent NOP instructions as an operand (see Figure 7). The effect is that the NOP is issued in parallel with the instruction requiring the latency. The NOP operand ranges from zero to the maximum latency of the instruction.

We found that this new instruction format reduce average code size by 6%. Control-oriented code, which contains more branches, saw a larger improvement. Loop-oriented code benefited more from eliminating the restrictions on spanning execute packets.
Eric J. Stotzer and Ernst L. Leiss

1 ; Branch with subsequent NOP 5
2 B label
3 NOP 5
4
5 ; branch with nop 5 encoded as an operand
6 BNOP label,5 ; b label || nop 6
7 ; Branch occurs

Fig. 7. Example using the branch with parallel NOP instruction

4 Results

The section summarizes, the code size reduction and performance impact of NOP compression.

The 84 benchmarks used for this analysis are organized into the groups enumerated below. The EEMBC telecom, automotive, and networking groups are taken directly from the EEMBC-v1 embedded benchmark suite [10].

- **EEMBC telecom**: signal processing loop kernels typically found in telecommunication applications.
- **EEMBC automotive**: control functions found in automotive engine applications.
- **EEMBC networking**: packet processing algorithms taken from network and communication infrastructure applications.
- **DSP codecs**: voice compression decoder/encoders (codecs) used in wireless communication applications including: evrc, g723.1, g729, gsmAMR, gsmefr, gsmfr, gsmhr, Reed-Solomon, modem, trau, and wbamr.
- **Multimedia codecs**: image, video, music and data compression codecs including: jpeg, mpeg4, mp3, ac3, aes, and des.
- **Control code**: tcpip, zlib, and hard-disk drive.
- **Other applications**: miscellaneous benchmarks such as drhystones, dijkstra, susan, and others.

The benchmarks were compiled with the TI C6X compiler version 6.0.8. The C6X compiler has a speed-or-size option that determines how the compiler makes tradeoffs between optimizing for code size or performance. The three speed-or-size options used in this analysis are:

- **Size**: aggressively minimize code size at the expense of performance.
- **Size and speed**: minimize code size with nominal impact on performance.
- **Speed**: aggressively maximize performance at the expense of code size (default).

The compiler provides options to select the processor generation and to disable optimization passes that target specific processor features. For the following results, the baseline configuration is the C6X-1 generation processor and
the speed-or-size option set to speed. All results are normalized to this baseline configuration. Benchmark code size reduction and speedup (performance improvement) are measured on the following four configurations:

- **C6X-1**: C6X-1 generation processor
- **C6X-2**: C6X-2 generation processors with NOP compression.

The differences in the benchmark results for the C6X-1 and C6X-2, and configurations correspond to the improvements that are seen when upgrading to the new processor generations.

For each configuration, Figures 8, 9, 10, and 11 present the normalized average code size reduction and speed improvement (the y-axis). For code size reduction, smaller is better, and for speed improvement, larger is better. Each configuration is compiled for all three speed-or-size options (the x-axis).

Figure 8 summarizes the results for all benchmarks relative to C6X-1 compiled for speed with no loop collapsing. The results are averages across all benchmarks. Recall that because the compiler is disabling optimizations that increase code size, performance will degrade at the speed&size and size options. The goal is to minimize the degradation as much as possible.

Fig. 8. Code-size reduction and performance improvement on all benchmarks

The following analysis of the results in Figure 8 is grouped by compiler speed-or-size option.

- **Speed**: The 6.1% speedup is predominately from increasing the size of the register file. NOP compression has reduced the code-size by 17.5%.
- **Speed and Size**: The 4.4% speedup improvement is from the larger register file. NOP compression has again reduced the code-size reduction by 17.5%.
- **Size**: A large code-size reduction of 18.1% with a slight improvement in the performance of 1.8

Figure 9 shows the results for the EEMBC telecom, automotive, and networking benchmarks. These benchmarks are smaller and representative of code in specific application spaces. The automotive benchmarks are dominated by
Fig. 9. Code-size reduction and performance improvement on the EEMBC benchmarks

current control code, the telecom code is primarily loop-oriented, and the networking algorithms are a mixture of control- and loop-oriented code.

The results for the DSP codecs, multimedia codecs, control code, and other applications are shown in Figures 10 and 11. Many of these benchmarks are complete applications. The DSP and multimedia codecs are loop-oriented applications, the control code is obviously control-oriented, and the other applications are a mixture both.

The results include both control- and loop-oriented benchmarks. However, NOP compression improves both loop- and control-oriented code.

When a programmer’s primary desire is to control code size, this additional range can be useful in balancing performance and code size in a memory-constrained application, which is common in embedded systems.
Fig. 10. Code-size reduction and performance improvement on DSP and multimedia application benchmarks

Fig. 11. Code-size reduction and performance improvement on control code and other miscellaneous application benchmarks
5 Conclusion

We believe that it is best to develop new processor architecture features in tandem with a compiler that uses these features. In this way, the efficacy of processor architecture features can be gauged quickly. The processor and compiler evolve together. The compiler developer influences how the designer builds a processor so that it is easier to program.

Code size is a primary concern in the embedded computing community. Minimizing physical memory requirements reduces total system cost, improves system performance by allowing more code to fit in on-chip memory and program caches, and improves power efficiency.

We have presented the co-design of compiler optimizations and architecture features, which reduce the code size for a VLIW processor. NOP compression: a hardware technique that compresses the encoding of padding and pipeline NOP instructions.

We presented the code-size reduction and performance impact of using these techniques to compile a set of 84 benchmarks. With the compiler's speed-or-size option set to maximize speed, the results showed an impressive cumulative average code size reduction of 17.5%.

6 Related Work

The Intel Itanium IA-64 processor [13] is a VLIW design, although Intel refers to it as an explicitly parallel instruction computing (EPIC) processor. Today (in 2010), the VLIW philosophy is popular in embedded processors. Besides the C6X, other examples of embedded VLIW processors include the Analog Devices SHARC DSP [1], the Trimedia ST200 [2], the Infineon Carmel [7] and, Tensilica’s Xtensa LX2 [23].

Another approach to reduce code size is to store a compressed image of the VLIW program code in external memory and to use run-time software or hardware to decompress the code as it is executed or loaded into a program cache [14, 19]. Other approaches have used variable length instruction encoding techniques to reduce the size of execute packets [3]. Finally, some embedded processors have modes that implement smaller opcode encodings for a subset of frequently occurring instructions. Examples of mode-based architectures are the ARM architecture’s Thumb mode [6, 17] and the MIPS32 architecture’s MIPS16 mode [15].

References

Eric J. Stotzer and Ernst L. Leiss

10. EEMBC: http://www.eembc.org