Generation of Control and Data Flow Graphs from Scheduled and Pipelined Assembly Code

David C. Zaretsky[1], Gaurav Mittal[1], Robert Dick[1], and Prith Banerjee[2]

[1] Department of Electrical Engineering and Computer Science, Northwestern University
2145 N. Sheridan Road, Evanston, IL 60208-3118
{dcz, mittal, dickrp}@ece.northwestern.edu
[2] College of Engineering, University of Illinois at Chicago
851 South Morgan Street, Chicago, IL 60607-7043
prith@uic.edu

Abstract. High-level synthesis tools generally convert abstract designs described in a high-level language into a control and data flow graph (CDFG), which is then optimized and mapped to hardware. However, there has been little work on generating CDFGs from highly pipelined software binaries, which complicate the problem of determining data flow propagation and dependencies. This paper presents a methodology for generating CDFGs from highly pipelined and scheduled assembly code that correctly represents the data dependencies and propagation of data through the program control flow. This process consists of three stages: generating a control flow graph, linearizing the assembly code, and generating the data flow graph. The proposed methodology was implemented in the FREEDOM compiler and tested on 8 highly pipelined software binaries. Results indicate that data dependencies were correctly identified in the designs, allowing the compiler to perform complex optimizations to reduce clock cycles.

1 Introduction

Traditionally, the high-level synthesis problem is one of transforming an abstract, timing-independent specification of an application into a detailed hardware design. High-level synthesis tools generally convert the abstract design into a control and data flow graph (CDFG) that is composed of nodes representing inputs, outputs, and operations. The CDFG is a fundamental component of most compilers, where most optimizations and design decisions are performed to improve frequency, power, timing, and area. Building a CDFG consists of a two-step process: building the control flow graph (CFG), which represents the path of control in the design, and building the data flow graph (DFG), which represents the data dependencies in the design. However, when high-level language constructs are not readily available, such as in the case where legacy code for an application on an older processor is to be migrated to a new processor architecture, a more interesting problem presents itself, known as *binary translation*. Much research has been performed on CDFG generation from software binaries and assembly code. However, there has been very little work on generating complete CDFGs from scheduled or pipelined software binaries. Data

E. Ayguadé et al. (Eds.): LCPC 2005, LNCS 4339 , pp. 76 – 90, 2007.

dependency analysis of such binaries is more challenging than that of sequential binaries or high-level language applications.

When translating assembly codes from digital signal processors (DSPs), it is common to encounter highly pipelined software binaries that have been optimized manually or by a compiler. Consider the Texas Instrument C6000 DSP assembly code for the *vectorsum* function in Figure 1. In this architecture, branch operations contain 5 delay slots and loads contain 4 delay slots. The || symbol indicates the instruction is executed in parallel with the previous instruction and the [ ] symbol indicates the operation is predicated on an operand. Clearly, the *vectorsum* code is highly pipelined; each branch instruction is executed in consecutive iterations of the loop. Moreover, the dependencies of the ADD instruction in the loop body change with each iteration of the loop: A6 is dependent on the load at instruction 0x0004 in the first iteration of the loop, A6 is dependent on the load at instruction 0x000C in the second iteration of the loop, etc. Generating a CDFG to represent this pipelined structure is very challenging. In doing so, one must consider the varying data dependencies and also ensure that each branch is executed at its proper time and place. Branch instructions that fall within the delay slots of other branch instructions complicate the structure of the control flow graph. For instance, when the predicate condition, A1, on the branch instruction in the loop body is *false*, the previous branch instructions that were encountered during the execution sequence will continue to propagate and execute. This may occur within the loop, or possibly after exiting the loop. More complex software pipelines may contain branch instructions with various targets, producing multiple exit points in a CDFG block.

```
0x0000 VECTORSUM:   ZERO  A7
0x0004              LDW   *A4++, A6   ; 4 delay slots
0x0008      ||      B     LOOP        ; 5 delay slots
0x000C              LDW   *A4++, A6
0x0010      ||      B     LOOP
0x0014              LDW   *A4++, A6
0x0018      ||      B     LOOP
0x001C              LDW   *A4++, A6
0x0020      ||      B     LOOP
0x0024              LDW   *A4++, A6
0x0028      ||      B     LOOP
0x002C      ||      SUB   A1, 4, A1
0x0030 LOOP:        ADD   A6, A7, A7
0x0034      || [A1] LDW   *A4++, A6
0x0038      || [A1] SUB   A1, 1, A1
0x003C      || [A1] B     LOOP        ; branches executes here
0x0040              STW   A7, *A5
0x0044              NOP   4
```

Fig. 1. TI C6000 assembly code for a pipelined *vectorsum* procedure

In this paper, we present a methodology for generating CDFGs from scheduled and pipelined assembly code. This process consists of three stages: generating a control flow graph, linearizing the assembly code, and generating the data flow graph. We use the methods described by Cooper et al. [6] for generating a CFG from scheduled assembly code. In addition, we extend their work to support more complex architectures that employ parallel instruction sets and dynamic branching. We also present a linearization process, in which pipelined structures are serialized into linear assembly. This allows for proper data dependency analysis when constructing data flow graphs. This methodology was incorporated in the FREEDOM compiler, which translates DSP assembly code into hardware descriptions for FPGAs. The techniques described in this paper were briefly discussed in previous work [11,19]; here we present a more refined and elegant approach in greater detail.

The remainder of this paper is structured as follows: Section 2 discusses related work in the area of CDFG generation from assembly code. Section 3 provides an overview of the FREEDOM compiler infrastructure and its intermediate language architecture. Section 4 describes our method of generating a CDFG from scheduled and pipelined assembly code in detail. Finally, Sections 5 and 6 present experimental results and conclusions, respectively.

2 Related Work

There has been a great deal of fundamental research and study of binary translation and decompilation. Cifuentes et al. [3,4,5] described methods for converting assembly or binary code from one processor's instruction set architecture (ISA) to another, as well as decompilation of software binaries to high-level languages. Kruegel et al. [9] described a technique for decompilation of obfuscated binaries. Stitt and Vahid [16,17] reported work on hardware-software partitioning of software binaries. Levine and Schmidt [10] proposed a hybrid architecture called HASTE, in which instructions from an embedded processor are dynamically compiled onto a reconfigurable computational fabric using a hardware compilation unit. Ye et al. [18] developed a similar compiler system for the CHIMAERA architecture.

Control and data flow analysis is essential to binary translation. Cifuentes et al. [5] described methods of control and data flow analysis in translating assembly to a high-level language. Kastner and Wilhelm [8] reported work on generating CFGs from assembly code. Decker and Kastner [7] described a method of reconstructing a CFG from predicated assembly code. Amme et al. [1] presented work on a memory aliasing technique, in which data dependency analysis is computed on memory operations using a value-based analysis and modified version of the GCD test [2].

There has been very little work on generating CDFGs from highly pipelined software binaries in which branch instructions appear in the delay slots of other branch instructions. The most comprehensive work on building CFGs from pipelined assembly code was reported by Cooper et al. [6]. However, their method does not consider the complexities of modern processor architectures that utilize instruction-level parallelism and dynamic branching techniques. In this paper, we address these issues and present methods to handle CDFG generation from software binaries that feature these sophisticated scheduling techniques.

3 Overview of the FREEDOM Compiler

This section provides a brief overview of the FREEDOM compiler infrastructure, as shown in Figure 2. The compiler was designed to have a common entry point for all assembly languages. Towards this effort, the front-end uses a description of the source processor's ISA in order to configure the assembly language parser. The ISA specifications are written in SLED from the New Jersey Machine-Code toolkit [14,15]. The parser generates a virtual assembly representation called the Machine Language Syntax Tree (MST), which has a syntax similar to the MIPS ISA. The MST is generic enough to encapsulate most ISAs, including those that support predicated and parallel instruction sets. All MST instructions are three-operand, predicated instructions in the format: *[pred] op src1 src2 dst.* A CDFG is generated from the MST, where optimizations, scheduling, and resource binding are preformed. The CDFG is then translated into a high-level Hardware Description Language (HDL) that models processes, concurrency, and finite state machines. Additional optimizations and customizations are performed on the HDL for the target architecture. This information is acquired via the Architecture Description Language (ADL) files. The HDL is translated directly to RTL VHDL and Verilog to be mapped onto FPGAs, and a testbench is generated to verify that the output is correct.

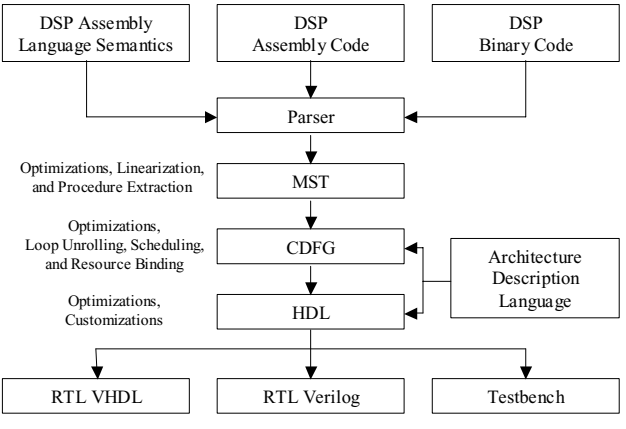

Fig. 2. Overview of the FREEDOM compiler infrastructure

The fixed number of physical registers on a processor necessitates advanced register reuse algorithms in compilers. These optimizations often introduce false dependencies based on register names, resulting in difficulties when determining data dependencies for scheduled or pipelined binaries and parallel instruction sets. To resolve these discrepancies, each MST instruction is assigned a timestep, specifying a linear instruction order, and an operation *delay*, equivalent to the number of execution cycles. Each cycle begins with an integer-based *timestep, T*. Each instruction *n* in a parallel instruction set is assigned the timestep $T_n = T + (0.01 * n)$. Assembly instructions may be translated into more than one MST instruction. Each instruction *m*

in an expanded instruction set is assigned the timestep $T_m = T_n + (0.0001 * m)$. The write-back time for the instruction, or the cycle in which the result is valid, is defined as *wb = timestep + delay*. If an operation delay is zero, the resulting data is valid instantaneously. However, an operation with delay greater than zero has its write-back time rounded down to the nearest whole number, or *floor(timestep)*, resulting in valid data at the *beginning* of the write-back cycle.

Figure 3 illustrates how the instruction timestep and delay are used to determine data dependencies in the MST. In the first instruction, the MULT operation has one delay slot, and the resulting value in register A4 is not valid until the beginning of cycle 3. In cycle 2, the result of the LD instruction is not valid until the beginning of cycle 7, and the result of the ADD instruction is not valid until the beginning of cycle 3. Consequently, the ADD instruction in cycle 3 is dependant upon the result of the MULT operation in cycle 1 and the result of the ADD operation in cycle 2. Likewise, the first three instructions are dependant upon the same source register, A4.

TIMESTEP	PC	OP	DELAY	SRC1	SRC2	DST
1.0000	0X0020	MULT	(2)	$A4,	2,	$A4
2.0000	0X0024	LD	(5)	*($A4),		$A2
2.0100	0X0028	ADD	(1)	$A4,	4,	$A2
3.0000	0X002c	ADD	(1)	$A4,	$A2,	$A3

Fig. 3. MST instructions containing timesteps and delays for determining data dependencies

4 Building a Control and Data Flow Graph

This section presents our methodology for generating a CDFG from scheduled and pipelined assembly code. This process consists of three stages: generating a control flow graph, linearizing the assembly code, and generating a data flow graph.

4.1 Generating a Control Flow Graph

Cooper et al. [6] presented a three-step process for building a CFG from scheduled assembly code, which was used as the first stage in the proposed work. The first step of their algorithm partitions the code at labels (entry points) into a set of basic blocks. During this process, they assume all entry points are complete, and no branch targets an instruction without a label. The second step adds edges between basic blocks in the CFG to represent the normal flow of control. Here, they only consider non-pipelined branch instructions, or those that do not appear within the delay slots of other branch instructions. Pipelined branches are handled in the third step using an iterative algorithm that simulates the flow of control for the program by propagating branch and counter information from block to block. Their method is shown to terminate in linear time for CFGs containing only branches with explicit targets. Figure 4 illustrates the CFG generated for the *vectorsum* procedure in Figure 1.

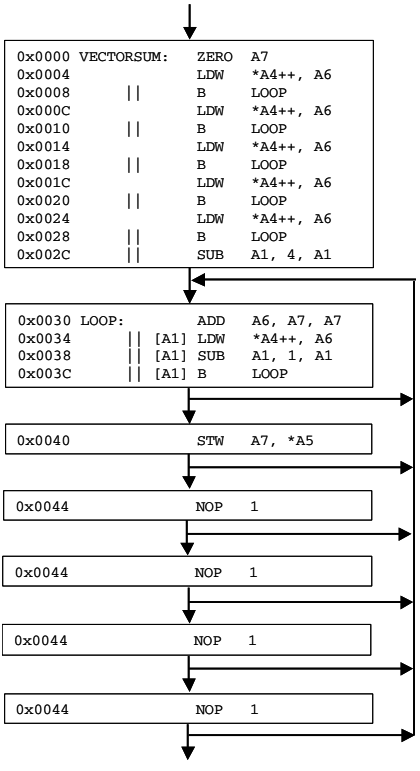

Fig. 4. Control flow graph for *vectorsum*

In practice, the assumptions made in their work pose some difficulties in generating CFGs for some modern processor architectures. For instance, they assume all labels and branch targets are well defined. However, some disassemblers limit the labels to a procedure level only and refrain from including them locally within procedure bounds. In some architectures, registers may be used in branch targets, as in the case of a long jump where a static PC value is loaded into the register prior to the branch instruction. To handle these situations, we introduce a pre-processing step that determines all static branch targets and adds the respective labels to the instructions. Some architectures may also support dynamic branch targets, in which the destination value may be passed to a register as a function parameter, such as with procedure prologues and epilogues. In these situations, we take an optimistic approach by assuming the dynamic branch operation is a procedure call. The branch is temporarily treated as a NOP instruction when building the initial CFG to allow the control flow to propagate through. We rely on post-processing steps, such as alias analysis and procedure extraction to determine the possible destinations [12]. The CFG is then regenerated with the newly identified destination values.

Many of today's processor architectures utilize instruction-level parallelism to achieve higher performances, which complicates generation of CFGs. For instance, a branch destination may have a target within a parallel set of instructions. This would

break up the control flow at intermediate points within a basic block, creating erroneous data dependencies. In Figure 5, the ADD, SUB, and SRL instructions are scheduled in parallel. However, if the predicated branch is taken, the ADD instruction is not executed. Consequently, the entry label on the SUB instruction partitions the control flow in the middle of the parallel set, placing the latter two instructions in a separate basic block. This forces the A7 operand in the SRL instruction to use the resulting value from the ADD instruction in the previous block. To account for such discrepancies, we introduce a procedure that checks for entry points (labels) within a parallel set of instructions. If such an entry point exists, the instructions falling below the entry point are replicated and added to the top portion of the parallel set. Figure 6 shows the MST code after instruction replication. The SUB and SRL instructions have been replicated and a branch operation has been added to jump over the replicated code segment. We rely on subsequent optimizations in the CDFG, such as code-hoisting [13], to eliminate superfluous operations.

```
0x0800        [A1] B     L1
0x0804             NOP   5
0x0808             ADD   A4, A7, A7
0x080C  L1: ||     SUB   A4, 1, A4
0x0810      ||     SRL   A4, A7, A8
0x0814  L2:        ...
```

Fig. 5. Branch target inside a parallel instruction set

```
10.0000  0x0800   [A1] GOTO (6) L1
11.0000  0x0804        NOP  (5) 5
16.0000  0x0808        ADD  (1) $A4, $A7, $A7
16.0100  0x080C        SUB  (1) $A4, 1, $A4    ; replicated SUB
16.0200  0x0810        SRL  (1) $A4, $A7, $A8  ; replicated SRL
16.0300  0x0810        GOTO (0) L2             ; added 'branch-over'
17.0000  0x080C  L1:   SUB  (1) $A4, 1, $A4
17.0100  0x0810        SRL  (1) $A4, $A7, $A8
18.0000  0x0814  L2:   ...
```

Fig. 6. MST representation with instruction replication

4.2 Event-Triggered Operations

In the previous section, a methodology for generating a CFG from pipelined assembly code was presented. The CFG represents the flow of control in the program via edges connecting basic blocks in the graph. However, the CFG does not inherently contain any information regarding propagation delay. In translating pipelined or scheduled assembly code from one architecture to another, it is essential that the compiler capture the propagation delay and data dependencies correctly. Failure to do so may result in false data dependencies, incorrect data value propagation, and possibly an ill-terminated or non-terminating program. Referring back to the *vectorsum* procedure in

Figure 1, we find that the main loop body will execute an unknown number of times until the predicate condition on the branch instruction is *false*, namely, when *A1 = 0*. At that point, the loop will continue to iterate for 5 more cycles until the branches within the pipeline have completed. During this time, data is still computed and propagated through the loop. Should the compiler not consider the propagation delay on the branch instructions, the loop may terminate early, producing erroneous data. Similarly, failure to consider the propagation delay in the pipelined load instructions will also result in erroneous data.

As a solution, we introduce the concept of an *event-triggered* operation, composed of a *trigger* and an *execute* stage. An event *trigger* is analogous to the read stage in a pipelined architecture, where the instruction is fetched and register values are read; an event *execute* is analogous to the write-back stage in the pipeline, during which the values are written to the destination register or memory. The event triggering and execution stages are offset by the delay of the operation.

An operation event is encapsulated in the MST language using a virtual shift register with a precision *d*, corresponding to the number of delay cycles for the operation. Virtual registers are temporary operands created by the compiler that do not exist within the framework of the source architecture's physical registers. In practice, this results in the addition of a very small shift register since most ISAs generally have no more than 4-6 delay slots in any given multi-cycle instruction. When a pipelined instruction is encountered during the normal flow of the program, an event is triggered by assigning a '1' to the highest bit (*d-1*) in the shift register. In each successive cycle, a shift-right logical operation is performed on the register. The event is executed after *d* cycles, when a '1' appears in the zero bit of the shift register.

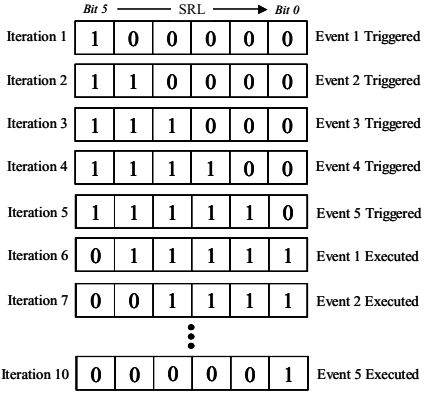

Fig. 7. Event-triggering for a pipelined branch operation in a loop body

Figure 7 illustrates the event triggering for the branch operation in the loop body of the *vectorsum* procedure, which has an operation delay of 6 cycles. In the first iteration of the loop, an event is triggered when the branch instruction is encountered by setting the high bit of shift register. In each subsequent cycle, the register is shifted right while a new event is triggered. After six iterations, event 1 is executed and the

branch to LOOP is taken. This is followed by subsequent event executions through the tenth iteration of the loop, until the pipeline in the shift register has been cleared.

The technique described here is utilized in the linearization process for pipelined operations as discussed in the following sections.

4.3 Linearizing Pipelined Operations

This section describes the linearization process for pipelined operations. The concept of this process is to serialize the pipelined assembly instructions into linear assembly, such that the each pipelined instruction has a well-defined data flow path. The process for linearizing computational operations (arithmetic, logical, memory, etc.) and branch operations are described independently, as they function differently in pipeline architectures. The linearization process assumes that the CFG is complete, i.e., no edges will be inserted between blocks in the future. Consequently, if new edges are added in the future, data propagation and data dependencies are not guaranteed to be correct. To ensure its completeness, we force the algorithm to cover all possible control paths when generating the CFG. This is accomplished in a preprocessing pass that ensures all branch instructions in the program are predicated. A constant predicate of '1', whose condition always resolves to *true*, is added to all non-predicated branch instructions. This forces the branch to be treated as a conditional, and allows the control flow to propagate to the fall-through block. Subsequent optimizations, such as dead-code elimination [13], will remove any resulting extraneous operations.

4.3.1 Linearizing Computational Operations

In the linearization process for computational operations, multi-cycle instructions are serialized into a well-defined data flow path along the pipeline. In order to accomplish this task, virtual registers are introduced to break multi-cycle instructions into a sequence of multiple *single-cycle* instructions. Each instruction in the sequence is guarded by a predicate on an event-triggering register, as described above. Should the program encounter the instruction through a path outside the normal pipeline data flow path, the predicate will prevent the operation from executing.

The linearization process works as follows: For an instruction with n delay slots, the original instruction is modified to write to a temporary virtual register R_n, and the delay of the instruction is changed to a single cycle. In each of the subsequent $n-1$ cycles, the value is propagated through virtual registers along the pipelined data flow path by assigning $R_{n-1} \leftarrow R_n$, $R_{n-2} \leftarrow R_{n-1}$, …, $R_0 \leftarrow R_1$ in sequence, where R_0 is the original register name. Each of these instructions is predicated on its respective cycle bit of the shift register: $P[n-1]$ through $P[0]$. If the end of a basic block is reached, the linearization is propagated to the successor blocks. This approach assumes that no two instructions are scheduled such that both have the same destination register and write-back stages in the same cycle. This is a fair assumption, since compilers generally do not produce code resulting in race conditions. If two or more identical instructions have intersecting pipeline paths, redundant instructions may be avoided by tracking the timesteps to which they have been written. We rely on optimizations, such as copy and constant propagation [13], to remove any extraneous operations.

```
          :                 :
12.000  0x000C            MOVE(0) 1, $P1[4]    ; LD event cycle 1
12.001  0x000C            SRL(1) $P1, 1, $P1
12.002  0x000C  [$P1[4]]  LD(1) *mem($A4), $A6_4
          :                 :
13.000  0x000C            SRL(1) $P1, 1, $P1   ; LD event cycle 2
13.001  0x000C  [$P1[3]]  MOVE(1) $A6_4, $A6_3
          :                 :
14.000  0x000C            SRL(1) $P1, 1, $P1   ; LD event cycle 3
14.001  0x000C  [$P1[2]]  MOVE(1) $A6_3, $A6_2
          :                 :
15.000  0x000C            SRL(1) $P1, 1, $P1   ; LD event cycle 4
15.001  0x000C  [$P1[1]]  MOVE(1) $A6_2, $A6_1
          :                 :
16.000  0x000C  LOOP:     SRL(1) $P1, 1, $P1  ; LD event cycle 5
16.001  0x0014            OR(0) $P1[0], $P2[0], $MP0
16.002  0x001C            OR(0) $MP0, $P3[0], $MP1
16.003  0x0024            OR(0) $MP1, $P4[0], $MP2
16.004  0x0034            OR(0) $MP2, $P5[0], $MP3
16.005  0x000C  [$MP3]    MOVE(1) $A6_1, $A6  ; intersecting LDs 1-5
          :                 :
```

Fig. 8. Linearization of pipelined load (LD) instruction in the *vectorsum* procedure

Figure 8 illustrates the linearization process in the MST for the first pipelined LD instruction in the *vectorsum* example of Figure 1. In timestep 12, an event is triggered for the LD instruction by posting a '1' to the high bit in the virtual shift register *P1*. Additionally, the LD instruction is modified to write to virtual register *A6_4*, and the operation delay is changed from 5 cycles to 1 cycle. In the subsequent cycles, *A6_4* is written to *A6_3*, *A6_3* is written to *A6_2*, and *A6_2* is written to *A6_1*, at which point the linearization is propagated to the *LOOP* block. *A6_1* is finally written to the physical register *A6* in timestep 16. Each of these move instructions is guarded by a predicate on a *P1* bit, which is right-shifted in each cycle along the same control path. The same methodology is applied to each LD instruction in program. Although the propagation instructions may read and write to the same register in parallel, the one-cycle delay on each instruction enforces the correct data dependencies.

It is interesting to note that the pipelined LD instructions have intersecting paths. As an example, all five LD instructions will have their 5^{th} cycles intersect in the same timestep (16), where *A6* ← *A6_1*. To avoid extraneous instructions, the propagation instructions are merged by OR-ing their predicates, as shown in the figure.

4.3.2 Linearizing Branch Operations

Unlike computational instructions, branch instructions do not propagate data. Rather, they trigger a change in control flow after a certain number of delay cycles. In linearizing branch operations, only the *event* is propagated through the CFG, as

described above. At each branch execution point in the CFG, which can only be the end of a basic block, a copy of the branch instruction is inserted. The branch instruction is predicated on the event shift-register. Similar to the process above, if two or more of the same branch instruction have intersecting paths, redundant instructions may be eliminated by tracking the timesteps to which the instructions have been written. Two or more of the same branch instruction that execute at the same point can be merged by OR-ing their predicates. The original branch instructions are replaced with NOP instructions in order to maintain the correct instruction flow. Figure 10 illustrates the linearization process for pipelined branch operations.

```
          :            :
11.000  0x0008         MOVE(0) 1, $P1[5]    ; branch event cycle 1
11.001  0x0008         SRL(1)  $P1, 1, $P1
11.002  0x0008         NOP(1)  1            ; branch replaced with NOP
          :            :
12.000  0x0008         SRL(1) $P1, 1, $P1   ; branch event cycle 2
          :            :
13.000  0x0008         SRL(1) $P1, 1, $P1   ; branch event cycle 3
          :            :
14.000  0x0008         SRL(1) $P1, 1, $P1   ; branch event cycle 4
          :            :
15.000  0x0008         SRL(1) $P1, 1, $P1   ; branch event cycle 5
          :            :
16.000  0x0008  LOOP:  SRL(1) $P1, 1, $P1   ; branch event cycle 6
16.008  0x0008         OR(0) $P1[0], $P2[0], $MP0
16.009  0x0010         OR(0) $MP0, $P3[0], $MP1
16.010  0x0018         OR(0) $MP1, $P4[0], $MP2
16.011  0x0020         OR(0) $MP2, $P5[0], $MP3
16.012  0x0028         OR(0) $MP3, $P6[0], $MP4
16.013  0x003C  [$MP4] GOTO(0)  LOOP   ; intersection branches 1-6
          :            :
```

Fig. 9. Linearization of a pipelined branch instruction in the *vectorsum* procedure

4.3.3 The Linearization Algorithm

Figure 9 presents our algorithm for linearizing pipelined operations. The procedure has the same general organization as the algorithm presented by Cooper et al. [6] for generating CFGs. The algorithm initially creates a *worklist* of *instruction counters* for each basic block in the CFG in lines 1-3, and then iterates through the worklist in lines 4-25. An instruction counter is particular to a block, and holds a list of pending instructions and a counter representing the remaining clock cycles before each instruction is executed. To prevent redundant iterations over blocks, in lines 8-9, the algorithm checks that the block has not seen any of the pending instruction counters before continuing. The algorithm then iterates over the block by *whole timesteps* in

lines 10-20. The instructions in each timestep are iterated through in lines 11-17, as the algorithm searches in line 12 for previously unvisited pipelined instructions to add to the instruction counter. Lines 13-15 add a counter for the branch instructions with cycle delays greater than zero; the original branch instruction is replaced with a NOP instruction to maintain the correct program flow. Lines 16-17 add counters for all multi-cycle instructions whose write-back time falls outside the block. Unique *event* instructions are inserted for each pending instruction in lines 18-20, as described above; those that have completed are removed from the instruction counter list. After iterating over the instructions within each timestep, the pending instruction counters are decremented in line 21. At the conclusion of the iteration over timesteps in the block, lines 22-26 propagate all pending counters to new instruction counters for each successor block edge; the new instruction counters are added to the worklist. The algorithm terminates once no new instruction counters are encountered by any block and the worklist is empty. The algorithm runs in $O(n)$ time, where n is the number of instructions in the program, assuming a small, constant number of outgoing edges between blocks.

```
 Linearize_Pipelined_Operations( CFG )
 1  worklist = empty list of InstrCounters
 2  for each basic block in CFG do
 3    add InstrCounter(block) to worklist
 4  while worklist->size() > 0 do
 5    instr_counter = worklist->front()
 6    remove instr_counter from worklist
 7    block = instr_counter->block
 8    if block has seen all live counters in instr_counter then
 9      continue
10    for each whole timestep ts in block do
11      for each instruction i in timestep ts do
12        if i has not been seen by instr_counter then
13          if i is a branch instruction and i->delay > 0 then
14            add {i:i->delay} to instr_counter
15            replace branch instruction i with NOP instruction
16          else if (i->timestep + i->delay) > block->max_time
17            add {i:i->delay} to instr_counter
18        for each counter c in instr_counter do
19          insert a unique event instruction for c in timestep ts
20          if c = 0 then remove c from instr_counter
21      instr_counter->DecrementCounters()
22    if instr_counter has live counters
23      for each successor s of block do
24        target_instr_counter = InstrCounter(s)
25        add unique live counters to target_instr_counter
26        add target_instr_counter to worklist
```

Fig. 10. Linearization algorithm for pipelined operations

4.4 Generating the Control and Data Flow Graph

In the previous sections we described how to build a CFG and break data dependencies in pipelined and scheduled assembly code. In this section we combine the two techniques to generate the complete CDFG. The procedure is described in Figure 12, which takes a list of assembly instructions as input and returns a CDFG. The procedure begins with a preprocessing step to ensure that all branch instructions

in the program are predicated as described in the previous section. The algorithm constructs the CFG using Cooper's algorithm, and then linearizes the pipelined operations as described above. The data flow graph is then generated from the newly serialized instructions, based on the data dependency analysis technique described in Section 3. The procedure concludes by implementing single static-variable assignment (SSA) [13], which is a method of breaking data dependencies by ensuring that every assignment in the CDFG has a unique variable name.

Traditionally, a *Φ-function* is used in SSA to join multiple assignments to a variable, stemming from different paths in the CFG. The number of arguments to the *Φ-function* is equal to the number of definitions of the variable from each point in the CFG. This method often causes a significant bottleneck when handling numerous data paths. Interestingly, once the pipelined operations in the CDFG have been linearized, the *Φ-function* becomes superfluous, as only the latest definition of a variable will reach the end of the block and propagate through the control flow. Those instructions with multi-cycle delays that originally crossed basic block boundaries have since been serialized into multiple single-cycle instructions. As a result, the latest definition of each SSA variable may be assigned back to its original variable name at the end of the block, thus eliminating the need for the *Φ-function*. Optimizations, such as copy propagation and dead-code elimination [13], will remove extraneous assignment operations created by this process.

```
  Generate_CDFG( instr_list )
1  Predicate_Pipelined_Instrs( CFG )
2  CFG = Generate_Ctrl_Flow_Graph( instr_list )
3  Linearize_Pipelined_Operations( CFG )
4  CDFG = Generate_Data_Flow_Graph( CFG )
5  Generate_SSA( CDFG )
6  return CDFG
```

Fig. 11. Procedure for generating a CDFG

5 Experimental Results

The correctness of the methodology presented in this paper was verified using the FREEDOM compiler [11,19] on 8 highly pipelined benchmarks in the Texas Instruments C6000 DSP assembly language. The FREEDOM compiler generated CDFGs and RTL code targeting the Xilinx Virtex II FPGA. Each benchmark was simulated using Mentor Graphic's ModelSim to verify bit-true accuracy and obtain cycle counts.

There has been little work reported on translating highly pipelined software binaries to RTL code for FPGAs. This makes comparison with other approaches difficult. However, it is interesting to consider the impact and effectiveness of this algorithm in a high-level synthesis tool. Table 1 shows comparisons in cycle counts for the TI C6000 DSP and the Virtex II FPGA, generated by the FREEDOM compiler. Also shown is the number of pipelined operations in each benchmark and

the number of instructions inserted during the linearization process to demonstrate the impact on code size when using this approach.

Results indicate the FREEDOM compiler successfully generated the correct CDFGs from the pipelined assembly code, allowing complex optimizations and scheduling to significantly reduce clock cycles in the FPGA design. On average, approximately 9 instructions were added for each pipelined operation and there was a 27% increase in code size during the linearization process. *Please note that these values reflect the size of the design before CDFG optimizations, which will further reduce implementation complexity. A detailed evaluation of the performance and optimizations of the FREEDOM compiler has been presented in other work [11,19].*

Table 1. Experimental results on pipelined benchmarks using the FREEDOM compiler

Benchmark	DSP Cycles	FPGA Cycles	# Pipelined Instructions	# Added Instructions
memmove	125747	2516	33	352 (24.7%)
memcpy	69615	2004	14	136 (52.3%)
divi	282301	16127	17	141 (27.3%)
mpyd	1329176	39669	26	269 (14.0%)
remi	260148	16888	13	130 (34.6%)
dsp_fir_gen	30851	685	49	683 (43.1%)
lms_filter	33537580	773288	147	967 (13.7%)
noise_canceller_fir	8239397	163778	21	105 (5.3%)

6 Conclusions

This paper presents a methodology for correctly representing the data dependencies and data propagation when generating CDFGs from highly pipelined and scheduled assembly code. This process consists of three stages: generating a control flow graph, linearizing the assembly code, and generating the data flow graph. We use a known method for generating the control flow graph from scheduled assembly code and describe further techniques for handling more complex architectures that employ parallel instruction sets and dynamic branching. We present a linearization process, in which pipelined structures are serialized into linear assembly. This allows for proper data dependency analysis when generating the data flow graph.

The work was verified in the FREEDOM compiler on 8 highly pipelined software binaries for the TI C6000 DSP, targeting the Xilinx Virtex II FPGA. Results indicate that data dependencies were correctly identified, enabling the compiler to perform complex optimizations and scheduling to reduce clock cycles in the designs.

References

1. Amme W, Braun P, Thomasset F, and Zehendner E (2000) Data Dependence Analysis of Assembly Code. International Journal of Parallel Programming, vol. 28, issue 5.
2. Banerjee U (1988) Dependence Analysis for Supercomputers. Kluwer Academic Publishers, Norwell, MA.

3. Cifuentes C and Gough K (1993) A Methodology for Decomposition. Proceedings for XIX Conferencia Latinoamericana de Informatica. Buenos Aires, Argentina, pp 257-266.
4. Cifuentas C and Malhotra V (1996) Binary Translation: Static, Dynamic, Retargetable? Proceedings for the International Conference On Software Maintenance (ICSM). Monterey, CA, pp 340-349.
5. Cifuentes C, Simon D, and Fraboulet A (1998) Assembly to High-Level Language Translation. Proceedings of the International Conference on Software Maintenance (ICSM). Washington, DC, pp 228-237.
6. Cooper K, Harvey T, and Waterman T (2002) Building a Control-Flow Graph from Scheduled Assembly Code. Technical Report 02-399. Department of Computer Science, Rice University, Houston, TX.
7. Decker B and Kästner D (2003) Reconstructing Control Flow from Predicated Assembly Code. Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES). Vienna, Austria, pp 81-100.
8. Kästner D and Wilhelm S (2002) Generic Control Flow Reconstruction from Assembly Code. Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES), vol. 37, issue 7, pp 46-55.
9. Kruegel C, Robertson W, Valeur F, and Vigna G (2004) Static Disassembly of Obfuscated Binaries. Proceedings of USENIX Security 2004. San Diego, CA, pp 255-270.
10. Levine B and Schmidt H (2003) Efficient Application Representation for HASTE: Hybrid Architectures with a Single Executable. Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa, CA, pp 101-107.
11. Mittal G, Zaretsky D, Tang X, and Banerjee P (2004) Automatic Translation of Software Binaries onto FPGAs. Proceedings of the 41st Annual Conference on Design Automation. San Diego, CA, pp 389-394.
12. Mittal G, Zaretsky D, Memik G, and Banerjee P (2005) Automatic Extraction of Function Bodies from Software Binaries. Proceedings for the IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC). Beijing, China.
13. Muchnick S (1997) Advanced Compiler Design Implementation. Morgan Kaufmann Publishers, San Francisco, CA.
14. Ramsey N and Fernandez M (1995) New Jersey Machine-Code Toolkit. Proceedings of the 1995 USENIX Technical Conference. New Orleans, LA, pp 289-302.
15. Ramsey N and Fernandez M (1997) Specifying Representations of Machine Instructions. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 19, issue 3. New York, NY, pp 492-524.
16. Stitt G and Vahid F (2003) Dynamic Hardware/Software Partitioning: A First Approach. Proceedings of the Design Automation Conference. Anaheim, CA, pp 250-255.
17. Stitt G and Vahid F (2002) Hardware/Software Partitioning of Software Binaries. Proceedings of the International Conference of Computer Aided Design (ICCAD). Santa Clara, CA, pp 164-170.
18. Ye Z, Moshovos A, Hauck S, and Banerjee P (2000) CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. Proceedings of the 27th International Symposium on Computer Architecture. Vancouver, Canada pp 225-235.
19. Zaretsky D, Mittal G, Tang X, and Banerjee P (2004) Overview of the FREEDOM Compiler for Mapping DSP Software to FPGAs. Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines. Napa, CA, pp 37-46.