Coordinated Transformations for High-Level Synthesis of High Performance Microprocessor Blocks

Sumit Gupta[*] Timothy Kam[§] Michael Kishinevsky[§] Shai Rotem[§]
Nick Savoiu[†] Nikil Dutt[†] Rajesh Gupta[†] Alex Nicolau[†]

[†]Center for Embedded Computer Systems, University of California, Irvine
{sumitg,savoiu,dutt,rgupta,nicolau}@cecs.uci.edu; http://www.cecs.uci.edu/∼spark

[§]Strategic CAD Labs, Intel Incorporated, Hillsboro, Oregon
{timothy.kam,michael.kishinevsky,shai.rotem}@intel.com; http://www.intel.com/research/scl

ABSTRACT

High performance microprocessor designs are partially characterized by functional blocks consisting of a large number of operations that are packed into very few cycles (often single-cycle) with little or no resource constraints but tight bounds on the cycle time. Extreme parallelization, conditional and speculative execution of operations is essential to meet the processor performance goals. However, this is a tedious task for which classical high-level synthesis (HLS) formulations are inadequate and thus rarely used. In this paper, we present a new methodology for application of HLS targeted to such microprocessor functional blocks that can potentially speed up the design space exploration for microprocessor designs. Our methodology consists of a coordinated set of source-level and £negrain parallelizing compiler transformations that targets these behavioral descriptions, speci£cally loop constructs in them and enables ef£cient chaining of operations and high-level synthesis of the functional blocks. As a case study in understanding the complexity and challenges in the use of HLS, we walk the reader through the detailed design of an instruction length decoder drawn from the *Pentium*® -family of processors. The chief contribution of this paper is formulation of a domain-speci£c methodology for application of high-level synthesis techniques to a domain that rarely, if ever, £nds use for it.

Categories and Subject Descriptors: B.5.1 [Register-Transfer-Level Implementation] Design Aids C.5.3 [Computer System Implementation] Microcomputers – Microprocessors
General Terms: Design
Keywords: High-level synthesis, microprocessor design

1. INTRODUCTION

The classical high-level synthesis problem is one of transforming a behavioral description of an application through scheduling and binding tasks with constraints on the number of resources, into a multi-cycle schedule of operations. HLS is currently a mature £eld with a number of university and commercially-available tools. The current application of HLS is heavily concentrated on the design of moderately complex ASIC designs. The target architecture is a multi-cycle design with latencies in the 10s and 100s of cycles. These designs are usually area constrained, which often limits the extent of parallelism in operations the £nal design can support. Pipelining is generally the preferred means to improve system performance though it does not help latency. Accordingly, support for automatic pipelining (to a limited extent) of behavioral descriptions can be found in most commercial HLS tools.

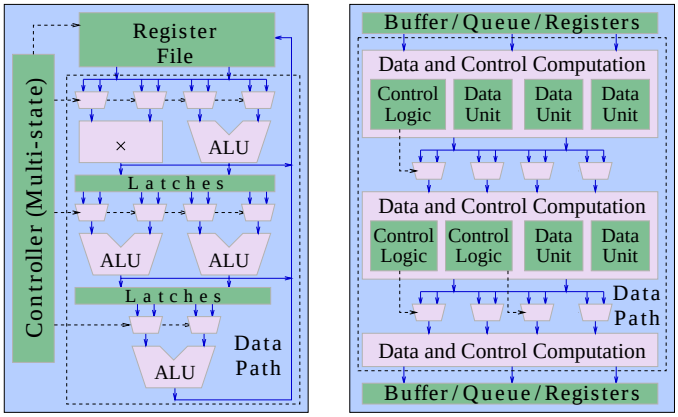
High performance microprocessor designs are typically considered to lie on the other end of the spectrum where much of the HLS optimizations in scheduling, resource binding and allocation do not £nd much use. There exist a good number of functional blocks within microprocessors, which are most naturally and succinctly described by a behavioral description. However, the lack of responsiveness to design constraints in HLS formulations leads to little or no use of traditional HLS tools in such high-performance functional blocks. The chief problem is that, one major microprocessor design challenge – especially in the high end – is of identifying maximum parallelism and creating additional parallelization opportunities above and beyond afforded by the algorithmic speci£cation, and then packing all the resulting operations in a safe manner in the smallest number of cycles and in the shortest cycle time. Pure pipelining is of limited value since functional block latencies are critical in the presence of signi£cant control in the behavior.

Our work in this area has been motivated by the advances in parallelizing compiler technology that enable exploitation of extreme amounts of parallelization through a range of code motion techniques. While we have found no single code motion technique (including the ones we have developed speci£cally for HLS) to be universally useful, we have found that a judicious balance of a number of these techniques driven by well considered heuristics is likely to yield HLS results that compare in quality to the manually designed functional blocks. The challenge then is to identify and isolate a useful set of transformations and couple these with the rest of a high-level synthesis system. This system then provides a working environment for the microprocessor block designer to explore alternative designs and speed up the overall design process. This in essence is the contribution of the Spark synthesis system.

This paper demonstrates the utility of the Spark system through a case study of an instruction length decoder block derived from an advanced microprocessor. The choice of this block is made for a few reasons: (a) it is moderately complex and yet small enough that a detailed walk through the design – in an attempt to understand the challenges in application of HLS to high-performance microprocessor block designs – is possible; (b) the synthesis of this design employs several parallelizing transformations that validate the underlying motivation for building Spark; (c) designs of this nature are most naturally described by a behavioral description rather than a structural model, making them ideal for HLS.

[*]This author was an intern at Intel in the summer of 2001.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.



(a) (b)

Figure 1: Generic architectures (a) for ASIC designs and (b) for high performance microprocessor blocks

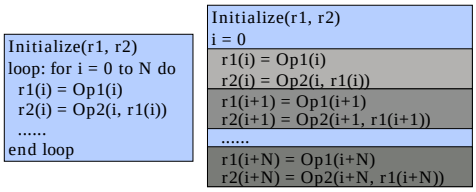
We can begin our understanding of the microprocessor synthesis domain, by comparing the generic architectures of microprocessor blocks and ASICs. ASICs (as shown in Figure 1(a)) are typically multi-cycle and pipelined, consisting of several functional units, steering logic (multiplexors), a controller (often a £nite state machine) and a register £le. Intermediate results are usually stored in latches and inter-stage forwarding paths may exist in the data path. On the other hand, as shown in Figure 1(b), microprocessor blocks are often single cycle and have several small computation blocks that operate in tandem and whose results are steered by, and even used to generate, control logic. Inputs and outputs to these type of blocks are stored in memory elements such as buffers and queues.

Keeping these differences in mind, we present Spark's synthesis methodology for microprocessor blocks. The rest of this paper is organized as follows: the next section reviews relevant previous work, followed by an outline of Spark's synthesis strategy and several of its transformations. The Spark system itself is brie¤y described in Section 4. Sections 5 and 6 describe the instruction length decoder and the steps employed in synthesizing this design. We conclude with a discussion and an outline of future work.

2. PREVIOUS WORK

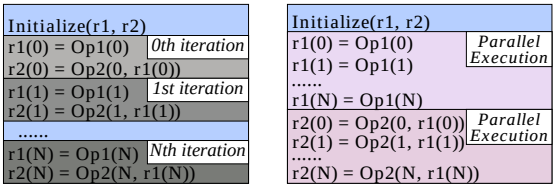
There has been much work in the £eld of high-level synthesis (HLS). Early work concentrated on data-¤ow designs and applied optimizations such as algebraic transformations, retiming and code motions across multiplexors for improved synthesis results [1, 2]. Pipelining has been the primary technique to improve performance [3, 4]. Subsequent work has demonstrated the use of speculative code motions on mixed control-data ¤ow type of designs to reduce schedule lengths [5, 6, 7, 8, 9].

A range of code motion techniques similar to those required for HLS has also been developed previously for software compilers (especially parallelizing compilers) [10, 11]. Although the basic transformations (e.g. dead code elimination, copy propagation) can be used in synthesis as well, other compiler transformations need to be re-instrumented for synthesis. They have to be modi£ed to incorporate the ideas of resource sharing, steering logic and control costs, when used for synthesis. Unlike compilers, in synthesis, mutually exclusive operations can be scheduled in the same clock cycle on the same resource. Also, mapping an operation to a resource can lead to the generation of additional steering logic and associated control logic. So, the cost models in compilers and synthesis tools for the various transformations have to be different. Previous work in HLS for microprocessor designs is limited. Brayton et al. [12] synthesized the 801 processor; a small processor with a simple data path, whereas Gupta et al. synthesized long latency functional units to embed into VLIW processors [13].



(a) (b)

Figure 2: (a) A sample piece of pseudo-code with a loop and some operations (b) the loop is completely unrolled



(a) (b)

Figure 3: (a) Constant propagation of loop index variable (b) all Op_1 operations are executed in parallel followed by concurrent execution of all the Op_2 operations

3. SYNTHESIS TRANSFORMATIONS FOR MICROPROCESSOR BLOCKS

Among the parallelizing transformations, the most useful are a number of loop transformations and beyond basic block code motions such as speculation and conditional speculation. In speculative execution, operations are executed before the conditions they depend on, have been evaluated. Conditional execution duplicates operations into the branches of conditional blocks to enhance resource utilization. These transformations have been explored and extended to a set of code motions that include reverse speculation and early condition execution [9, 14] and have been shown to be effective in improving the quality of synthesis results. The effectiveness of these code motions is often limited by the number of resources available to the design; a constraint that is more lax for microprocessor blocks.

The scope for application of code motions can be further increased by loop transformations such as *loop unrolling*. Loop unrolling was developed to enable software compilers to perform optimizations across loop iterations and facilitate global code optimizations [15]. However, loop unrolling can lead to code explosion; so, loops are unrolled one iteration at a time, followed by code compaction by parallelizing transformations, until no further improvements can be obtained. Loops are seldom unrolled fully.

On the other hand, for microprocessor functional blocks, loops are only a programming convenience and latency constraints generally dictate the amount of unrolling a loop has to undergo. For instance, if a design is targeted to, say, three clock cycles, it implies that *all* the operations within *all* the iterations of the loop have to be executed in these three cycles. Hence, when this design is mapped to hardware, it will generate a design in which the loop is, in essence, unrolled within these three cycles. Loops in single cycle designs must, of course, be unrolled completely.

Loop unrolling is demonstrated in Figure 2. Figure 2(a) shows the pseudo-code of a synthetic example, which has a loop and some operations within this loop. Operations Op_1 use the loop index variable i and some other inputs (not shown) to generate a set of results $r1$. These results are used by operations Op_2 to generate the £nal results $r2$ of the loop. This loop can be unrolled completely, i.e., N times to obtain the pseudo-code shown in Figure 2(b). In this £gure, only the ith, $(i+1)$th and $(i+N)$th iterations are shown.

The Op_1 operations still have a dependency with the loop index variable i. However, since the loop is unrolled completely, the

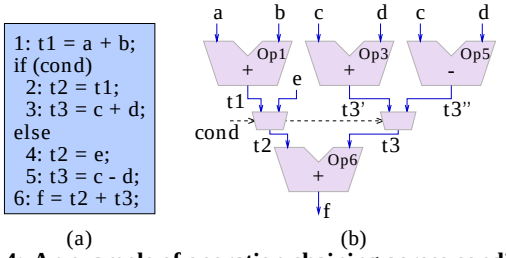


Figure 4: An example of operation chaining across conditional boundaries; (a) sample "C" code (b) corresponding hardware, with functional units connected via steering logic.

value of the loop index variable is known statically in all the iterations. Hence, the initial value assigned to the loop index variable can be propagated as a constant throughout all the iterations. This is known as *constant propagation* [16] and is shown in Figure 3(a) for the example from Figure 2. In this £gure, the constant assignment $i = 0$ has been propagated through all the unrolled iterations and the loop index variable is completely eliminated from the code. This frees up more operations for the application of code parallelizing transformations. In this example, the code motion transformations can execute the Op_1 operations concurrently followed by the concurrent execution of all Op_2 operations (assuming that the resources to do so are available), as shown in Figure 3(b).

The coarse-grain transformations presented above can also be coupled with £ne-grain transformations, such as copy propagation, dead code elimination and other standard compiler transformations. Also, another enabling transformation for synthesis is operation chaining, which is discussed in the next section.

3.1 Chaining Operations Across Conditional Boundaries

Operation chaining is an important technique that is supported by most high-level synthesis tools [17]. *Chaining* of operations means that the result of one operation is used immediately by the next operation without storing it in an intermediary latch or register. In the generated hardware, the resources corresponding to the operations have to be connected to each other in combinational blocks. However, in designs that are a mix of control and data operations, such as microprocessor functional blocks, these resources are often connected via steering logic such as multiplexors. In terms of the behavioral description, this implies that operations have to be chained across basic blocks, i.e., across conditional blocks.

Consider the sample fragment of "C" code in Figure 4(a). To achieve a single cycle schedule for this description, all the operations in the description have to be chained together, across the if-then-else conditional block. One possible hardware implementation for this is shown in Figure 4(b). The operations Op_1 to Op_6 correspond to the line numbers in Figure 4(a). As shown in this £gure, the inputs to the operation Op_6 are obtained by multiplexing the outputs of the Op_1, Op_3 and Op_5, based on the condition *cond*.

This example demonstrates that operations in several basic blocks may be scheduled in the same clock cycle. Thus, resource utilization in a clock cycle has to keep track of the resource utilization of multiple operations in several basic blocks. Hence, scheduling with operation chaining across conditional boundaries has to use a modi£ed resource utilization and operation scheduling model that looks across the conditional boundaries. Furthermore, chaining an operation with operations that are in the branches of a conditional check requires a more detailed analysis.

3.1.1 Chaining Operations with Operations in the Branches of a Conditional Block

Consider the hierarchical task graph (HTG) [14] representation of a design in Figure 5; we want to schedule operation 4 in the same

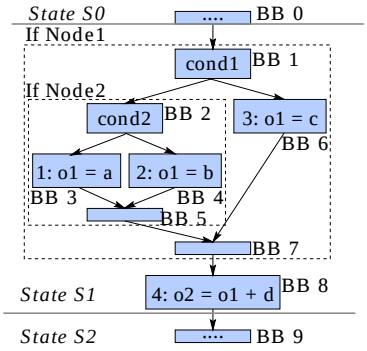


Figure 5: Operation 4 is scheduled in the same cycle as operations 1, 2 and 3. Hence, we have to check that chaining is possible on all *trails* up from basic block BB_8.

cycle as operation 1. The chaining heuristic has to validate that operation 4 can be chained with the other operations in this cycle. It traverses all the paths or *trails* backwards from the basic block that operation 4 is in (BB_8), looking for operations that are scheduled in the same cycle. In this example, there are three trails comprising the basic blocks: $<BB_8, BB_7, BB_5, BB_3, BB_2, BB_1>$,$<BB_8, BB_7, BB_5, BB_4, BB_2, BB_1>$ and $<BB_8, BB_7, BB_6, BB_1>$. These trails have the three operations 1, 2 and 3 respectively, each of which writes to the variable $o1$. The heuristic determines that operation 4 can be executed in the same cycle by using the appropriate value of $o1$ depending on the evaluation of the condition.

Besides checking that there is enough time in the cycle for chaining the operation with other operations in the chaining trails, the chaining heuristic also has to ensure that the correct hardware is generated to implement the schedule, as discussed next.

3.1.2 Creating Wire-Variables to enable Chaining on each Chaining Trail

The Spark synthesis tool initially assumes that each variable in the input behavioral description is mapped to a virtual register. After scheduling, during register binding, a variable life-time analysis pass determines which variables are actually mapped to registers. However, since registers can only be read in the next cycle after being written, therefore, to enable operation chaining, we introduce the notion of a *wire-variable*. Wire-variables are explicitly marked as being wires and are not mapped to registers, and thus, can be read in the same cycle as they are written to.

Consider an operation Op_1 that writes a result, r_1 and another operation Op_2 that reads this result, i.e., a situation that looks like: $r_1 = Op_1(arguments); r_2 = Op_2(r_1)$. To chain operations Op_1 and Op_2, the code has to be modi£ed to: $temp = Op_1(arguments); r_2 = Op_2(temp); r_1 = temp$, where variable *temp* is marked as being a wire and r_1 is (potentially) mapped to a register[1].

Often, as was the case in the example in Figure 5, a variable may be written by several operations in different basic blocks. When operations are chained across conditional checks, writes to "wire-variables" have to be inserted in all the trails leading back from the chained operation, i.e., in all the branches of the preceding conditional blocks. This is explained by an example in Figure 6(a). In this hierarchical task graph (HTG) representation, variable $o1$ is written to by operations 1 and 2 in basic blocks BB_0 and BB_2 respectively. Operation 3 in basic block BB_5 reads the value of this variable to produce another variable $o2$. Consider that the scheduling algorithm schedules the entire fragment of code in this £gure within one clock cycle. Then, to enable the operation chaining, a wire-variable $t1$ is introduced and the copy operations 4 and 5 are

[1]In the RTL VHDL generated after synthesis, r_1 is mapped to a VHDL signal and *temp* is mapped to a VHDL variable

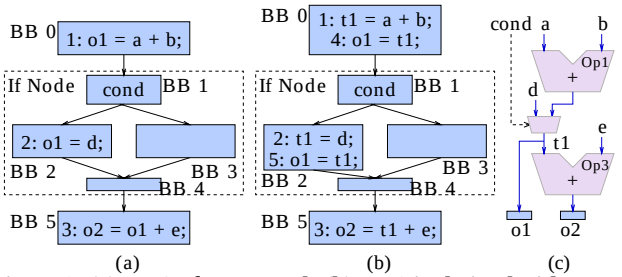


Figure 6: (a) HTG of an example (b) op. 3 is chained with ops. 1 and 2; so, wire-variable $t1$ and copy ops. 4 and 5 are inserted (c) corresponding hardware; $t1$ becomes a wire and $o1$ a register.

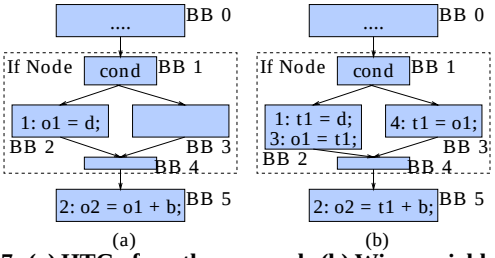


Figure 7: (a) HTG of another example (b) Wire-variable $t1$ and copy operations (3 and 4) are added in all chaining trails.

inserted, as shown in Figure 6(b). In the resulting hardware, shown in Figure 6(c), variable $t1$ becomes a wire and the variables $o1$ and $o2$ are bound to registers. Operation 3 uses the multiplexed result of both the operations that write to wire-variable $t1$.

Similarly consider the fragment of code in the Figure 7(a). In this example, variable $o1$ is written to only in the true branch of a conditional block and is read by operation 2 in basic block BB_5. This code implies that if the condition evaluates to "false", then a value of $o1$ from a previous write (not shown here) will be used by operation 2. In order to chain the operations in this code, a variable copy to wire-variable $t1$ has to be inserted in both branches of the conditional block, as shown in Figure 7(b). So, the operation 2 now reads the variable $t1$ instead. In hardware, variable $t1$ will be mapped to a wire and variable $o1$ to a register. In this way, wire-variables are introduced as and when required and a dead code elimination pass later removes any unnecessary variables and variable copies.

4. THE SPARK SYNTHESIS SYSTEM

The code transformations presented in this paper have been implemented in a high-level synthesis research framework called *Spark* [14, 9]. This synthesis system takes a behavioral description in ANSI-C as input and generates synthesizable register-transfer level VHDL. This enables the system to evaluate the effects of several coarse and £ne-grain optimizations on logic synthesis results. Code motion techniques such as Trailblazing [18] are used to enable the parallelizing, speculative code motion transformations. Loop unrolling has been implemented as part of the resource directed loop pipelining (RDLP) technique [11]. These transformations are supported by standard compiler transformations such as constant propagation and dead code elimination.

The rich set of tunable transformations in Spark enable the system to aid in exploration of several alternative designs. Although Spark can apply the various transformations automatically, it also allows the designer to control the various passes and the degree of parallelization through script £les. For example, the designer may specify which loops to unroll and by how much. This enables Spark to provide design alternatives that may not be obvious to a designer from the design's behavioral description. In the next few sections, we show how we have used Spark to explore the architecture of a functional block from a modern microprocessor.

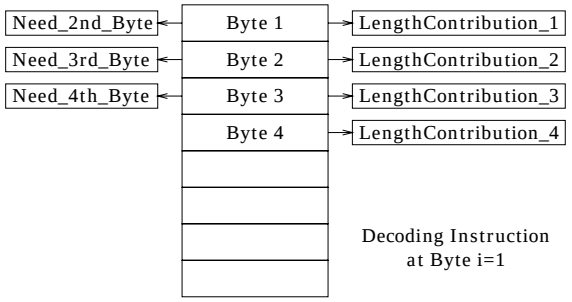


Figure 8: Instruction Length Decoder (ILD) of a modern microprocessor; calculating the length of the £rst instruction

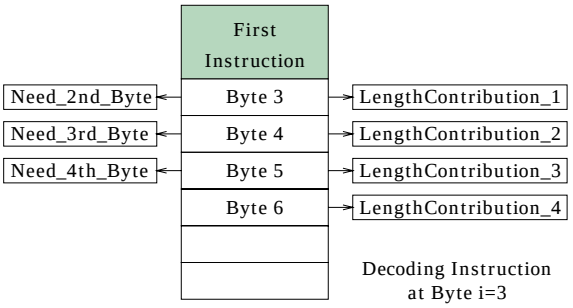


Figure 9: ILD: Calculating the length of the second instruction

5. A CASE STUDY: INSTRUCTION LENGTH DECODER

An important component of the *Pentium*® microprocessor architecture is the *instruction length decoder* [19]. This component determines the starting byte and the length of each instruction from a stream of bytes it receives from the instruction cache. We consider an implementation of this microprocessor architecture in which the instructions can be of variable length ranging from 1 to 11 bytes and the decoder has to look at up to 4 bytes to determine an instruction's length. Instead of processing a stream of bytes, the decoder can alternatively look at a set of bytes in an instruction buffer at every cycle. The instruction length decode then works as shown in Figure 8. The decoder looks at the length contribution of the £rst byte ($LengthContribution_1$), and checks to see whether it needs to look at the next byte as well (*Need_2nd_Byte*). If it does, then it looks at $LengthContribution_2$ of the second byte, and checks to see if it needs the third byte, and so on. In this way, say, the ILD calculates that the £rst instruction is two bytes long, then it must determine the length of the next instruction by (potentially) looking at the next 4 bytes as shown in Figure 9. This continues until the length of all the instructions in the buffer are determined.

A representation of this behavior in "C" is shown in Figure 10. In this code, a loop (indexed by i) iterates over the entire instruction buffer (of size n). If the start of the current instruction *NextStartByte* is the current byte i, then, it marks this as the starting point of an instruction ($Mark[i] = 1$) and calculates the length of the instruction at that byte by calling the function $CalculateLength(i)$. This function is the same as the behavior described above[2]. The £nal output of this description is a bit vector ($Mark[1..n]$) that contains a 1 at only those bit positions where an instruction starts.

There are several simpli£cations in this model of the ILD [19]. Since the ILD is decoding a stream of instructions arriving from memory, the behavioral description should have an in£nite outer loop, that synthesis should break into chunks of n iterations each. Also, consider that an instruction starts at the $(n-1)$th byte. Then the length calculation may need to check bytes from the next set

[2]We assume a zero length contribution from the $n+1$ to $n+3$ bytes

```
ResetArray(Mark);
NextStartByte = 1;
for (i = 1; i <= n; i++) {
  if (i == NextStartByte) {
    Mark[i] = 1;
    len[i] = CalculateLength(i);
    NextStartByte += len[i];
  }
}
int CalculateLength(i) {
  lc1 = LengthContribution_1(i);
  if (Need_2nd_Byte(i)) {
    lc2 = LengthContribution_2(i+1);
    if (Need_3rd_Byte(i+1)) {
      lc3 = LengthContribution_3(i+2);
      if (Need_4th_Byte(i+2)) {
        lc4 = LengthContribution_4(i+3);
        Length = lc1 + lc2 + lc3 + lc4;
      } else Length = lc1 + lc2 + lc3;
    } else Length = lc1 + lc2;
  } else Length = lc1;
  return Length;
}
```

Figure 10: Behavioral "C" code for the ILD

of bytes that £ll the buffer. So, the intermediate length calculation information must be saved across buffer decodes and passed to the next cycle. These simpli£cations are made to keep the discussion focused on the important code transformations used and do not alter the nature and applicability of the transformations presented here.

The processor architectural requirements imply that the whole buffer must be decoded in one cycle. Hence, a designer may choose to compute as much as possible in parallel and then, do the instruction marking after all the information has been calculated. The following section describes how Spark's synthesis methodology achieves this kind of a single cycle architecture for the decoder starting from a natural behavioral description.

6. TRANSFORMATIONS APPLIED BY SPARK TO SYNTHESIZE THE DECODER

In order to achieve a single cycle architecture for the ILD design, the Spark synthesis tool is given an unlimited resource allocation and full freedom to unroll loops. The £rst transformation Spark applies, tries to speculatively compute all the data and control calculations in the function *CalculateLength*. As shown in Figure 11, the length contributions due to the bytes, i through $i+3$, are calculated speculatively and so are the control variables *need*2 to *need*4, that determine which bytes contribute to the length of the current instruction. The lengths of the instruction for each case of these control variables (*TempLength*1 to *TempLength*3) are also speculatively computed. This results in a behavior where all the data calculation is performed up-front and speculatively, followed

```
int CalculateLength(i) {
  lc1 = LengthContribution_1(i);          Data
  lc2 = LengthContribution_2(i+           Calculation
  lc3 = LengthContribution_3(i+2);
  lc4 = LengthContribution_4(i+3);
  need2 = Need_2nd_Byte(i);
  need3 = Need_3rd_Byte(i+1);
  need4 = Need_4th_Byte(i+2);

  TempLength1 = lc1 + lc2 + lc3 + lc4;
  TempLength2 = lc1 + lc2 + lc3;
  TempLength3 = lc1 + lc2;
  if (need2) {                            Control
    if (need3) {                          Logic
      if (need4) {
        Length = TempLength1;
      } else Length = TempLength2;
    } else Length = TempLength3;
  } else Length = lc1;
  return Length;
}
```

Figure 11: ILD: All the data operations in the CalculateLength function are *speculatively* executed

```
ResetArray(Mark);
NextStartByte = 1;
for (i = 1; i <= n; i++) {
  Results(i) = DataCalculation(i,i+1,i+2,i+3);
  Length(i) = ControlLogic(Results(i));
  len[i] = Length(i);

  if (i == NextStartByte) {
    Mark[i] = 1;
    NextStartByte += len[i];
  }
}
```

Figure 12: ILD: The instruction length calculation function is *inlined* into the main function

```
ResetArray(Mark);
NextStartByte = 1;
i = 1;

Results(i) = DataCalculation(i,i+1,i+2,i+3);
len[i] = ControlLogic(Results(i));
if (i == NextStartByte) {                 1st Iteration
  Mark[i] = 1;
  NextStartByte += len[i];
}

Results(i+1) = DataCalculation(i+1,i+2,i+3,i+4);
len[i+1] = ControlLogic(Results(i+1));
if (i+1 == NextStartByte) {               2nd Iteration
  Mark[i+1] = 1;
  NextStartByte += len[i+1];
}

..... till nth iteration
```

Figure 13: ILD: The loop with index variable i is *unrolled* fully; the £gure only shows two iterations

by a control logic structure that uses this data and assigns the correct result to the output. This control logic maps to multiplexors in hardware. Hereafter, for brevity of presentation, we will refer to these data and control components in the *CalculateLength* function as *DataCalculation* and *ControlLogic* respectively.

In the next step, the *CalculateLength* function can be *inlined* into the main calling function as shown in Figure 12. *Inlining* refers to replacing a call to a function or a subroutine with the body of the function or subroutine [16]. This transformation allows the optimization of the inlined function with the rest of the code. In practice, Spark performs inlining £rst, but speculation within the *CalculateLength* has been shown £rst to simplify explanation.

Next, the loop is fully unrolled as shown by the code in Figure 13. However, the parallelization transformations are still limited due to a dependency that still exists between the operations and the loop index variable i. However, since the loop has been completely unrolled, the constant assignment of $i = 1$ can be propagated throughout the code and the loop index variable i can be eliminated. The resulting code is shown in Figure 14. This exposes further opportunities for early and speculative calculation of the lengths of instructions, as shown in Figure 15(a). In this description, the lengths of the instructions are calculated assuming that a new instruction starts at each byte.

This leads to a design, where all the data for all the bytes is calculated concurrently, followed by a control logic unit, which determines the length of the instructions if they were to start at each byte and £nally, a ripple control logic unit that determines the actual instruction start bytes. The hardware architecture corresponding to this code is shown in Figure 15(b). This is a maximally parallel architecture that can be targeted for implementation in a single cycle.

In this way, Spark achieves the single cycle architecture that is required by starting with the "C" behavioral description of the ILD shown in Figure 10, and producing the register-transfer level VHDL code corresponding to the architecture shown in Figure 15.

```
ResetArray(Mark);
NextStartByte = 1;
Results(1) = DataCalculation(1,2,3,4);
len[1] = ControlLogic(Results(1));
if (1 == NextStartByte) {
  Mark[1] = 1;
  NextStartByte += len[1];
}
Results(2) = DataCalculation(2,3,4,5);
len[2] = ControlLogic(Results(2));
if (2 == NextStartByte) {
  Mark[2] = 1;
  NextStartByte += len[2];
}
...
Results(n)=DataCalculation(n,...,n+3);
len[n] = ControlLogic(Results(n));
if (n == NextStartByte) {
  Mark[n] = 1;
  NextStartByte += len[n];
}
```

Figure 14: ILD: Constant Propagation of loop index variable *i* after the loop has been completely unrolled

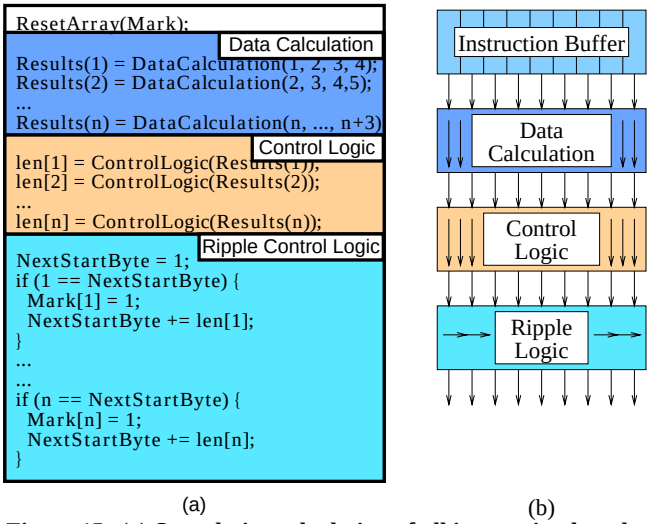


Figure 15: (a) Speculative calculation of all instruction lengths assuming an instruction starts at each byte (b) the £nal ILD architecture produced by the Spark system

Although not demonstrated in this section, Spark chains the operations in the decoder together as described in Section 3.1, to achieve the single cycle architecture. We are unable to compare the synthesis results of our design with any hand-design since the model of the ILD used has several simpli£cations (as discussed earlier).

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have explored a new domain for the application of high-level synthesis; the domain of high performance microprocessor functional blocks. A synthesis methodology has been presented that exploits the liberal resource allocation available to these blocks and achieves the tight cycle bounds by employing a set of aggressive coarse and £ne-grain transformations. This methodology systematically extracts parallelism from a behavioral description by employing parallelizing techniques such as loop unrolling and speculative code motions. Operations are packed into a few cycles by chaining operations, often, across conditional boundaries. This synthesis methodology and the associated transformations have been implemented in the Spark high-level synthesis framework. Finally, we have also demonstrated the effectiveness of this methodology by walking through the transformations applied by Spark to synthesize an instruction length decoder inspired by an advanced microprocessor.

While this case study is instructive in understanding the code

```
ResetArray(Mark);
NextStartByte = 1;
while(1) {
   Mark[NextStartByte] = 1;
   len = CalculateLength(NextStartByte);
   NextStartByte += len;
}
```

Figure 16: ILD: A succinct and natural behavioral description

transformations needed for high performance microprocessor blocks, there are several areas of HLS that require further investigation. For instance, the behavioral description we have used as a starting point for our work (Figure 10) may not be the most simple way to describe the design. A more natural and succinct way to describe the ILD's behavior could be as shown in Figure 16. Similar, short behavioral descriptions can be used to describe several such low latency functional blocks in microprocessors. This leads us to future work in developing a new set of source-level transformations that can transform these sort of descriptions into more easily synthesizable behavioral descriptions.

8. REFERENCES

[1] M. Potkonjak and J. Rabaey. Optimizing resource utlization using tranformations. *IEEE Trans. on CAD*, March 1994.

[2] R. Walker and D. Thomas. Behavioral transformation for algorithmic level IC design. *IEEE Trans. on CAD*, Oct. 1989.

[3] N. Park and A. Parker. Sehwa: A software package for synthesis of pipelines from behavioral speci£cations. *IEEE Trans. on Computer-Aided Design*, March 1988.

[4] P. G. Paulin and J. P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE TCAD*, June 1989.

[5] K. Wakabayashi and H. Tanaka. Global scheduling independent of control dependencies based on condition vectors. In *Design Automation Conference*, 1992.

[6] I. Radivojevic and F. Brewer. A new symbolic technique for control-dependent scheduling. *IEEE TCAD*, January 1996.

[7] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. Incorporating speculative execution into scheduling of control-¤ow intensive behavioral descriptions. *DAC*, 1998.

[8] L.C.V. dos Santos and J.A.G. Jess. A reordering technique for ef£cient code motion. In *Design Automation Conf.*, 1999.

[9] S. Gupta, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Conditional speculation and its effects on performance and area for high-level synthesis. In *ISSS*, 2001.

[10] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Comps.*, July 1981.

[11] S. Novack and A. Nicolau. An ef£cient, global resource-directed approach to exploiting instruction-level parallelism. In *Conference on Parallel Architectures and Compilation Techniques*, 1996.

[12] R.K. Brayton, R. Camposano, G. De Micheli, R.H.J.M. Otten, and J. van Eijndhoven. *The Yorktown Silicon Compiler System*, Silicon Compilation. Addison-Wesley, 1988.

[13] R. L. Gupta, A. Kumar, A. Van Der Werf, and G. N. Busa. Synthesizing a long latency unit within VLIW processor. In *Intl. Conf. on VLSI Design*, 2000.

[14] S. Gupta, N. Savoiu, S. Kim, N.D. Dutt, R.K. Gupta, and A. Nicolau. Speculation techniques for high level synthesis of control intensive designs. In *DAC*, 2001.

[15] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[16] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles and Techniques and Tools*. Addison-Wesley, 1986.

[17] D. D. Gajski, N. D. Dutt, A. C-H. Wu, and S. Y-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.

[18] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to percolation scheduling. In *International Conference on Parallel Processing*, 1993.

[19] Intel Inc., *PentiumPro® Programmer's Reference Manual*.