To appear in International Journal of Parall Programing, Springer.

A Compiler Intermediate Representation for Reconfigurable Fabrics

Zhi Guo[1], Betul Buyukkurt[2],
John Cortes[2], Abhishek Mitra[2] and Walid Najjar[2]

[1] Brocade Communications Systems, San Jose, CA, U.S.A.
zguo@brocade.com,
[2] University of California Riverside, Riverside, CA, U.S.A
{abuyukku, jcortes, amitra, najjar}@cs.ucr.edu

Abstract. Configurable computing relies on the expression of a computation as a circuit. Its main purpose is the hardware based acceleration of programs. Configurable computing has received renewed interest with the recent rapid increase in both size and speed of FPGAs. One of the major obstacles in the way of wider adoption of (re)configurable computing is the lack of high-level tools that support the efficient mapping of programs expressed in high-level languages (HLL) to reconfigurable fabrics. The major difficulty in such a mapping is the translation from a temporal execution model to a spatial execution model. An intermediate representation (IR) is the central structure around which tools such as compilers and synthesis tools are built. In this paper we propose an IR specifically designed for reconfigurable fabrics: CIRRF (Compiler Intermediate Representation for Reconfigurable Fabrics). We describe the design of CIRRF and its initial implementation as part of the ROCCC compiler for translating C code to VHDL. CIRRF is designed to support the creation of a data path and the scheduling of operations on it. It provides support for buffers, look-up tables, predication and pipelining in the data path. One of the important features of CIRRF, and ROCCC, is its support for the import of pre-designed IP cores into the original C source code allowing the user to leverage the huge wealth of existing IP cores while programming the configurable platform using a HLL. Using experiments and examples we show that CIRRF is a solid foundation to generate high-performance hardware.

Keyword: configurable computing, intermediate representation, FPGA, VHDL

1 Introduction

The main problem standing in the way of wider acceptance of reconfigurable computing platforms is their programmability. Currently, application developers must have extensive hardware expertise, in addition to their application area expertise, if they are to develop efficient designs that can fully exploit the potential of FPGA-based configurable platforms. Designing and mapping large applications onto FPGAs is a long and tedious task that involves a large amount of low-level design in a Hardware Description Language (HDL). This poses two problems: Traditional application developers are typically not HDL designers, and HDLs are not well suited to algorithm implementation. Several projects have looked at the translation of traditional programming languages, such as C/C++ or Java, to HDLs for mapping onto FPGAs or other similar fabrics. This is a challenging task. The FPGA is an amorphous mass of logic onto which the compiler must create a data-path and schedule the computation. Such a task requires the harnessing of technologies developed for parallelizing compilers as well as those developed for high-level synthesis. The fundamental differences between the spatial computing model and the temporal, or von Neumann, model are:

- Spatial computing is inherently parallel while temporal computing is sequential.
- Temporal computing relies on two centralized storage locations that are both explicitly addressed by the code: the register file and the memory. In spatial computing, storage is distributed throughout the circuit and is accessed implicitly rather than explicitly. Furthermore, it is the task of the compiler to explicitly create the storage on the FPGA and schedule its accesses.
- Scheduling in temporal computing is driven by control flow, while in spatial computing it is driven by data flow.

The main challenge in translating from a HLL to an HDL is in overcoming these fundamental differences. Optimizing compilers for traditional processors have benefited from several decades of extensive research that has led to efficient tools. Similarly, electronic design automation (EDA) tools have also benefited from several decades of research and development leading to powerful tools that can translate VHDL and Verilog code, and recently SystemC code, into efficient circuits. However, little work has been done to combine these two approaches into one integrated compilation tool where HLL are translated into a high-performance circuit.

At the heart of each compiler or synthesis tool is an intermediate representation (IR) around which the tool is built. In this paper we propose CIRRF (Compiler Intermediate Representation for Reconfigurable Fabrics), an IR designed for the compilation of traditional imperative, high-level languages, targeting reconfigurable devices. CIRRF is intended to be an open halfway-point representation between a high-level language and a specific reconfigurable platform. A front-end tool would translate C/C++, FORTRAN, Java or SystemC to CIRRF. Back tools would map CIRRF to a specific target. Loop and array transformations are dealt with in the front-end tools; target-specific optimizations are implemented in the back tools. CIRRF is designed to be both language and target independent. It is the intermediate representation of the ROCCC compiler (Riverside Optimizing Compiler for Configurable Computing). CIRRF differs from traditional compiler IRs in that it supports concurrency, both explicitly and implicitly, as well as the instantiation of and accesses to on-chip storage structures. It records information about loop types, memory interfacing, instruction predication and pipelining. Special instructions for efficient data-path generation are introduced. ROCCC does not support pointers and memory allocation.

The ROCCC compiler is designed to generate VHDL from C. However, not all application algorithms can be efficiently described by C. Furthermore, industry has invested tremendous

financial and technical efforts on pre-designed intellectual property (IP) cores for FPGA-based platforms that are not only very efficient but have been thoroughly tested and verified. These IP cores come in the form of synthesizable HDL code or even lower level descriptions. They vary drastically with respect to their control and timing protocol specifications which intended to be interfaced to HDL-based designs. Compilers for FPGA-based reconfigurable systems must therefore leverage that huge wealth of IP designs by allowing the user to import these into high-level language (HLL) source codes. To do so would require a wrapper structure that would hide the timing and stateful nature of the IP cores and make each look, to the HLL compiler, as an un-timed side-effect free function call.

We propose a mechanism for the automatic generation of such a wrapper using ROCCC. Users provide the high-level description of a wrapper, which is based on C with timing information. CIRRF records the timing information so that from the compiler's point of view, the wrapper described in C is essentially a timed control flow graph. The compiler's back-end converts this timed CFG into predicated DFG and eventually generates an IP wrapper in VHDL. Notice that a normal C code input to ROCCC does not have any timing implication, while a wrapper in C is a special case, which does have timing information.

The rest of this paper is organized as follows: Section 2 presents CIRRF's architecture and the method we build CIRRF; Section 4 extends CIRRF to support the compiler for IP wrapper generation. Results are given in the subsections of Section 2 and Section 4. Section 5 reviews related work; and Section 6 concludes the paper.

2 CIRRF Intermediate Representation

The major objective of CIRRF is to separate the language specific compiler concerns from those of the target platform. The architecture of FPGA-based platforms can vary widely in the number and types of FPGAs, the number and size of on-board memory modules, the bus

width connecting the FPGAs to the memories, etc. The compiler must therefore generate code that can take advantage of the unique features of each platform. This approach allows for an easy targeting or re-targeting of new or modified platforms.

In this section we describe the overall structure of CIRRF and the two levels currently implemented as part of the ROCCC compiler tool. We present ROCCC's workflow to build the CIRRF IR. The front-end optimizes the user-input code and generates Hi-CIRRF by adding macros into the source (subsection 2.3). Starting from a conventional CFG, the back-end first constructs data flow for do-all loops (Section 2.4), then converts non-do-all nodes into data flow nodes using predication (Section 2.4).

2.1 CIRRF Architecture and Execution Model

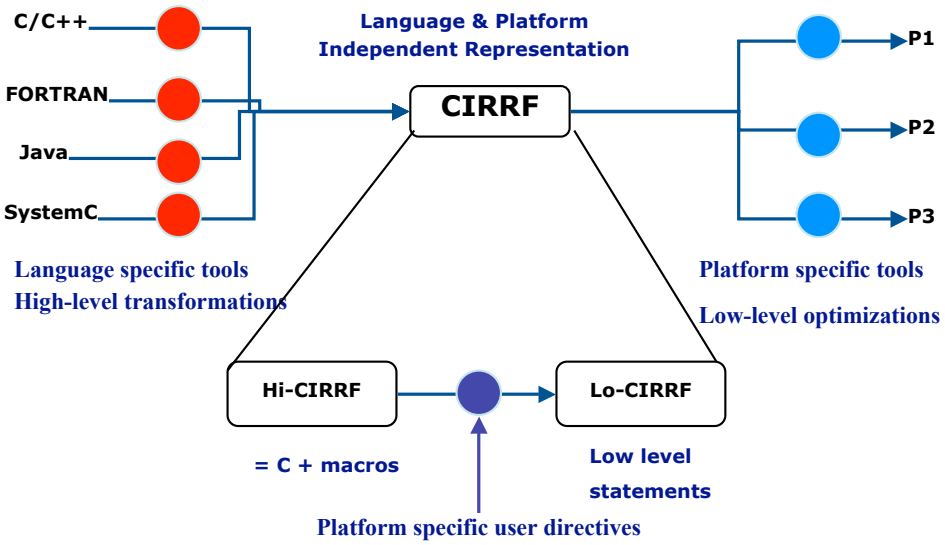


Fig. 1. Overview of CIRRF

ROCCC's intermediate representation, CIRRF, is built using SUIF2 [1] and Machine-SUIF [2]. SUIF2 consists of high-level statements, such as loop statements and if-else statements, while Machine-SUIF consists of virtual machine instructions. CIRRF consists of two distinct but equivalent representations, as shown in Figure 1. The Hi-CIRRF, built using

SUIF2, is essentially C code augmented with macros. The Lo-CIRRF format, extended from Machine-SUIF, is semantically similar to assembly code. The advantage of this approach, which is commonly used in various compiler IRs, is that it allows the user to have multiple levels of entry into the IR. Functional debugging, for example, would be a lot easier at the Hi-CIRRF level. A functional simulation of the generated code would be feasible at the Lo-CIRRF level.

Hi- and Lo-CIRRF The Hi-CIRRF representation is generated after the high-level compiler transformations have been applied. It has a C syntax that has been augmented with macros. The Lo-CIRRF serves as a platform for pipelining, interfacing to memory and generating VHDL. It semantic and syntax are similar to assembly language and rely on statically single-assigned virtual registers.

The macros in Hi-CIRRF are used to:

- Instantiate and access on-chip buffers.
- Implement special operations that eliminate recurrence or enforce a pipeline delay for IP wrapper generation.
- Invoke bitwise and arithmetic operations, such as bit-insert, bit-extract, and minimum of two values.
- Invoke look-up tables and IP cores.

In Lo-CIRRF the code is similar to an assembly code. It is implemented as a control and data flow graph (CDFG) with the following characteristics:

- Virtual statically single-assigned registers.
- Register name indicates type (signed, unsigned) and bit size.
- Predicated instructions.
- Pipelining information for each instruction.

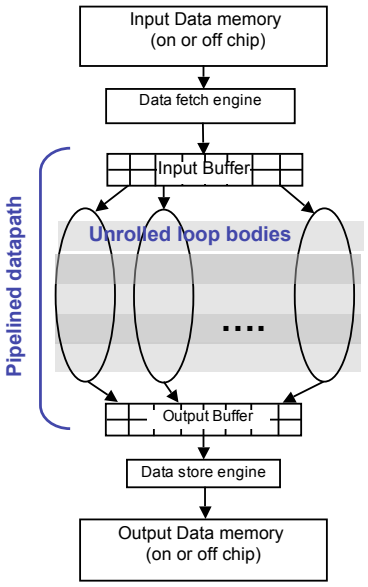


Fig. 2. Execution model of CIRRF

– Loop nodes that specify complete loop information such as loop type (parallel or sequential), nesting level, index and stride.

Execution Model CIRRF is essentially intended to represent loop nests that would be mapped to hardware. Its execution model, shown in Figure 2, is very simple: it consists of an input data memory which can be on or off-chip, a data fetch engine that collects the data into an input buffer. The data is then pushed, every cycle, into the pipelined data path. The same structure is replicated at the output side. Note that, unlike in a von Neumann architecture, the data path does not fetch the data, rather at each iteration, the correct set of data is selected by a controller from the input buffer and pushed into the data path. The CIRRF execution model can therefore be considered a *decoupled* model of execution.

2.2 On-Chip Buffers

One of the distinguishing features of spatial computing representation is that storage needs to be explicitly architectured by the code while in temporal computing it is implicitly available

in the form of registers and memory (cache, main memory etc). Furthermore, the accesses to memory for both data reads and writes have to generated as part of the computation.

Buffers are therefore first class citizens in CIRRF. They play several roles including:

- Interface to memory for both reads and writes.
- Interface between two segments of the data path that operate in a producer consumer relationship.
- Cache data fetched from memory for future reuse to reduce the number of memory accesses.
- Hold run-time constants.

The following is a partial list of buffers that are part of the CIRRF design to date:

- **Memory Interface.** The *mem_read_fifo_buffer* and *mem_write_fifo_buffer* serve as interfaces between the memory, off or on chip, and the data path. They are parameterized in both width and size and are accessed in one cycle.
- **Data Reuse.** The *smartbuffer* [3] is designed to facilitate the reuse of data fetched from off-chip memory. It is particularly well suited for window-based operation as is very common in signal and image processing algorithm where every data sample, or pixels, participates in the computations of several windows. The deallocation and replacement of a data value in the *smartbuffer* is scheduled by the compiler.
- **Run-time Constants.** The ROCCC compiler folds compile-time constants into the computation. However, run-time constants need to be made available to the computation. Two such structures are built into CIRRF.
  - The *constant_buffer* consists of a small set of scalar values that are directly written to registers in the data path.

```
for(i=0;i<N;i=i+1){
  sum = 0;
  for(j=0;j<N;j=j+1)
    sum=sum+A[i][j]*B[j];
  C[i] = sum;
}
```

(a) The original C code of a matrix multiplication.

```
for(i=0;i<N;i=i+4){
   constant_read_buffer(i);
   sum1=0; sum2=0; sum3=0; sum4=0;
   for(j=0;j<N;j=j+2){
     mem_read_fifo_buffer(A, i, j,
       a1,0,0,a2,0,1,a3,1,0,a4,1,1,
       a5,2,0,a6,2,1,a7,3,0,a8,3,1);
       mem_read_fifo_buffer(B,j,b1,0,b2,1);
      sum1=sum1+a1*b1;    sum1=sum1+a2*b2;
      sum2=sum2+a3*b1;    sum2=sum2+a4*b2;
      sum3=sum3+a5*b1;    sum3=sum3+a6*b2;
      sum4=sum4+a7*b1;    sum4=sum4+a8*b2;
   }
one_time_write_buffer(C, i,
      sum1, 0, sum2, 1,  sum3, 2, sum4, 3);
}
```

(b) The C code with the buffer macros. The i loop is unrolled four times and the j loop is unrolled twice.

Fig. 3. An example of the *mem_read_fifo_buffer* and the *one_time_write_buffer*.

- The *one_time_buffer* holds an array of constant values. It operates just like the *mem_write_fifo_buffer* but is read only once. It is used in the cases where the number of constants is too large to be enumerated by the compiler.

Figure 3 shows a matrix multiplication example with the *mem_read_fifo_buffer* and the *one_time_write_buffer*.

– **Producer/Consumer.** Buffers are also generated in CIRRF to implement on-chip producer consumer relationships between segments of the data path: *pc_fifo_buffer*.

2.3 Building Hi-CIRRF

The ROCCC system performs the following loop transformations: invariant code motion; partial and full loop unrolling; loop peeling; loop un-switching; loop tiling; strip-mining; loop fusion; constant propagation of scalars and constant array masks; constant folding; elimination of algebraic identities; copy propagation; dead and unreachable code elimination; code hoisting; code sinking; scalar renaming; and division by constant approximation using shifts and additions. ROCCC generates reduction on scalars that accumulate values

```
for (i=0; i<62; ++i) {
  for(j=0; j<62; ++j) {
     sum = (a[i][j] + a[i][j+1]) + (a[i+1][j] + a[i+1][j+1]);
     if(sum > 170)
           b[i][j] = 255;
     else {
           if(sum < 85)
              b[i][j]  = 0;
           else
              b[i][j]  = 127;
           }
  }
}
```

(a) The original C.

```
for (i=0; i<62; ++i) {
  for(j=0; j<62; ++j) {
     smartbuffer2(a, i, j, x1, 0, 0, x2, 0, 1, x3, 1, 0, x4, 1, 1);
     sum = (x1 + x2) + (x3 + x4);
     if(sum > 170)
        tmp = 255;
     else {
        if(sum < 85)
           tmp = 0;
        else
           tmp = 127;
        }
     memory_write_fifo_buffer2(b, i, j, tmp, 0, 0);
  }
}
```

(b) The intermediate C code.

Fig. 4. A gray scale transformation example in C. (a) The C code sums the gray scale values in a 2x2 window in the input image (array a) and assigns one of three possible values to the pixel in the output image (array b) depending on the value of *sum*. (b) The intermediate C code emitted by front-end. The highlighted segments are created by scalar replacement.

through associative and commutative operations on themselves. It also carries out the following hardware-specific analysis and transformations:

– Scalar replacement. The front-end decouples a do-all loops' array accesses from computation. Figure 4 (a) shows the original C code of a gray scale transformation example. After undergoing scalar replacement, the computation is isolated from memory accesses [Figure 4 (b)] by a smart buffer. The smart buffer will be synthesized on configurable fabrics as the interface with memory. One important characteristic of smart buffers is that they reuse input data between iterations and push one iteration's input data initiatively to the data-path, rather than being accessed by the data-path [3]. The syntax of a two-dimensional smart buffer macro is:

smartbuffer2 (input_array_name, address_index_1, address_index_2,

scalar_1, off set_1_1, offset_1_2, scalar_2, offset_2_1, offset_2_2,);

For example, in the smart buffer macro in Figure 4 (b), the last three parameters *(x4, 1, 1)* stands for: *x4 = a[i+1][j+1]*; Similarly, the syntax of a two-dimensional memory write FIFO buffer macro is:

mem_write_fifo_buffer2(output_array_name, address_index_1, address_index_2,

scalar_1, offset_1_1, offset_1_2, scalar_2, offset_2_1, offset_2_2,);

The memory write FIFO buffer macro in Figure 4 (b) stands for *b[i+0][j+0] = tmp*;

Currently we have the following constraints on buffer macros. An array can only appear in at most one buffer macro. The address indexes of buffers are also the loop counters. The operator between an address index variable and the offset can only be either addition or subtraction.

– Feedback variable detection. The compiler detects scalar recurrence between adjacent iterations. For example, for a loop having a statement $sum = sum + a[i]$, to eliminate the loop-carried dependency, the compiler replaces the sum on the left and the sum on the right with *store2next()* macro and *load_previous()* macro, respectively. These macros guide the backend to instantiate a feedback register to store the current sum for the next iteration.

The output from the front-end is in the forms of both an IR file and an intermediate C with macros. Users could do further optimizations and add pragmas onto the intermediate C.

2.4 Building Lo-CIRRF

The backend first constructs a conventional CFG [Figure 5 (a)]. The compiler finds loops and loop-depth. Loop types are recovered from pragmas provided by the user: Currently, these include one-dimensional do-all loop, two-dimensional perfect nested do-all loop, and non-do-all loop. The pre-process phase of the back-end converts macros in Hi-CIRRF into corresponding instructions. Particularly, buffer macros are converted into buffer instructions and put into separated nodes, as shown in Figure 5 (b) for example. The smart buffer instruction is shown below.

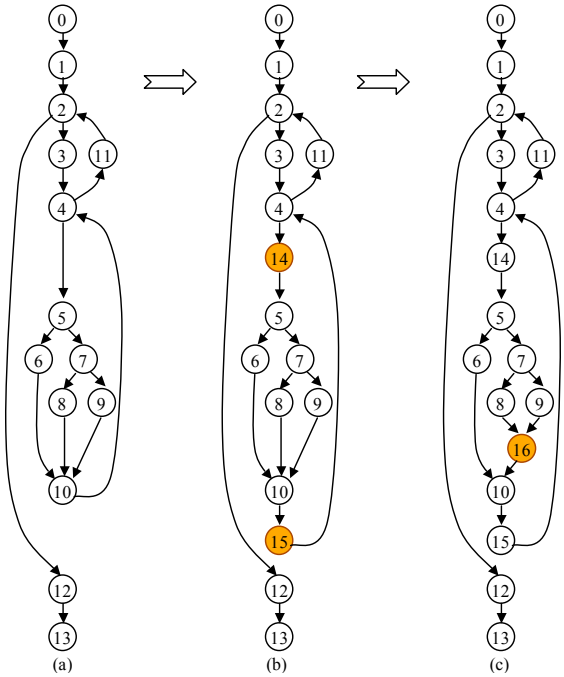


Fig. 5. CFG of the gray scale transformation example. (a) The original CFG. (b) The CFG after pre-processing: highlighted nodes are added. Node 14 is the smart buffer node and node 15 is the memory write FIFO buffer node. (c) The CFG right before performing if-conversion. Node 16 is added to make each joint node have only two predecessor nodes.

smb2 main.x1,main.x2,main.x3,main.x4 ←

main.a,$vr35.s32,$vr75.s32,0,0,0,1,1,0,1,1;

where *main.x1* through *main.x4* are the buffer's output ports, *main.a* is the input memory name, *vr35* and *vr75* are the address index variable (also loop counters). The integers are four pairs of offsets. We categorize basic nodes into two types: parallel *do-all* nodes and sequential *non-do-all* nodes. The compiler's back-end generates data-path using different pipelining and scheduling mechanisms accordingly.

Building Lo-CIRRF for parallel loops For a parallel loop, or do-all node, ROCCC exploits both instruction-level and loop-level parallelisms. The compiler first performs if-conversion to eliminate control flow within the loop body. It then walks through the loop body in a depth-first order and adds extra nodes for joint nodes that have more than two predecessors. Figure 5 (c) shows that a new node, node 16, is added as node 10's predecessor.

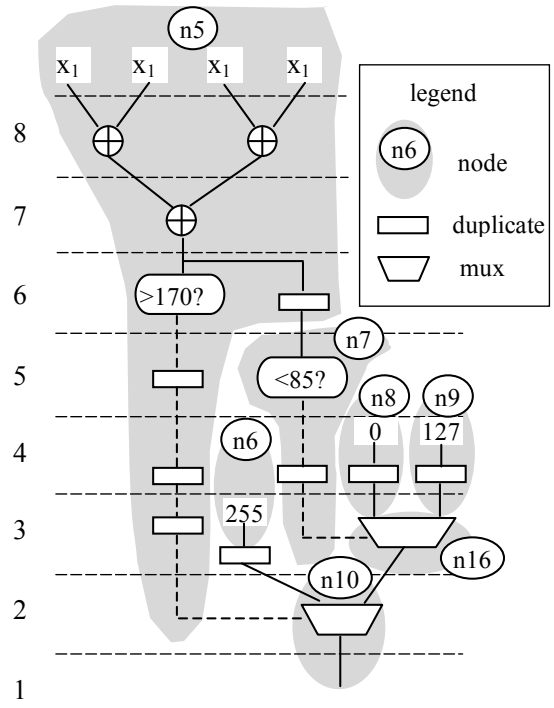


Fig. 6. DFG of the gray scale transformation example. The numbers on the left side are the execution levels. The dashed lines carry speculators, while the solid lines carry data values. The nested if-else statements are converted into a data-flow free of control. Speculators are duplicated along the execution levels. Each execution level is an instantiation of one iteration. Notice that latches can be added between any execution levels.

In order to allow the data-path to implement multiple concurrent loop iterations, the compiler groups the instructions inside a node into execution levels so that each level is an instantiation of one iteration. Statically single-assigned variables are added to duplicate a variable if that variable's definition reaches more than one level of execution lower. Therefore, the definition of each value is at a level strictly higher than that of its use. Multiplexers are added, and predicators are duplicated and propagated along with execution levels, as shown in Figure 6. Notice that each execution level, in the data flow in Figure 6, represents a single iteration at a given execution phase. Then the compiler pipelines the loop body by automatically inserting latches in some execution levels of the data-flow. Instructions belonging to the same execution level are either all latched or all non-latched. Multiple consecutive execution levels may be assigned to the same pipeline stage.

The performance of the generated do-all loop data-paths is described in [4]. The parameters of the data-path, such as the number of pipeline stages, are known at compile time and determined by the user. A parameterized do-all loop controller schedules the fully pipelined data-path's operation.

Building Lo-CIRRF for multi-cycle operation support The ROCCC compiler supports multi-cycle pipelined instructions. This requires synchronization of the dataflow at all levels and prior knowledge of latency. To accomplish this, we introduce an extra step in the compilation process after pipelining all the one-cycle operations. In this extra step, we process the DFG for the bottom up all stages that contain generic functions. If the generic function requires more than one cycle the compiler pushes all lower stages by equivalent amount of remaining stages. If data will not pass through the generic function then pipelined copies (*mov* instructions) are placed to propagate data from before the pipeline stage of the generic function until the end.

With the generic multi-cycle function added to ROCCC, we have added support for IP core math functions, most notably,the support for floating point functions (ADD, SUB, MUL, DIV). Floating point instructions are preprocessed and converted to generic multi-cycle IP cores.

Building Lo-CIRRF for sequential loops In a sequential loop, or non-do-all node, only one iteration is executed at a time. The compiler schedules instructions into different execution levels in a manner similar to a do-all node. But the definition of a variable does not have to be constrained to one level above. Multiple instructions might belong to the same execution level and can be executed simultaneously to exploit instruction level parallelism.

The compiler utilizes predication to schedule the execution of non-do-all nodes' instructions. Each pipeline stage is guarded by a predicator. Lo-CIRRF records predicated instruc-

tions in the following format:

ADD $vr4.s16, $vr3.s16, $vr2.s16, $vr1.u1

vr4 is the destination operand and *vr3* and *vr2* are the source operands. *vr1* is the predicator, which is also a source operand. Predicators are passed inside basic nodes for scheduling purpose. A special instruction, PFW (predicator forward), is created to pass a predicator from the current stage to the next stage, which may be or may not be in the same node:

PFW $vr2.u1, $vr1.u1

vr1 and *vr2* are two predicators. The instructions guarded by *vr2* are one pipeline stage later than the ones guarded by *vr1*. Their types are *u1*, which stands for unsigned one-bit. To convert CFG to DFG, the branch instructions of basic nodes are replaced by Boolean instructions, whose destination operands are evaluated by this basic nodes' successor nodes.

The back-end IR construction phases described so far translate both do-all nodes and non-do-all loop nodes into a DFG. Essentially, the original CFG now is a DFG, in which do-all loop nodes, if any, are connected together by non-do-all nodes. Then the compiler's VHDL generator emits VHDL code for the entire DFG, including buffers.

2.5 Case Study

In this case study, besides reporting the synthesis results of the gray scale transformation example discussed in previous sections, we examine CIRRF on another application, an alternative finite impulse response filter (FIR).

Figure 7 shows the original C code. In even outer iterations flag is one, while in odd outer iterations flag is zero. Therefore the two do-all inner loops (the two highlighted regions) are executed alternately. Each of these two inner loops is a 5-tap FIR. The upper FIR reads array a and writes array b, while the lower FIR reads array c and writes array d. ROCCC's front-end performs scalar replacement, and instantiates a one-dimensional smart buffer macro and

```
void alternative_fir5() {
 int i,j, m;
 int a[256], b[256];
 int c[256], d[256];
 int flag;

 flag = 1;
 for (m = 0; m < 10; m = m + 1) {
   if(flag == 1) {
     for(i = 1; i < 251; i = i + 1)
       b[i] = (3 * a[i-1] + 5 * a[i]) +
            (7 * a[i+1] + 9 * a[i+2]) + 11 * a[i+3];
                  }
    else          {
     for(j = 1; j < 251; j = j + 1)
       d[j] = (3 * c[j-1] + 5 * c[j]) +
          (7 * c[j+1] + 9 * c[j+2]) + 11 * c[j+3];
                  }
   flag = flag ^ 1;
 }
   return;
}
```

Fig. 7. An alternative FIR example in C. The first highlighted segment is a 5-tap FIR reading array a and writing array b, while the second highlighted segment is a 5-tap FIR reading array c and writing array d. $flag$ alternates the execution of these two segments.

a one-dimensional memory write FIFO buffer macro into each inner do-all loop, as described in Section 4.

The back-end first builds a CFG from the input. For this example, there are two do-all loops nested inside the outer non-do-all loop. The back-end scans the whole CFG and transforms the two inner 5-tap FIRs into data flow. Each FIR's loop body is aggressively pipelined, and the resulting data-path has a throughput of one iteration per clock cycle. These two FIRs are controlled by two do-all loop controllers, as shown in Figure 8 (a). The back-end then converts the outer loop and the rest of the CFG into a DFG by predicating all the non-do-all nodes. We list the instructions of these nodes in Figure 8 (b). Node 2 and node 10 are the head and tail nodes of the outer loop, respectively. The first instruction of node 2, the ior instructions, produces the predicator ($vr1321$) for the two instructions below it (pfw and sle) by examining a valid output predicator from either node 1 (not shown), the first

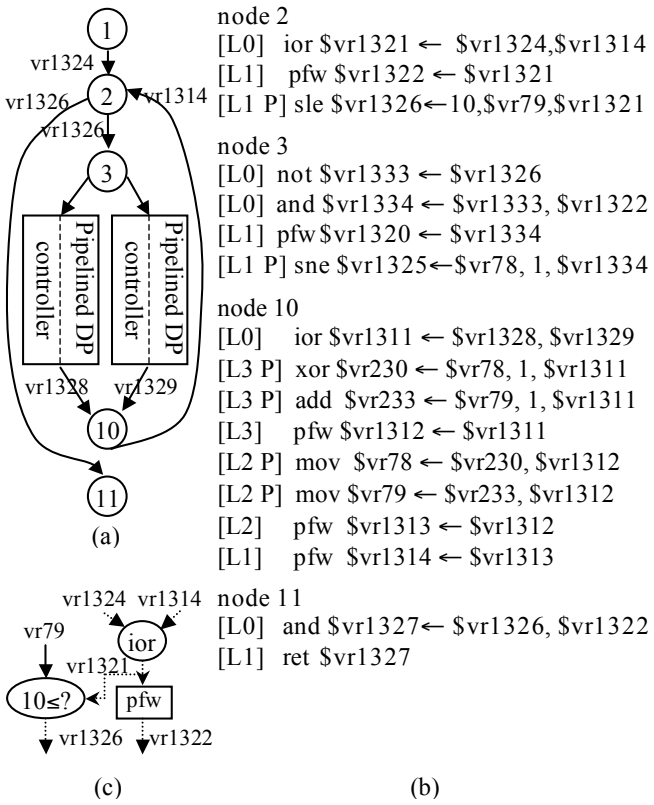


Fig. 8. The DFG and IR of alternative FIR. (a) is the DFG. Node 3's successors are two do-all loops controlled by their loop controllers. For simplicity, we are not presenting the details of these two loops. Operands' data types are not shown. Node 1 through 3 and node 10 through 12 are non-do-all nodes, scheduled by predicators that are carried by the edges. (b) The DFG IR of node 2, 3, 10 and 11. The L field is the pipeline stage (latch level) of the instruction within its node. An instruction with a P field is predicated by its last source operand, which is the predicator. (c) Shows node 2's circuit, in which a solid lines carry data and a dashed lines carry predicators.

active node; or node 11, the loop tail. Figure 8 (c) depicts node 2's circuit in detail. Guarded by $vr1321$, the *sle* instruction asserts its destination operand when the outer loop is done, or de-asserts its destination operand when needing to execute a new outer loop iteration. Node 2's pfw instruction forwards a valid $vr1321$ to the two successor nodes, node 3 and node 11, for their predicator evaluation. Node 3 enables one of the two FIRs by either asserting or de-asserting $vr1325$, depending the value of $flag$ ($vr78$). Node 10 is activated by the done signal from one of the FIRs' loop controller and updates the value of $flag$ ($vr230$) and the loop counter m ($vr233$). Node 11 indicates the completion of the whole procedure.

Table I. Synthesis results of case study examples

	gray scale	FIR
datapath bitsize	16	8
memory bus bitsize	16	8
slices	318	531
clock (MHz)	59.7	100
iterations per cycle	0.5	1

Table I shows the synthesis results of the gray scale transformation example discussed in previous sections and the alternative FIR. The target FPGA is the Xilinx xc2v8000-5 with 46592 slices. The generated VHDL is synthesized and placed-and-routed using the Xilinx tool chain. The second and the third rows are the data-path's bit-size and BRAM bus's bit-size. *slices* and *clock* are collected from place-and-route reports. The last row is the number of do-all loop iterations executed per clock cycle. For gray scale transformation, the do-all loop's loop body has nested if-else statements, as shown in Figure 4. After undergoing if-conversion, in the IR right before VHDL emission [Figure 6], the control flow is eliminated and the resulting data flow is capable of executing one iteration each clock cycle. Notice we configure the BRAM's data bus (the third row) to have the same bit-size as that of the data elements (pixels), and each iteration needs four (2x2) pixels. Though the smart buffer reuses one column of the pixels loaded in previous iteration, it still needs two cycles to load the remaining two new pixels. This explains why, for the gray scale transformation example, the number of iterations per cycle is 0.5. For the alternative FIR, the VHDL generator generates one smart buffer and one memory write FIFO buffer for each of the two do-all loops according to the buffer representations in the IR. When either one of the two do-all loops is active, the corresponding smart buffer exports one window of data (five elements) to the data-path every clock cycle, and therefore the circuit executes one iteration per cycle. *slices* consists of the hardware of two do-all loops (the data-path, the buffers and the controller for each FIR) and the hardware of the non-do-all nodes, as shown in Figure 8(a).

3 On-chip Buffers - Implementation and Evaluation

The on-chip buffer macro is one of the features of CIRRF. In [3] and [5] we introduced the smart buffer approach to input data reuse. The smart buffer is ideally suited for operations that involve a heavy reuse of fetched data, windowing operations on images being an obvious example. In order to support more memory access patterns this mechanism is extended by adding more types of FIFO buffers as described in Section 2.2. In this section we describe the implementation of various on-chip FIFO buffer structures and report on the evaluation of their area and speed.

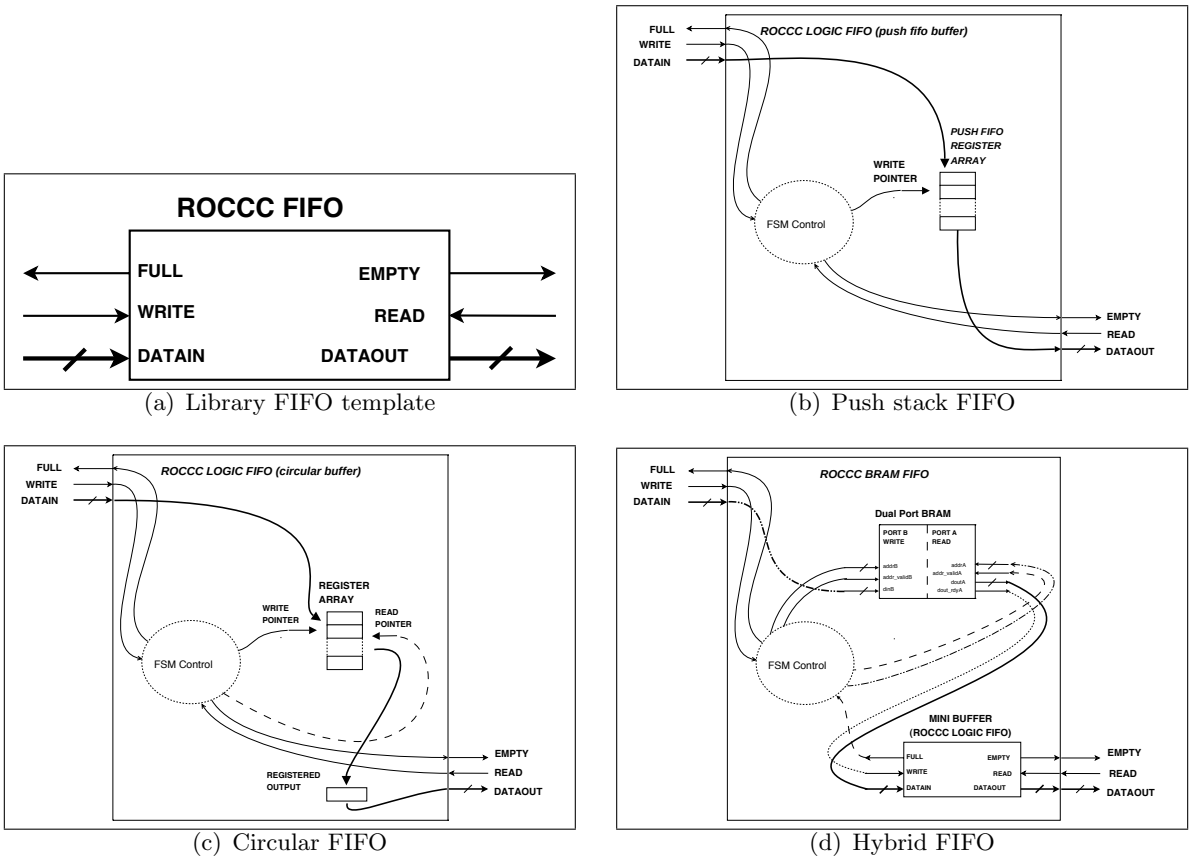
3.1 Implementation of FIFO Buffers

ROCCC supports both pre-designed VHDL library FIFO buffers and compiler generated buffers. The smart buffer falls in the later category. This section describes the three VHDL library FIFO buffers and an improvement to the smart buffer design.

The three FIFO buffers in the ROCCC VHDL library are push stack FIFO buffer, circular FIFO buffer and hybrid FIFO buffer. The first two are built with pure logic, while the hybrid FIFO buffer uses BRAM for storage. They are instantiated by the compiler when generating VHDL code. Each of these FIFOs have *registered outputs* - the data output and data output assertion are synchronized together instead of having to wait one cycle after assertion for the data output to be valid.

A template that all the library fifos follow is show in Figure 9(a): each fifo has datain, dataout, and handshaking signal ports. Figures 9(b), 9(c), 9(d) are different implementations based on that template.

– **Push stack FIFO buffer.** Shown in Figure 9(b). Items are pushed onto the top of the stack buffer (like a real FIFO system) and popped out from the bottom.



(a) Library FIFO template (b) Push stack FIFO

(c) Circular FIFO (d) Hybrid FIFO

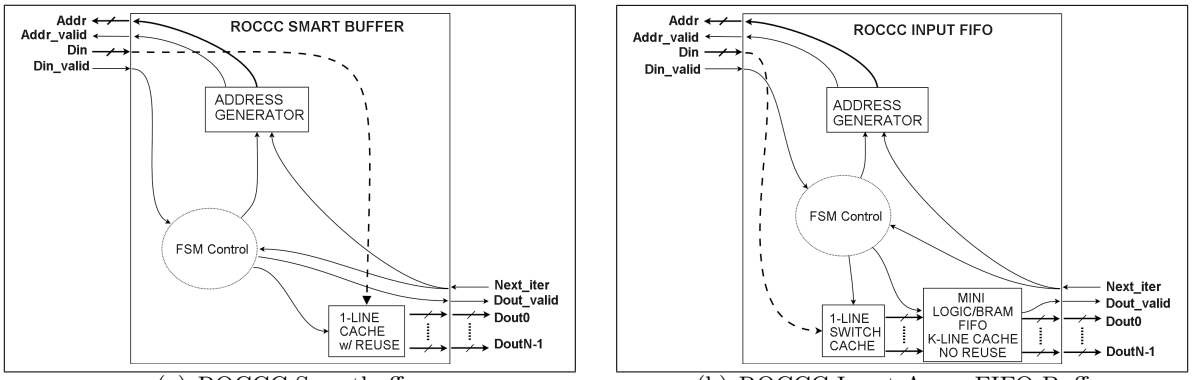
Fig. 9. Library FIFO BUFFERS

– **Circular FIFO buffer.**Shown in Figure 9(c). It uses a single buffer and two array pointers, *write_prt* and *read_ptr*, for keeping track of where the data has been written and where it can be read from. Due to the implementation, if a size of N elements is needed to be stored in the FIFO, N+1 elements must be allocated for the buffer.

– **Hybrid FIFO buffer.** Shown in Figure 9(d), its component's layout is actually similar to that of the logic FIFO - circular. Read and Write pointers are used, but the buffer is implemented with BRAM instead of a set of register buffers. To improve the latency, a mini buffer (a logic FIFO) is used to cache a few elements from the top of the FIFO. BRAM FIFOs do not spend FPGA fabrics on storage and are area efficient.

The implementation of the smart buffer (Figure 10(a)) is designed so that if no reuse is needed, the buffer generated acts like a FIFO. In such a case, the compiler builds a small

cache within the smart buffer, now used as a FIFO, so it can work in the background during pipeline stalls, memory fetch stalls, or buffer synchronization stalls. This structure is called the ROCCC Input Array FIFO Buffer (Figure 10(b)).



(a) ROCCC Smartbuffer (b) ROCCC Input Array FIFO Buffer

Fig. 10. Smartbuffer and Input Array FIFO Buffer

A couple of assumptions that are required before designing this input array FIFO buffer:

- The input array FIFO buffer must **not** be used in cases where re-use is needed. If this is done then extra logic will be used for storing the same element and thus wasting resources.
- The desired latency and acceptable area cost of the input FIFO. If an area efficient system is desirable then the use of the smartbuffer with no-reuse (which downgrades to a FIFO) would be best to use.

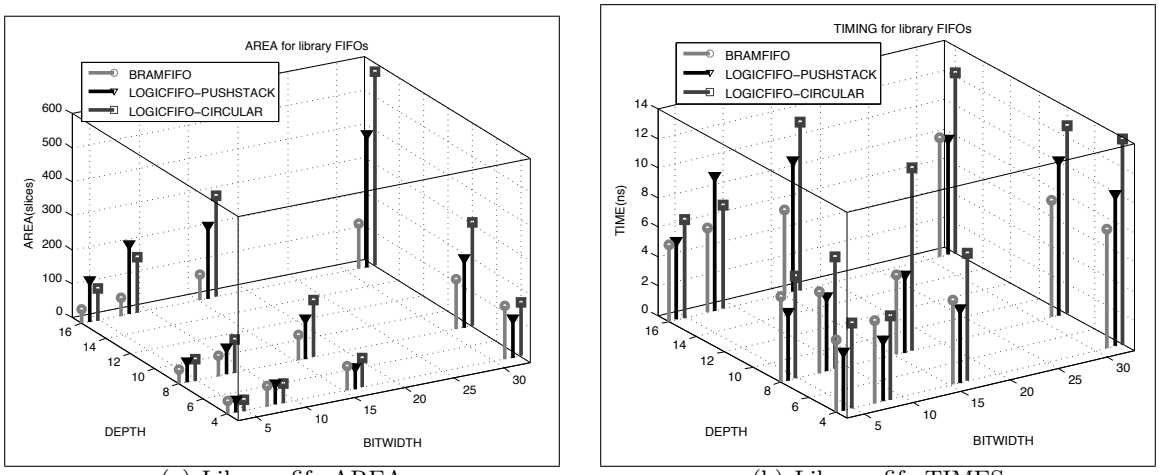
If the system can afford extra area cost to allow decrease of execution time, and no-reuse is needed, then the input array FIFO should be generated.

3.2 Evaluation

In this section we evaluate the performance of the various FIFO implementations and their tradeoffs.

ROCCC Lib FIFO Buffer Experimental Results The library FIFOs area and timing are as followings in Figure 11(a) and Figure 11(b), respectively. The figures suggest the

BRAM FIFO is one of the most efficient in area and with a decent clock speed. The shortcoming with this type of FIFO is the latency is about 4 cycles (from the time when data is inputed to when it is outputted). For the logic FIFOs the latency is about 1 cycle (which is a good as it can get), but as can be seen in the figures the area usage is more but the clock speed suffers. If we compare the two logic FIFOs we get that the push stack implementation is more area efficient than the circular implementation. This is most likely due to the routing of the outputs of each of the registers in the circular buffer to the output as opposed to routing only the last register to the output in the push stack implementation.

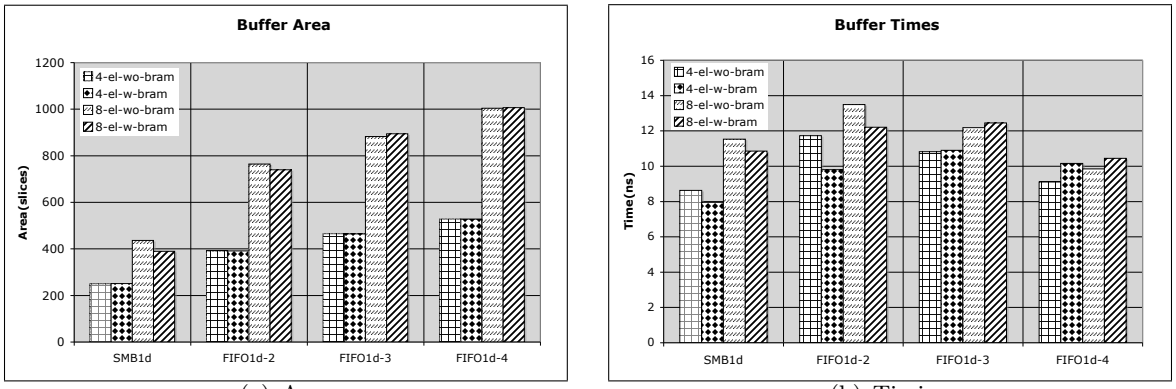


(a) Library fifo AREA (b) Library fifo TIMES

Fig. 11. 3D plots of library fifos

Input Array FIFO Buffer Experimental Results Figure 12(a) and Figure 12(b) show how the area or timing is affected by the type and depth of the buffer. Each buffer holds 4 or 8 elements (32 bits each) per line of cache. There is data for the SMB1D (smartbuffer 1-dimension), which acts as a fifo for 1 cache line. The rest are the input array FIFO buffer implementations of different depths (depths 2, 3, and 4), which are listed horizontally on the table. So the table lists buffers for sizes 1 to 4, but the size 1 implementation is the smartbuffer, and the rest are the FIFOs so that one can compare the relative growth in area

and time. On top of all the implementation we can include the buffers with some BRAM or not, so that the synthesis tools knows to use the internal board components or port map all I/O pins of the buffer to the I/O pines of the board. As can be seen, offers some area discrepancy.



(a) Area (b) Timing

Fig. 12. Input Array Buffers - Area and Timing plots

The plots show in Figure 12 exhibit a linear growth in area for different cache sizes of 2 to 4 elements. If the compilation process requires optimal area then a cache size of 1 is considered and the smartbuffer is generated. The timing on the other hand exibits a decrease in clock rate, faster frequency, from cache level 2 to 4 (all the input array fifos), due to the way the cache is implemented (in this case generated with a cache built from the library ROCCC logic fifo - stack). All things considered, if area can be sacrificed the compilation process may consider to reduce stalls by generating the input array FIFO buffer.

4 Interface Synthesis

Pre-designed IP core represent a huge intellectual and financial wealth that high-level compilation tools targeting FPGAs should not ignore. The ROCCC compiler does support the import of pre-designed IP cores into C source codes. Most often, the interface to these cores is timed and requires several cycles of synchronization and handshaking. These characteris-

tics do not fit well with the C semantics. In this section we describe our approach which is to generate a wrapper that would make the IP core look and behave like a C function. The workflow is shown in Figure 13. Taking the high-level wrapper abstractions as input, ROCCC generates synthesizable wrappers in VHDL separately and these wrappers are instantiated as components in the outer circuit. Notice that an IP core is not necessarily a mandatory element of the main untimed application C code. The grayed out part on the left only exists when there is an IP instantiation in the source code.

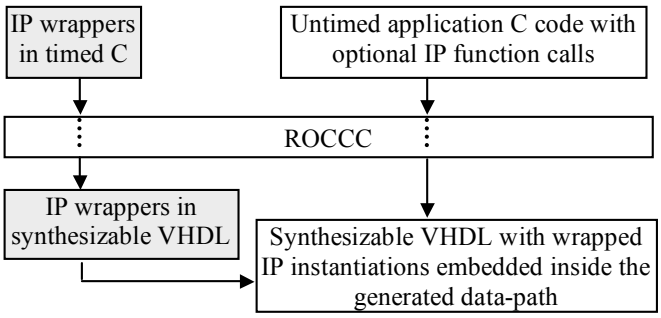


Fig. 13. ROCCC's workflow when IP function call present. Notice that an IP core is not necessarily a mandatory element of the main untimed application C code. The grayed out part on the left only exists when there is an IP instantiation in the source code.

We start with an example of a 16 samples complex FFT, in Section 4.1, taken from the Xilinx website that we use to demonstrate our approach in the remainder of this section.

4.1 An IP Core Example

The grayed out part of Figure 14 is a 16-point complex Fast Fourier Transform core (FFT16). Pins *di_r* and *di_i* are respectively the real and imaginary serial data input, *xk_r* and *xk_i* are the output. *Ce*, clock enable, must be asserted only when the core is active. *Start* must be asserted two clock cycles ahead of the first pair of input data. *Done* is asserted when the first pair of output data is ready. *Fwd_inv* selects between forward or inverse FFT. *Scale_mode* selects from two scale-coefficients: 1/16 or 1/32. The $ovflo$ pin indicates the

core has generated an arithmetic overflow. *mode_ce* input indicates when to sample fwd_inv and *scale_mode*.

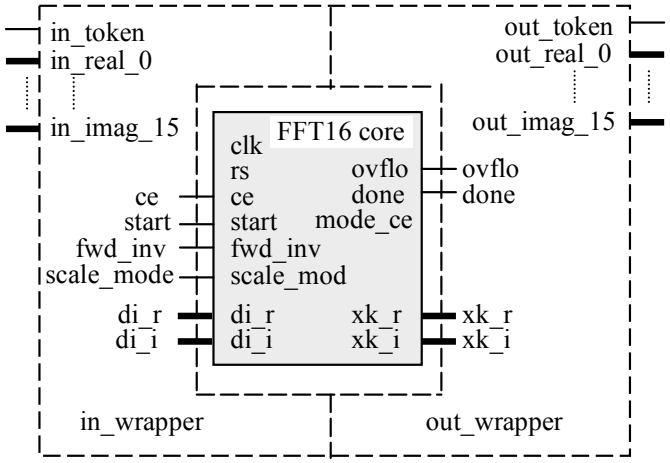


Fig. 14. The grayed out square is the FFT16 IP core. A wrappers interface consists of one or more data ports and one token (either input or output token).

4.2 High-level Wrapper Abstraction

An IP core requires a wrapper for both its input and output interfaces. In some cores these two interfaces have common signals that handle synchronization and handshaking. In our implementation this role is covered by the outer circuit within which the c ore is embedded.

Figure 15 lists the code for input wrapper of FFT16's in C. We use pointer type to distinguish output signals from input signals in the function declaration. The input set, which communicates with the outside, is composed of one token and several data variables. The output wrapper, not shown here, has the same structure.Thus both the input and output interfaces have the same structure as shown in Figure 16.

By its very nature, an interface to an embedded core must support timed activity. In Figure 15, the function call $wait\_cycles\_for(n)$ indicates the statements following it must be executed n cycles later. Any statements between two adjacent $wait\_cycles\_for$ calls must be executed in one clock cycle. For example, FFT16's timing protocol requires that the *start*