ABSTRACT
Multiple clock cycles are needed to cross the global interconnects for multi-gigahertz designs in nanometer technologies. For synchronous design, this requires the consideration of multi-cycle on-chip communication at the high level. In this paper, we present a new architectural synthesis system integrated with global placement, named MCAS (Multi-Cycle Architectural Synthesis), on top of the recently-proposed Regular Distributed Register (RDR) micro-architecture [3]. The RDR architecture provides a regular synthesis platform for supporting multi-cycle communication. Novel architectural synthesis algorithms that integrate high-level synthesis with global placement have been developed in MCAS, including scheduling-driven placement and distributed controller generation, etc. Experimental results show that our methodology can achieve a clock period improvement of 31% and a total latency improvement of 24% on average compared to the conventional architectural synthesis flow.

1. INTRODUCTION
There are two important inflection points in the development of nanometer process technologies. The first point is when the average interconnect delay exceeds the gate delay, which happened during mid 1990s and led to the so-called timing closure problem. The second point will occur when single-cycle full chip synchronization is no longer possible. For multi-gigahertz designs in nanometer technologies, multiple clock cycles are needed to cross the chip as shown in [2]. This is not supported in the current design tools and methodologies, as most of these implicitly assume that full chip synchronization in a single clock cycle is feasible.

Several design methodologies can be adopted to address the multi-cycle communication problem, including asynchronous design, global asynchronous locally synchronous (GALS) design, and synchronous design with multi-cycle communication. This paper focuses on the synchronous design, which is still by far the most popular design methodology. It is well understood and supported by the mature CAD toolset.

There are some efforts to address the problem of multi-cycle on-chip communication for synchronous designs. Most of them are at the gate level to perform retiming with placement or floorplanning [4][1][17][6] to alleviate the performance degradation caused by long interconnects. Although the benefit of applying these methods can be significant, exploring multi-cycle communication during logic synthesis has a big limitation, as the minimum clock period that can be achieved by logic optimization is bounded by the maximum delay-to-register (DR) ratio of the loops in the circuit [14][5]. Figure 1 illustrates this problem by showing a piece of circuitry with 4 logic cells and 2 registers in a loop. Assuming that cell delay is 1ns and interconnect delay is 4ns, the DR ratio of this loop can be calculated by:

$$\text{DR} = \frac{(\text{Delay}_{\text{cell}} + \text{Delay}_{\text{int}})}{\#\text{Registers}} = \frac{(4 \times 4 + 4)}{2} = 10\text{ns}$$

Therefore, the best possible clock period is 10ns under any retiming solution. To further reduce the delay to below 10ns, we need to consider multi-cycle communication during the architectural (or behavioral) synthesis stage. A major progress in this direction is the Regular Distributed Register (RDR) micro-architecture proposed in [3]. The RDR micro-architecture provides a regular synthesis platform for supporting multi-cycle communication. We will review the RDR micro-architecture in Section 2.

In this paper, we present a new architectural synthesis system integrated with global placement, named MCAS (Multi-cycle Communication Architectural Synthesis), on top of the RDR architecture. The main contributions of this work are: (1) We proposed an efficient method called scheduling-driven placement, to tightly integrate scheduling with the global placement; (2) We developed practical solutions, such as the distributed controllers generation and the use of Static Single Assignment (SSA) [7], to handle the control flow for the RDR architecture.

The remainder of the paper is organized as follows. Sections 2 and 3 review the RDR micro-architecture and the overall design flow of MCAS system. Section 4 describes the algorithms for integrating scheduling with global placement. Section 5 discusses how to extend the algorithms to handle the control flow. Experimental results are shown in Section 6, followed by the conclusions and future work in Section 7.
2. REVIEW OF RDR ARCHITECTURE
The RDR architecture divides the entire chip into an array of islands. The registers are distributed to each island, and the size of each island is chosen such that all local computation and communication within an island can be performed in a single clock cycle.

Each island consists of the following components:

1) Local Computational Cluster (LCC): The functional elements in an LCC provide the computational power of the circuit. They can be simple logic such as NAND gates, multiplexers (MUX), or datapath elements such as multipliers, dividers, and ALUs, etc.

2) Register file: The dedicated local storages reside in the register file, which can be partitioned into k banks (where k is the maximum number of cycles needed to communicate across the chip) such that registers in bank i will hold the results for i cycles for communicating with another island that is i cycles away.

3) Finite State Machine (FSM): Each island contains a local controller (i.e., an FSM) to control the behaviors of the computational elements and registers.

The RDR architecture is similar to the recently-proposed distributed-register architecture (DRA) [9][10] in the sense that both architectures distribute registers close to the local computational units, allow multi-cycle communication, and enable concurrent computation and communication. However, unlike DRA, the RDR architecture is highly regular, which greatly facilitates the interconnect delay estimation. Interconnect delay can be easily estimated from the island indices of the source and the destination. Moreover, the RDR architecture has the advantage in that by varying the size of the basic island, one can easily target different clock periods and systematically explore the cycle time vs. latency tradeoff. ¹

More details of the RDR architecture are available in [3].

Given the target clock period and the RDR architecture specification (including the island structure, functional unit library and delay table), MCAS starts with a synthesizable C description. It first generates the control data flow graph (CDFG) from the behavioral descriptions through the intermediate representations of the SUIF compiler infrastructure [24] and Machine-SUIF [18]. At the front-end, MCAS performs resource allocation, followed by an initial functional unit binding. It uses the time-constrained force-directed scheduling (FDS) algorithm [15] to obtain the resource allocation. After the FDS-based resource allocation, it performs a similar algorithm in [10] to bind operation nodes to functional units for minimizing the number of potential data transfers among the components with the same types. Then an interconnected component graph (ICG), which is first defined in [10], is derived from the bound CDFG. An ICG consists of a set of components to which operation nodes are bound. They are interconnected by a set of connections which denote data transfers between components.

At the core, MCAS performs the scheduling-driven placement, which takes the ICG as input, places the components in the island structure of the RDR architecture, and returns the island index of each component. After the scheduling-driven placement, both the CDFG schedule and the layout information are produced. To further minimize the schedule latency, it performs the placement-driven rescheduling-and-rebinding. The algorithm is based on the force-directed list-scheduling framework, and integrated with simultaneous rebinding.

At the backend, MCAS performs register and port binding followed by datapath and distributed controllers generation. The final outputs of MCAS include the RT-level VHDL files for logic synthesis, floorplan and multi-cycle path constraints for place-and-rout. If the final design cannot meet the performance requirement, we can adjust the clock period and the basic island size by binary search and redo the synthesis.

3. OVERVIEW OF MCAS SYSTEM
Our architectural synthesis system for multi-cycle communication (MCAS) is built on top of the RDR architecture. The overall design flow is shown in Figure 3.

3.1. Overview of MCAS System
Our MCAS system starts with the RDR architecture specification, including the island structure, functional unit library and delay table. It then generates the control data flow graph (CDFG) and performs resource allocation. The CDFG is then used to define the interconnected component graph (ICG), which is used for scheduling and placement.

3.2. Scheduling and Placement
After resource allocation, MCAS performs scheduling-driven placement. The placement algorithm is based on force-directed scheduling (FDS) and minimizes schedule latency by considering both the CDFG schedule and the layout information.

3.3. Register and Port Binding
MCAS performs register and port binding after the scheduling-driven placement. The binding process takes into account the data transfers and ensures data consistency across the islands.

3.4. Datapath and FSM Generation
MCAS generates the datapath and finite state machines (FSMs) for each island. The generated code is then converted to RT-level VHDL files.

4. INTEGRATION OF SCHEDULING WITH GLOBAL PLACEMENT
There are two approaches for integrating high-level synthesis with placement (or floorplanning) to achieve timing closure at the high level. One is sequential and the other is simultaneous. In the sequential approach [10][20], placement and scheduling (or binding) are performed separately. [10] performed operation...
binding, placement and post-layout scheduling sequentially and
the placement was driven by the inter-clock slack time obtained
by an initial scheduling. [20] formulated the placement problem
into a linear programming (LP) model to eliminate the potential
slack time violation, and resource sharing is performed after
placement. As pointed out in [3], the sequential approach may
produce suboptimal solutions. In the simultaneous approach
[21][8][16], placement and high-level synthesis are coupled
tightly for performance optimization. The simultaneous approach
has the advantage of efficiently generating better results.
Unfortunately, the existing simultaneous approaches only
minimize the interconnect delay within a single clock cycle thus
reducing the clock period, and cannot consider multi-cycle
communication to overcome long global interconnect delays.

In this section, we present an efficient solution, called scheduling-
driven placement, to perform scheduling simultaneously with
global placement for multi-cycle communication. It takes the full
advantage of the regularity of the RDR architecture to determine
the data path schedule and the island assignment of each
functional unit.

4.1 Scheduling-Driven Placement

The main idea of scheduling-driven placement is to use
scheduling to identify the critical connections in ICG and assign a
high weight to them. A simulated annealing-based coarse
placement engine minimizes the weighted wirelength to shorten
the critical connections and thus potentially reducing the latency.

4.1.1 Problem Formulation

The placement problem for RDR architecture is formulated as
follows. The inputs include (1) island structure, denoted by \( I_x \times I_y \),
which defines the two dimensions of the island-based array; (2)
the bound CDFG and its derivative, ICG. Given the above inputs,
the placer places the ICG components in the given island structure
for minimizing the number of clock cycles needed to schedule the
bound CDFG.

4.1.2 Scheduling-Based Timing Analysis

Since ICG is not acyclic in general, conventional static timing
analysis does not work. One possible solution is to apply ASAP
(As Soon As Possible) scheduling and ALAP (As Late As
Possible) scheduling to determine the critical edges in ICG. We
compute the criticality of each edge \( e = (s, t) \) by \( \text{edge}_\text{slack}(s, t) = \text{ALAP}(s) - \text{ASAP}(t) \) where \( \text{ASAP}(s) \) is the ASAP schedule of
operation node \( s \) and \( \text{ALAP}(t) \) is the ALAP schedule of operation
node \( t \). The edges with zero \( \text{edge}_\text{slack} \) are critical. Although this
method is simple and intuitive, the edges of ICG can be overly
constrained. Since both ASAP and ALAP ignore any resource
constraints, the delays of their schedules may be far shorter than
the real ones.

In order to get a more accurate delay estimation of ICG, we
perform a list-scheduling on the bound CDFG. Specifically, our
scheduling algorithm picks the ready operation node with the
largest critical path length, i.e., the longest path from the node to
the primary outputs (POs), and schedules it to the first feasible
control step.

To compute the criticality of each edge in the scheduled CDFG,
we first define the arrival time and required time of each
operation node.

The arrival time of node \( t \), denoted as \( \text{ARR}(t) \), generally refers to
the data ready time when all inputs of \( t \) are available. In this case,
however, even though all inputs are available, \( t \) can be deferred
due to the resource conflicts. Since our scheduling algorithm
always schedules the node \( t \) in the earliest feasible control-step,
we define the arrival time as \( \text{ARR}(t) = \text{cstep}(t) \). The longest path
delay of the scheduled CDFG is \( T = \max_{t \in \text{PO}} \{ \text{ARR}(t) \} \).

The required time of node \( t \), denoted as \( \text{REQ}(t) \), is the time when
node \( t \) must be scheduled. Otherwise, the timing constraint could
not be made. We first set the required time of all primary outputs
to be \( T \). Required time is then propagated backwards in CDFG
with the following equation:

\[
\text{REQ}(s) = \begin{cases} T, & s \in \text{PO} \\ \min \{ \text{REQ}(t) - \text{edge}_\text{delay}(s, t) - \text{logic}_\text{delay}(s) \}, & \text{otherwise} \end{cases}
\]

Required time analysis also accounts for the resource conflict
during the computation of \( \text{edge}_\text{delay}(s, t) \). It is defined as
\( \text{edge}_\text{delay}(s, t) = \text{int}_\text{delay}(s, t) + \text{conflict}_\text{delay}(t) \) where \( \text{int}_\text{delay} \)
denotes the interconnect delay (in terms of clock cycle number)
between the FUs to which operations \( s \) and \( t \) are bound, and
\( \text{conflict}_\text{delay} \) denotes the number of cycles that \( t \) is delayed due
to the resource conflicts. The introduction of \( \text{conflict}_\text{delay} \) brings
the consideration of resource conflicts into the required time
analysis.

Then we compute the slack of an edge \( (s, t) \) in CDFG by:

\[
\text{EDGE}_\text{SLACK}(s, t) = \text{REQ}(t) - \text{ARR}(s) - \text{int}_\text{delay}(s, t) - \text{logic}_\text{delay}(s)
\]

Given a scheduled and bound CDFG \( \text{CDFG} = (V, E) \), we have the
following conclusion:

DEFINITION. An edge \( e \) is defined to be critical if the
increment on its \( \text{int}_\text{delay} \) by \( \sigma \) will increase the total latency by
\( \sigma \) as well, assuming that the same schedule (i.e., node ordering) is
followed.

THEOREM. \( \forall e \in E, e \) is critical if and only if
\( \text{EDGE}_\text{SLACK}(e) = 0 \), where \( \text{EDGE}_\text{SLACK}(e) \) is computed by
the above timing analysis technique.

We will omit the proof of this theorem due to the page limit. Note
that the above method works correctly with respect to a particular
schedule instead of all possible schedules.

Finally, the criticality of an edge \( (s, t) \) in CDFG is defined as:

\[
\text{EDGE}_\text{CRIT}(s, t) = 1 - \frac{\text{SLACK}(s, t)}{\max_{\forall e \in E} \text{SLACK}(e)}
\]

4.1.3 Net Weighting

At each temperature in the SA process, the interconnect delays
are extracted from the current layout and back-annotated to the
dges in the bound CDFG. Then we determine the criticality of
each edge in the ICG by the scheduling-based timing analysis.
The criticalities of the edges in CDFG are obtained and
transferred to the weights on the corresponding edges in ICG. We
use the method proposed in [12] to compute the weight, i.e.,

\[
\text{weight}(e) = \text{criticality}(e')^{\text{exponent}},
\]
where e is an edge in ICG, e’ is the corresponding edge in CDFG and exponent is a user-defined parameter to heavily weight connections that are critical.

Figure 4. (a) Schedule generated by list-scheduling (b) Current layout at temperature i

Figure 4 illustrates this process using a CDFG with 10 operation nodes bound to 2 ALUs (ALU1 and ALU2) and 2 multipliers (MUL1 and MUL2). We assume uniform node delay that is equal to the target clock period. The nodes in the same pattern are bound to the same functional unit. The double lines, solid lines and the dotted lines represent the two-cycle inter-island data transfer, the one-cycle inter-island data transfer and the intra-island data transfer. Figure 4 (b) shows the current layout at temperature i, and Figure 4 (a) shows the schedule generated by the list-scheduling. After the scheduling-based timing analysis, we can identify that the edges (3,8), (4,7), (7,9), and (8,10) have the zero EDGE_SLACK, namely they are the most critical ones.

Then we assign high weight to the connection between ALU1 and MUL1, and the one between ALU2 and MUL2. At the next temperature, the weighted wirelength minimization engine will try to reduce the delay of these connections.

Figure 5. (a) Schedule generated by list-scheduling (b) Current layout at temperature i+1

Figure 5 shows the layout and schedule at temperature i+1. By swapping the locations of ALU1 and ALU2 to shorten the critical connection between ALU1 and MUL1, and the one between ALU2 and MUL2 as well. We can then reduce the latency by one cycle.

4.1.4 SA-Based Coarse Placement Engine
The details of the SA engine we use are described below.

- Solution space: We define the bin structure to be the same as the given island structure. Components in ICG are placed at island centers subject to the area constraint.
- Neighborhood structure: Two types of moves are used; (1) component move, which randomly selects a module and moves it to another island; (2) component swap, which randomly swaps two components in different islands.
- Cost function: We use the same cost function as proposed in [12]. The delay cost of an edge in ICG is the product of the edge delay and its weight, defined as the following: 
  \[ \text{delay}(e) = \text{delay}(e) \times \text{weight}(e) \]

where \( e \) is the edge in ICG, \( e' \) is the corresponding edge in CDFG and \( \text{weight}(e) \) is the connection between components \( c_i \) and \( c_j \).

5. EXTENSIONS FOR CONTROL FLOW
In this section, we present the extensions of our MCAS system to handle the control flow. Section 5.1 introduces our two-level CDFG representation. Section 5.2 discusses our approach to resolve the multiple definitions problem. Section 5.3 reviews the basic distributed control approach [3] for pure DFG, and shows distributed control generation algorithm for CDFG with multi-cycle control signal interactions.

5.1 CDFG Representation
A two-level CDFG representation is used in MCAS. The first-level CDFG is a Control Flow Graph (CFG). Each node corresponds to a basic block. The edges represent the control dependencies between the basic blocks. Each basic block contains, at most, one operation producing the control signal. If there are two successors, the labels on the control edges indicate which branch is the fall-through path and which one is the taken path. At the second level, each basic block has an internal data flow graph (DFG) representation, which contains a set of operation nodes and edges that represent data dependencies among operation nodes.

Figure 6 gives an example of a two-level CDFG.
introduction of the control flow may lead to the multi-cycle control signals that go over a global interconnect. This is because the producer and the consumer of this signal may be placed in different islands. To account for the multi-cycle control signal during placement, we can extend the ICG by adding in control edges which correspond to the control dependencies in the CDFG.

5.2 Solving Multiple Definitions Problem by Static Single Assignment

A variable may have multiple definitions (assignments) on different execution paths in a CDFG. This can be naturally handled by the conventional centralized register file architecture, which allocates exactly one register for each variable. In the RDR architecture, however, we have to allocate several distinct registers for a single variable when the operations that define this variable are placed in different islands.

Allocating multiple registers for a single variable will cause problems for controlling, which is illustrated in Figure 7. For the sake of simplicity, we only list an assignment statement in every basic block. In Figure 7 (a), variable $a$ has two definitions in basic block 2 (operation $E_2$) and 3 (operation $E_3$). Suppose that operations $E_2$ and $E_3$ are bound to two functional units located in different islands, two local registers within the corresponding islands should be allocated for variable $a$. During the execution of this program, at the use point of variable $a$ (i.e., $c = a + b$ in basic block 4), the controller must decide which register is storing the correct value according to the execution path. Unfortunately, it is prohibitive for an FSM-based controller to record all the traces, which would lead to state explosion.

To address this problem, we apply Static Single Assignment (SSA) [7] transformation to the CDFG. SSA was originally developed to ease the dataflow analysis and optimization for microprocessor architectures. In the SSA form, each variable is only defined once and any redefinition is renamed to be a new variable. A $\phi$ node is inserted at the joint point of multiple definitions for the same variable. It takes a set of possible names and outputs the correct one depending on the path of execution.

![Figure 7. Variable definition problem and SSA solution](image)

(a) Original CDFG; (b) CDFG in SSA format

Figure 7 (b) shows the CDFG in SSA format, where variable $a$ is renamed to $a_1$, $a_2$, and $a_3$. A $\phi$ operation joins the definitions before the use of variable $a$. We use multiplexors (MUX) to implement the $\phi$ operations. After the assignment of $a_1$ or $a_2$, the value will also be transmitted into the register for $a_3$ (denoted as $R_{a_3}$). Therefore, $R_{a_3}$ is ready before the execution of basic block 4. At the use point, the controller can directly use the value in this register regardless of the execution path being taken, thereby avoiding the state explosion problem.

5.3 Distributed Control Generation

5.3.1 Distributed Control for DFG

The RDR architecture requires distributed control for each island. Every local control signal transmission should be made within one clock cycle. Figure 8 illustrates a basic distributed control scheme for the RDR architecture. Each island contains a FSM, which generates control signals for the datapath components in the same island. The control signals include function selections for functional units, MUX selections, and clock enables for registers and memories, etc.

![Figure 8. A basic distributed control scheme for the RDR architecture](image)

For a pure DFG, each controller is essentially a linear sequence of control steps. The control step transitions are triggered by the system clock, regardless of the status of the datapath (i.e., there is no signal from the datapath to the controller). The distributed controllers in different island are independent. They have the identical control step transition diagrams but different output signals.

5.3.2 Distributed Control for CDFG

5.3.2.1 Preliminaries and Motivation

A traditional approach to generate a control unit for an unbounded-latency CDFG is discussed in [13]. A state is generated for each basic block in the CDFG. It consists of a sequence of control steps, and controls the execution of the corresponding basic block. Branch signals generated by the current state activate the next state, according to the status signal generated from the datapath.

A Mealy-type FSM $M$ is a six-tuple $(S, X, \delta, s_0, O, \lambda)$, where

- $S$ is a finite set of states
- $X$ is the input alphabet (signal)
- $\delta : S \times X \to S$ is the next state transition function
- $s_0 \in S$ is the initial state
- $O$ is the output alphabet (signal)
- $\lambda : S \times X \to O$ is the output function

For the RDR architecture, we need to decompose the FSM into a set of distributed FSMS. Assuming the RDR architecture is defined as $I = \{I_1, \ldots, I_n\}$, where $I_i$ represents an island, we should generate a local FSM $M_{I_i}$ for island $I_i$, where

$$M_{I_i} = (S_{I_i}, X_{I_i}, \delta_{I_i}, s_{0_{I_i}}, O_{I_i}, \lambda_{I_i})$$
In addition, we have the constraint that every output signal \( o \in O_{I_i} \) should drive a local resource inside island \( I_i \).

One feasible way is to duplicate the FSM into every island and maintain their synchronization. However, this method is not efficient in terms of both area and delay. When one FSM makes a state transition on a trigger event, all other FSMs should wait for this event to be synchronized. Since the trigger event may take multi-cycles to reach the FSMs in other islands, the synchronization delay should be the delay from the location generating the trigger signal to the furthest island, i.e., for state transition \( \delta(s_y, x) = s_k \),

\[
\text{Delay}(\delta) = \max_{1 \leq i \leq n} \{\text{Delay}(I_{(x, I_i)})\}
\]  

where signal \( x \) is generated in island \( I_{(x, I_i)} \).

### 5.3.2.2 Our Approach

We take a partial state duplication approach to generate distributed controllers. We duplicate states to an island only when they are required by this island. Precisely, suppose \( s_j \) is a state for basic block \( j \) of a CDFG. If in island \( I_{I_i} \), there is no resource allocated for basic block \( j \), then \( s_j \) is not duplicated into \( I_{I_i} \). After the duplication phase, we create state transitions to combine these duplicated states together to form a local FSM.

#### Figure 9. Distributed FSMs generation algorithm

Our distributed control generation algorithm is described in Figure 9. The input is an FSM \( M \) generated by the traditional approach. For the local FSM \( M_{I_i} \) of island \( I_i \), we first generate required states whose outputs drive the logics in this island. The physical locations of the resources are determined by the scheduling-driven placement. In the algorithm, \( s_j \) denotes a state of \( M_{I_i} \) and is a duplicated state of state \( s_j \) of \( M \). We then generate the state transitions and output functions according to FSM \( M \). Note that we also generate a new initial state and reset signal for every distributed FSM.

By this method, we avoid unnecessary state duplications, generate smaller distributed FSMs, and potentially reduce the interaction delay between distributed FSMs, thereby achieving more efficient area and delay cost for controllers. The delay for state transition \( \delta(s_y, x) = s_k \) is

\[
\text{Delay}(\delta) = \max_{1 \leq i \leq n} \{\text{Delay}(I_{(x, I_i)}) | s_k \in S_{I_i}\}
\]

where signal \( x \) is generated in island \( I_{(x, I_i)} \) and \( s_k \) is \( I_i \)'s local duplication of \( s_k \). On average, this delay will be less than the delay of equation (1).

Figure 10 illustrates the distributed controller generation. In this example, we assume a two-island RDR architecture, denoted by islands \( L \) and \( R \). Suppose after the scheduling-driven placement, island \( L \) contains the computation logics for basic blocks 2, 4, and island \( R \) contains computation logics for basic blocks 1, 3, 4. We decompose the original FSM into two FSMs as shown in Figure 10 (c). Interactions between these FSMs, represented as dotted lines, are required to maintain the synchronization. For example, the event signal that trigger the transition from state 2 to state 4 in island \( L \), may be held for multi-cycles. It guarantees that the inter-island communication can reach island \( R \), and trigger the transition from the initial state to state 4 in island \( R \).

#### Figure 10. Distributed control generation for CDFG based on the RDR architecture

### 6. EXPERIMENTAL RESULTS

#### 6.1 MCAS vs. Conventional Flow

We implemented our MCAS system in C++/UNIX environments and compared it with the conventional architectural synthesis flow. The conventional flow is based on the centralized register file architecture. It performs the binding and list-scheduling algorithm sequentially without considering the layout. Both MCAS and the conventional flow share the same backend to generate datapath and controllers.

To obtain the final performance results, Altera’s Quartus II version 2.2 [22] is used to implement the datapath part into a real FPGA device, Stratix™ EP1S40F1508C5. All of the pipelined multipliers are implemented into the dedicated DSP blocks in the Stratix™ device. We set the target clock frequency at 200 MHz and use the default compilation options. We use the LogicLock™ feature to restrict every instance into its corresponding island, and set multi-cycle path constraints for multi-cycle communication paths.

A set of real-life benchmarks is used in our experiments. Eight of them are pure DFG designs from [19], including several different

---

2 Due to the lack of interfaces between different input and output formats, we are not able to provide the comparisons with the existing performance-driven synthesis techniques, such as [9][10].
DCT algorithms, such as PR, WANG, LEE, and DIR, and several DSP programs such as MCM, HONDA, CHEM, and U5ML12. Three other benchmarks containing the control flow come from MediaBench [11] and FFT package[23]. All the benchmarks are data-intensive applications.

The island size of the targeting RDR architecture is determined by the equation discussed in [3], and the consideration of the regularity of the targeting device, which contains 7×2 DSP blocks. We applied a 7×4 RDR architecture in the experiments.

Table 1 shows the experimental results for these designs, including scheduling and binding results, and performance and area results reported by QuartusII. The second column lists the node numbers of the DFG examples. ALU and MUL are the corresponding functional unit usage after the initial binding. For both the conventional flow and MCAS flow, we list register usage (Reg#, by register binding), logic element cost (LE, by QuartusII), control step (CS, by scheduling), clock period (CP, by QuartusII), and total latency (Lat, the product of CS and CP).

On average, MCAS flow introduces more cycles (11%) for the communication between registers. However, since MCAS separates the communications from the computations and applies multi-cycle path constraints for communications, the individual paths in the final layout are reduced, resulting in much smaller clock periods (more than a 30% reduction). Compared with the conventional flow, MCAS reduces the total latencies of the designs by 24% on average.

Although both flows use the same functional unit allocation solution, MCAS uses 18% more LEs and 24% more registers than the conventional flow in order to support the RDR architecture. Such area overhead is non-trivial especially for some heavy-capacity FPGA designs. One can adjust the target clock period (or island size) to explore the area/performance trade-off.

6.2 MCAS vs. Synopsys Behavioral Compiler

We further validate the advantage of MCAS by comparing it with a state-of-the-art commercially available behavioral synthesis tool, Behavioral Compiler (BC) from Synopsys. The experimental flows are illustrated in Figure 11. Since the MCAS system currently only supports C input and BC accepts VHDL format, we should first transform the C descriptions into behavioral VHDL manually. Due to the difficulty of the transformation and the capability of BC, we selected four representative benchmarks, PR, WANG, MCM, and HONDA, from the benchmark set used in Section 6.1. The best benchmarks either contain syntax difficult to transform from C to VHDL (such as array access), or are too large for BC to run through. For example, we cannot obtain the schedule results from BC for DIR, CHEM, and U5ML12 etc. due to the long run time.

For the high-level VHDL or C descriptions respectively, BC and MCAS obtain RTL VHDLs using DesignWare components. We then use the FPGA Compiler to map the RTL VHDL to Altera’s Stratix device. Again, QuartusII is used to get the final placement and routing results.

The Synopsys package we used, including BC, FPGA Compiler, and DesignWare Developer, is Version 2000.11 for sparcOS5. We set the default options for BC and high mapping effort for FPGA Compiler for the experiments.

Table 2 lists the comparison results. The two flows achieve the similar resource cost and scheduled clock cycles. However, MCAS flow obtains a 21% improvement in clock period and a 29% improvement in total latency.

7. CONCLUSIONS AND FUTURE WORK

We presented an architectural synthesis system MCAS for multicycle communication based on the RDR micro-architecture. An efficient solution for integrating scheduling into the global placement is proposed. We also described the generation of distributed controllers for the RDR architecture. Experimental results reported by the commercial place-and-route tools show that effective synthesis methodology can be developed on top of the RDR architecture, and the improvement by applying this methodology is significant.

We are currently working on the synthesis of the control-intensive application to the RDR architecture. Specifically, we are developing global scheduling and resource-sharing techniques to exploit inter basic block parallelism in our two-level CDFG. In addition, we have observed that the steering logic (e.g., MUX) has a big impact on both performance and area in the final layout. We will consider the minimization of the MUX input count for area optimization and layout-driven MUX decomposition for delay optimization.

REFERENCES


Table 1. Comparison of the conventional flow vs. MCAS on CDFG examples, including scheduling and binding results, performance and area usage results from Quartus II

<table>
<thead>
<tr>
<th>Design</th>
<th>Function Unit Binding</th>
<th>Conventional Flow</th>
<th>MCAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Node#</td>
<td>ALU#</td>
<td>MULT#</td>
</tr>
<tr>
<td>pr</td>
<td>46</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>wang</td>
<td>52</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>lee</td>
<td>53</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>mcm</td>
<td>98</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>honda</td>
<td>101</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>dir</td>
<td>152</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>chem</td>
<td>351</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>u5ml12</td>
<td>551</td>
<td>18</td>
<td>13</td>
</tr>
<tr>
<td>matmul</td>
<td>63</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>cftmdl</td>
<td>199</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>cft1st</td>
<td>255</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>Ave Ratio</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2. Comparison of Behavioral Compiler vs. MCAS

<table>
<thead>
<tr>
<th>Design</th>
<th>Flow</th>
<th>ALU#</th>
<th>MULT#</th>
<th>Reg#</th>
<th>LE</th>
<th>CS</th>
<th>CP (ns)</th>
<th>Lat (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>pr</td>
<td>Synopsys BC</td>
<td>5</td>
<td>8</td>
<td>28</td>
<td>2945</td>
<td>25</td>
<td>11.07</td>
<td>276.82</td>
</tr>
<tr>
<td></td>
<td>MCAS</td>
<td>6</td>
<td>2</td>
<td>35</td>
<td>2575</td>
<td>29</td>
<td>8.99</td>
<td>260.77</td>
</tr>
<tr>
<td>wang</td>
<td>Synopsys BC</td>
<td>7</td>
<td>8</td>
<td>36</td>
<td>3605</td>
<td>29</td>
<td>11.96</td>
<td>346.85</td>
</tr>
<tr>
<td></td>
<td>MCAS</td>
<td>5</td>
<td>8</td>
<td>46</td>
<td>4194</td>
<td>20</td>
<td>9.01</td>
<td>180.28</td>
</tr>
<tr>
<td>mcm</td>
<td>Synopsys BC</td>
<td>23</td>
<td>7</td>
<td>142</td>
<td>6253</td>
<td>43</td>
<td>12.55</td>
<td>539.86</td>
</tr>
<tr>
<td></td>
<td>MCAS</td>
<td>6</td>
<td>3</td>
<td>50</td>
<td>4350</td>
<td>38</td>
<td>9.76</td>
<td>370.99</td>
</tr>
<tr>
<td>honda</td>
<td>Synopsys BC</td>
<td>8</td>
<td>14</td>
<td>44</td>
<td>6128</td>
<td>29</td>
<td>11.75</td>
<td>340.62</td>
</tr>
<tr>
<td></td>
<td>MCAS</td>
<td>6</td>
<td>8</td>
<td>56</td>
<td>6294</td>
<td>25</td>
<td>9.42</td>
<td>235.60</td>
</tr>
<tr>
<td>Ave Ratio</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.94</td>
<td>0.90</td>
<td>0.79</td>
<td>0.71</td>
<td></td>
</tr>
</tbody>
</table>