Efficient Resource Arbitration in Reconfigurable Computing Environments*

Iyad Ouaiss and Ranga Vemuri
Digital Design Environments Lab, University of Cincinnati
Cincinnati, OH 45221-0030, USA
{iouaiss, ranga}@ececs.uc.edu

Abstract

*In a multi-*FPGA *synthesis system, ideally the designer has only an abstract view of the board architecture. This abstract modeling of the underlying reconfigurable computer poses complex challenges to the synthesis and partitioning tools. Since the design specification is not constrained by the number of memory segments on the board or the number of pins between* FPGAs*, it is difficult for the* CAD *tools to transform the design into one that maps onto the multi-*FPGA *board. This paper describes an arbitration mechanism that bridges the abstraction between the input design and the reconfigurable architecture. Since this mechanism allows such architecture abstraction between the design and the board, it becomes easier to port a design from one target architecture to another. This arbitration mechanism introduces very little overhead in terms of area and delay. It has been used in data-dominated applications; in this paper, Fast Fourier Transform (*FFT*) is shown as an illustrative example.*

1 Introduction

FPGAs enable designers to perform fast prototyping of an ASIC design. Also, when a small number of design implementations is required, FPGAs provide a cheap and performance efficient alternative to hardware (ASICs) or software implementations. However, due to the rather limited programmable hardware area that FPGAs offer, many vendors introduced multi-FPGA reconfigurable boards [3, 16, 6].

These multi-FPGA reconfigurable computers eased the area constraint but introduced design complexities. One of the major problems in synthesis tools for reconfigurable computers is the lack of flexibility. Each tool is usually targeted for one specific Reconfigurable Computer (RC) board. Furthermore, the tools expect a tight relationship between the input design and the actual board. If the target board changes, the design would require major modifications before it can be synthesized.

1.1 Memory conflicts

If the design makes use of L logical data segments and the board has P physical memory segments, then two cases arise: when L is less than or equal to P, and when L is greater than P. It is assumed that the total amount of memory used at one time in the design must not exceed the total memory available on the board. Obviously, if L is less than or equal to P, then the mapping is straightforward: each data segment is mapped to an individual physical memory bank.

On the other hand, when L is greater than P, there are more data segments in the design than there are physical memory banks on the multi-FPGA board. In this case, the mapping becomes difficult since more than one data segment has to be mapped to the same physical bank. Even if the two data segments can fit on a single physical bank, there might still be memory access conflicts.

So far, memory access arbitration has not posed a problem since very few synthesis systems cater to several target architectures [2]: the trend is to model a design at a lower level of abstraction where knowledge of the target board is given.

1.2 I/O pin limitations

FPGA pins limitation poses a great problem for partitioning and synthesis. The cutset between partitions limits how much the partitioner can fit on each partition. FPGA pins limitation is a problem that is posed in any synthesis framework. With the advent of technology, the trend points to an increase in gates faster than an increase in pins. Effectively, FPGAs are getting bigger without an increase in the number of available pins. This might not pose a problem when the design fully fits on a single FPGA. However, when the design requires a multi-FPGA system, cutsets between the different partitions typically govern the amount of logic that can go in each FPGA: The bigger the partition, the larger the cutset between the partitions. Thus, the low number of pins on FPGAs (compared to the number of gates that the FPGA offers) forces designers to under-use the FPGA.

*This work is supported in part by the US Air Force, Wright Laboratory, WPAFB, under contract number F33615-97-C-1043.

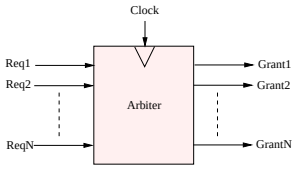


Figure 1. Generic N-bit arbiter

I/O pin management comes in two flavors: hardware specific and synthesis specific. Hardware specific management refers to RC architectures that have complex interconnect structures such as programmable crossbars or meshes [14, 4]. In order to bridge the gap between designs and variable board architectures, software techniques are used to make the design board-independent. It becomes the partitioner and the synthesis tools' responsibility to adapt the design to the RC architecture utilized. Several mechanisms exist to reuse pins for several connections; Virtual wires [12] offer a way of overcoming pin limitations in FPGAs by statically scheduling data transfers so that multiple transfers re-use the same set of pins. This comes at the price of statically scheduling accesses. On the other hand, Vahid used functional partitioning and the concepts of Function-Bus interprocessor bus and port calling to reduce the I/O requirements [8]. This solution came at the price of intrusive modifications to the partitioning and synthesis process.

1.3 Generic arbitration

It would be advantageous to have a mechanism that would solve both memory conflict and pin limitation problems. At the same time, this mechanism should not restrict scheduling of resource accesses or introduce complexity to the partitioning/synthesis process.

An arbiter should be introduced for each resource that is to be shared between processes executing in parallel. The size of the arbiter depends on the number of processes accessing that resource; and a general N-bit arbiter is shown in Figure 1. In the literature, arbiters are also referred to as mutual-exclusion circuits or interlocks [13].

For each process accessing a shared resource, two wires are introduced — *Request* and *Grant* — between the process and the resource's arbiter. When a process wants to access the shared resource, it asserts its *Request* line and waits until its *Grant* is asserted. Thus, at any given point, the duty of the arbiter is to receive zero or more *Request*s from processes and issue zero or exactly one *Grant*.

Most importantly, for the arbitration mechanism to be successful, it should be automated in the RC design environment. The advantages of this automation are two-fold: first, it allows the designer to produce architecture-independent designs; second, it allows the tools to target a generic set of RC boards. A generic target architecture might have a variable number of processing elements, local memory banks, shared memory banks, as well as a variable interconnection topology. This paper presents an automatic arbitration

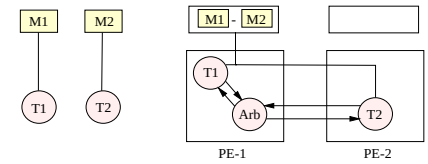


Figure 2. Memory access arbitration

mechanism in the synthesis/partitioning RC environment.

The rest of this paper is organized as follows: Section 2 describes the functioning of arbitration for both memory and I/O pins resolution. Section 3 lists the features that an arbiter should have. Section 4 introduces one implementation of an arbitration mechanism and discusses its suitability to this framework. Section 5 shows how arbitration fits in the RC framework and shows an actual implementation of arbitration in a popular digital signal- processing algorithm. Finally, Section 6 provides a brief conclusion.

2 Arbitration Mechanism

In this discussion, the input designs being partitioned and synthesized are assumed to be in the form of taskgraphs. Taskgraphs contain two types of objects called tasks and memory segments. *Tasks* represent synthesizable elements of computation and *memory segments* represent elements of data storage. *Channels* are used to represent inter-task and task-to-memory communications.

The Unified Specification Model format, USM [9], is a candidate specification language that provides a hierarchical representation for specifying the behavior of a design. All tasks in the USM are simultaneously executing so as to model concurrency. Other specification languages based on the taskgraph representation are available in the literature [7, 15, 5].

2.1 Memory arbitration

Consider the case when a task T1 reads/writes from data segment M1 and task T2 reads/writes from data segment M2 (Figure 2a). If the two memory segments M1 and M2 are assigned to the same physical memory bank on the RC board, then tasks T1 and T2 are sharing the same address lines, data lines, and read/write mode line of the memory bank. But this creates a conflict since tasks T1 and T2 might be independent from one another (i.e. executing in parallel).

Mutual exclusive access cannot be ensured for the address/data lines as well as the select mode line. So, if T1 is writing to the address lines in clock step c1, T2 cannot be accessing the memory during this step. Moreover, during clock step c1, T2 must tristate its access to the address lines. In conclusion, when two memory accesses are occurring through the same physical memory bank, an arbitration scheme has to be present to avoid any conflicts on the bank. For the example shown in Figure 2a, an arbiter solution is shown in Figure 2b.

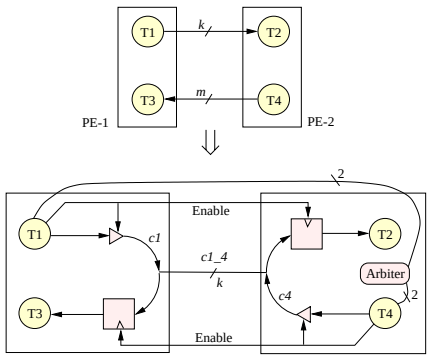
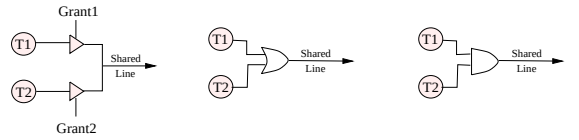


Figure 3. Channel arbitration



a) Tristate access b) Active-high access c) Active-low access

Figure 4. Accessing shared lines

2.2 Channel arbitration

Pin limitation between processing elements might cause a practical problem when a design has to be partitioned across several connected processing elements. Similar to the memory sharing mechanism described earlier, when the number of physical channels on the board is less than the number of logical connections required, then physical channels can be re-used. A single physical channel can be used by more than one pair of writer/reader provided that arbitration is introduced to avoid access conflicts.

An example of channel sharing is shown in Figure 3. Two logical channels (k-bit and m-bit wide, with $m<k$) are merged onto one k-bit physical channel. Arbitration circuitry (registers and tristate buffers) is required to ensure proper functioning of the shared channel; they are discussed in detail in Section 4.3.

Finally, irrespective of the type of resource that is being shared, tristating of shared lines is required when a task is not accessing the resource. This can be seen in Figure 4a where the processes' *Grant* lines control the "enable" lines of their tristates. Only when a process is granted access to the shared resource (i.e. its *Grant* is asserted), should the process drive the shared line. But what happens if all tasks are tristating their access to the shared line (in other words, none of the tasks is accessing the resource at a specific instant)? In Figure 4a, if both T1 and T2 are *not* accessing the shared resource, then the line connected to the shared resource is in a high- impedance state. This might cause the design to malfunction since the actual value of the shared line is unknown. In the case of address and data lines, this is not a problem, but for the select mode of a memory (write on high), for instance, it can produce unwanted effects. In this case, instead of tristating the shared line, a task should disable its access to the select mode of the memory by driving a zero, and all lines are OR-ed to drive the select mode of the shared memory. This ensures that even if the memory is idle at some time, its select mode will be driven to zero (read mode) and no unwanted writes would occur. Hence, all resource inputs that are active-high should follow the scheme presented in Figure 4b; whereas active-low inputs should follow the one presented in Figure 4c.

In conclusion, if two or more tasks are accessing a single physical resource (memory, channel, etc.), arbitration has to resolve any access unless the tasks are globally scheduled to avoid conflicts. The latter statement refers to the case where scheduling of all tasks across *all* processing elements is done simultaneously so as to avoid resource conflicts. Global scheduling of the design is feasible but it requires a complicated controller model and it prohibits real parallelism in the execution when processes contain unpredictable loops and conditionals.

3 Choice of Arbiters

The implementation and functioning of arbiters depend on the environment that they will be used in as well as other constraints that the application imposes. In the RC framework, the constraints that an arbiter should follow are fairness, low overhead in terms of area and delay, and ease of insertion and synthesis.

1. *Fairness:* Similar to concerns in many aspects of multi-tasking operating systems, the arbiter should ensure mutual exclusion, prevent starvation, and prevent deadlock [1]. The reader is referred to [1] for an extensive explanation of these concepts.
2. *Low overhead:* The introduction of arbitration to the design should not involve a substantial increase in area (function generators or CLBs) or a substantial slowdown in the design's clock speed. Also, the latency increase due to arbitration should be kept to a minimum.
3. *Extendibility and ease of insertion:* The process of introducing arbitration to the design should be simple, fast, and fully automatable. The arbiter generation should be parameterized such that the mechanism can be extended to any number of tasks being arbitered.

In the next section, a specific implementation of the arbitration mechanism is shown, and its conformity to the above requirements is analyzed.

4 Implementation of Arbitration

In the literature, there exist several algorithms for contention resolution each with its shares of advantages and drawbacks [13, 1]. Techniques such as *random*, FIFO, *round-robin*, and *priority-based* were examined. Given

its complexity and the type of applications that RC architectures help solve, the *round-robin* technique proved to best fit our RC framework. In this technique, requests are handled in a cyclic manner; whereas in the *random* technique, requests are handled in a random manner; for FIFO, requests are handled in the order in which they arrive; and for *priority-based*, requests are handled in a statically-determined weighed order. With the exception of the *round-robin* technique, all other techniques introduced considerable complexity in the required hardware. In the RC framework, the required hardware made the arbiter either too slow or too large thus placing a considerable constraint on the synthesized design.

At anytime during the execution of a design, a *round-robin* arbiter — corresponding to a shared resource — resides in a single state. The number of states in the arbiter depends on the number of tasks accessing that resource. Each task being arbitrated introduces two states. For task *i*:
C_i corresponds to the state when task i is exclusively accessing the shared resource.
F_i corresponds to the state when none of the tasks are accessing the shared resource and task i has the highest access priority to the shared resource.

Thus, for *N* tasks accessing a single resource, the *round-robin* arbiter moves within the following set of states:

$\Phi = C_1, C_2, ..., C_N, F_1, F_2, ..., F_N$

The arbiter takes, as input, request signals from all tasks and produces, as output, a grant signal for each task. The set of input signals and output signals are respectively:

$\sigma = R_1, R_2, ..., R_N$

$\Omega = G_1, G_2, ..., G_N$

Based on the set of inputs (σ), outputs (Ω), and the possible set of states (Φ), the transition mechanism of the *round-robin* arbiter is shown in Figure 5.

4.1 Fairness

Since the *round-robin* arbiter is implemented as an FSM and since each state in the FSM acknowledges *at most* one request, mutual exclusion is ensured. Given the above assumption, the *round-robin* arbiter is designed such that starvation is avoided. Since the order of requests is cyclic, it is guaranteed that all tasks requesting access to the shared resource will be acknowledged. Furthermore, with the *N*-input arbiter implementation presented in this paper, it is also guaranteed that a task requesting at a certain instant will have its grant at most after (*N*-1) tasks. This is the upper limit where not only all other (*N*-1) tasks happen to request access to the resource, but also the task in question happens to be at the end of the current order. Furthermore, the *round-robin* implementation prevents deadlock. The arbiter can handle any number of requests occurring at the same time. In the current form in which the *round-robin* arbiter is presented, it does not support preemption. However,

```
case current_state is
    when F_i ⇒
        case σ is
            when zeroes ⇒
                next_state = F_i
                Ω = zeroes
            when R_i ⇒
                next_state = C_i
                Ω = G_i
            when not(R_i) and R_{i+1} ⇒
                next_state = C_{i+1}
                Ω = G_{i+1}
            when not(R_i) and not(R_{i+1}) and R_{i+2} ⇒
                next_state = C_{i+2}
                Ω = G_{i+2}
            when etc...
        end case
    when C_i ⇒
        case σ is
            when zeroes ⇒
                next_state = F_{i+1}
                Ω = zeroes
            when R_i ⇒
                next_state = C_i
                Ω = G_i
            when not(R_i) and R_{i+1} ⇒
                next_state = C_{i+1}
                Ω = G_{i+1}
            when not(R_i) and not(R_{i+1}) and R_{i+2} ⇒
                next_state = C_{i+2}
                Ω = G_{i+2}
            when etc...
        end case
end case
```

Figure 5. round-robin transition algorithm

in this RC framework, since arbitration will be automated, preemption is not required.

4.2 Low overhead

In order to quantitatively evaluate the overhead introduced by the *round-robin* arbiter, an arbiter generator was implemented. It takes the number of tasks to be arbitrated (*N*) as input and it generates a corresponding VHDL file. The generator also has the option to produce different encoding schemes for the FSM (e.g. one-hot encoding, compact encoding, or synthesis tool's default encoding). The arbiter generator was executed for *N* in the range [2; 10] and each of the generated VHDL arbiters was then synthesized using two popular synthesis tools (Synplify 5.1.4 by Synplicity, Inc. and FPGA_express 2.1 by Synopsys, Inc.) targeted for the Xilinx XC4000e series with a -3 speed grade [17]. The synthesized files were then taken through Xilinx M1.5 logic and layout synthesis tools and the area values are reported in Figure 6 (in terms of CLBs). Note that Synplify used one-hot encoding regardless of what the VHDL files specified. FPGA_express, on the other hand, implemented both schemes. Also, note that for *N*=9 and *N*=10, even though the tool execution time of Synplify was very small compared to FPGA_express, its results were still satisfactory.

Similarly, maximum clocking speeds for each arbiter were obtained from Xilinx's estimates. These values are shown in Figure 7; they were obtained by placing timing constraints on the Xilinx partitioning and routing tools. It is important to note that no timing constraints were issued to Synplify or FPGA_express. Thus, it is possible to obtain even faster implementations.

It can be seen from the values reported in Figure 6 and

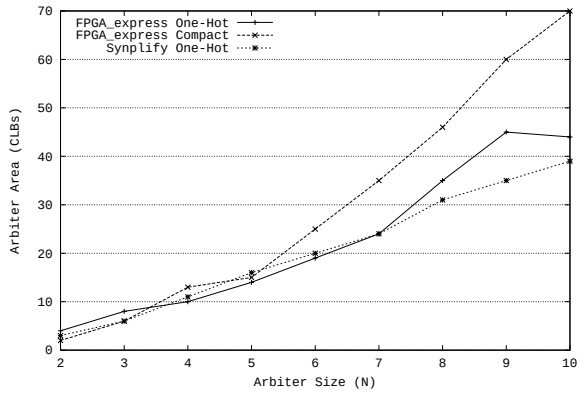


Figure 6. N-input arbiter sizes in CLB s

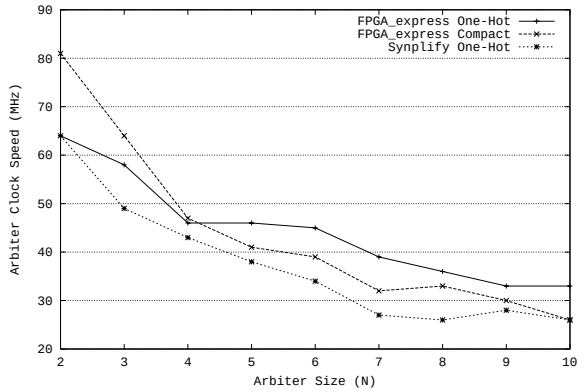


Figure 7. N-input arbiter clock speed in MHz

Figure 7 that *round-robin* arbitration introduces very little overhead to the design. In terms of function generators, a 10-bit arbiter added about 40 CLBs to the design. In our experience, arbiters in the range [2; 6] were the arbiters mostly used for our example taskgraphs; larger arbiters were very seldom introduced. With the Xilinx XC4000e series FPGAs, this area overhead is very acceptable. Similarly, in terms of clocking speeds, the arbiters seem to outperform any design of reasonable size. Without, arbitration, designs occupying 50% or more of the FPGA usually reported clocking speeds of less than 25MHz (XC4000e -3 speed grade). Since 10-bit arbiters clocked at 26MHz, they did not introduce any overhead on the clock speed.

4.3 Extendibility and ease of insertion

If a logical data segment is being accessed by more than one task, the designer is responsible for arbitrating access to the segment. However, an arbiter is automatically introduced when multiple data segments are mapped onto the same physical memory bank, and *multiple* tasks are accessing this physical memory bank.

Since arbiters are pre-characterized for the number of inputs and outputs, their area, and their delay, a precise estimation can be performed by the partitioners to ensure the fitness and speed of the contemplated design.

The arbiter synthesis is extendible for all values of N.

```
c := 13                    c := 13
mem[1] := ...              Req := 1
mem[2] := ...              Wait for (Grant == 1)
...                        mem[1] := ...
a) Original code           mem[2] := ...
                           Req := 0
                           ...
                           b) Arbitrated access
```

Figure 8. Task modification process

Time Step	Task 1	Task 2	Task 3	Task 4
1	c1 := 10	...	...	...
2	...	...	...	c4 := 102
3	...	x := c1	...	...

Table 1. Shared channel example

First, it generates arbiters of the appropriate sizes. Second, for each affected task, a (*Request*, *Grant*) pair of lines is added to the task's ports. And, within the task, for each access to the resource, the code is modified such that the task requests access from the arbiter, waits until it receives a grant, performs its usual access to the resource, then deasserts its request.

A task that wants to continuously access a shared resource, has to make its *Request*=0 between each "M" accesses. This is done in order to ensure that no task would have to wait a long time before it can access the resource. An example for M=2 is shown in Figure 8. Assuming a task will receive its grant immediately, each arbitered access incurs two extra clock cycles due to the arbitration protocol.

Note that when a task is not accessing the shared resource, it must set all shared lines to their default states; e.g. data and address lines are tri-stated, memory write/read select is set to read.

The above discussion applies to all shared resources. In the case of channel sharing, however, an additional concern must be addressed: As seen in Figure 3, for each receiving end of a shared channel, a register will be introduced whose enable originates from the source task (whereas for non-shared channels, a register is introduced at the source end). The reason for having registers at the receiving ends of each transfer is to ensure that data going to one of the targets will not be overwritten by future transfers. In addition, the presence of the registers allows transferred data to be stored and subsequent transfers to take place immediately.

For the example of Table 1, if c1 and c4 were to be merged into a single shared channel, c1_4, then we need to store the *c1 := 10* assignment from Task 1 before Task 4 performs the *c4 := 102* assignment. By having the register of c1 at Task 2's end, the value will remain indefinitely for Task 2 to consume regardless of when Task 4 writes to the shared channel.

An arbiter is required when different sources of the shared channels belong to different tasks. If all sources belong to the same task, then there is no need to introduce an arbiter since the channel access would be implicitly arbitrated by the schedule of that task. Arbiter lines (*Request* & *Grant*) are added for every task containing one or more

shared channel writes. Also, a tri-state buffer will be introduced at the output of each source task whose enable is the same as the one for the introduced register.

In conclusion, arbiters are introduced after spatial partitioning occurs. The hardware required for arbitration is pre-characterized for area and speed thus making the partitioners' estimation accurate. In addition, the number of clock cycles introduced by the task modification process is fixed and known in advance of synthesis.

5 Arbiter Synthesis in SPARCS

SPARCS (Synthesis and Partitioning for Adaptive Reconfigurable Computing Systems) [10] is an integrated design system for automatically partitioning and synthesizing designs for reconfigurable boards with multiple field-programmable devices. The SPARCS system accepts design specifications at the behavior level, in the form of task graphs, where each task is specified in VHDL [11]. In SPARCS' view, a reconfigurable computer contains multiple FPGAs and multiple memory modules connected to each other through a static or reconfigurable interconnection fabric. This view admits the use of SPARCS to retarget the specification to a variety of RCs containing local and/or shared memories among the FPGAs and dedicated and/or shared connections among the memories and the FPGA units.

SPARCS contains: 1) a temporal partitioning tool to temporally divide and schedule the tasks on the reconfigurable architecture; 2) a spatial partitioning tool to map the tasks to individual FPGAs; and 3) a high-level synthesis tool to synthesize efficient register-transfer level designs for each set of tasks destined to be downloaded on each FPGA. Commercial logic and layout synthesis tools are used to complete logic synthesis, placement, and routing for each FPGA design segment. In addition to these tools, SPARCS automatically inserts resource arbitration, performs memory synthesis, and generates interconnection information.

Figure 9 shows the role of arbiter synthesis in the SPARCS flow. The arbiter synthesis tool can be adapted to a variety of synthesis/partitioning flows since it is contained as a separate module.

A variety of applications have been synthesized through SPARCS. In this paper, we describe the Fast Fourier Transform (FFT) application. The 4x4 pixel, 2-dimension FFT algorithm was partitioned and synthesized in the integrated SPARCS environment [10]. The main inputs to SPARCS consisted of:

1. *The* FFT *taskgraph:* The taskgraph for this application is shown in Figure 10. It follows the USM format [9] where the "F" tasks represent the first FFT dimension that is performed on an input image, whereas the "g" tasks represent the second FFT dimension performed on the complex-valued output of the first dimension.

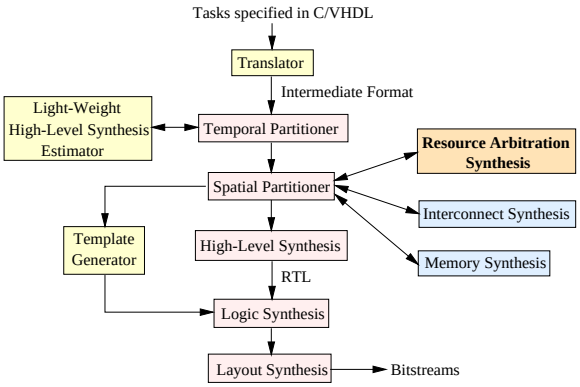


Figure 9. Arbiter synthesis in SPARCS

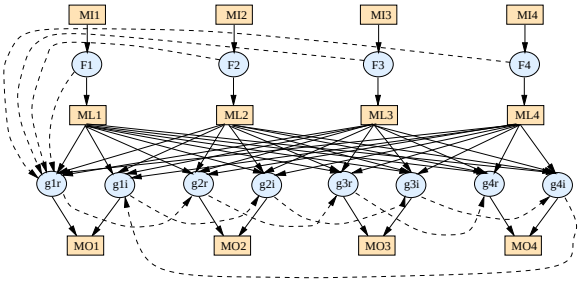


Figure 10. FFT Taskgraph

   The solid arrows represent the data transfer between tasks and memory segments and the dashed arrows are used to specify control dependencies among tasks.
2. *The target* RC *architecture:* The $Wildforce^{TM}$ board from Annapolis MicroSystems Inc. was used for this application. The board has four processing elements (Xilinx XC4013e-3 FPGAs) with each a local memory (32Kbytes) attached to it. Each processing element is connected to its neighbor(s) by a set of 36 fixed pins. Also, each processing element has a 36-bit connection to a programmable crossbar interconnection structure. The crossbar can be programmed to connect any two or more processing elements together.

After partitioning, arbiter insertion, and synthesis, the tool produced three temporal partitions, of which temporal partition #0 is shown in Figure 11. This partition contains two arbiters: a 6-bit (Arb6) and a 2-bit (Arb2). The 6-bit arbitrates access to the local memory that contains all "ML" memory segments. Since all 6 tasks in this temporal partition access the "ML" memory segments (as can be seen in Figure 10), a 6-bit arbiter was introduced. The arbiter insertion assumed that all 6 tasks were executing in parallel, thus access should be arbitrated. In reality, since the "g" tasks execute after termination of the "F" tasks ("g" tasks have to wait until the "F" tasks finish writing their outputs), there is no memory conflicts between them. The arbiter insertion tool can easily detect this scenario based on the dependencies between the tasks. Instead of inserting an arbiter between these tasks, it should only ensure that the shared

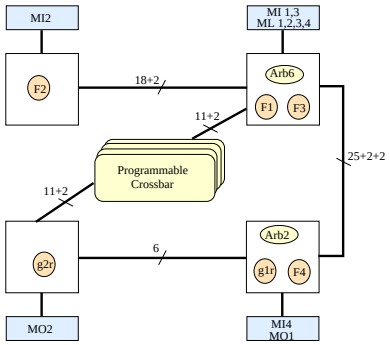


Figure 11. FFT temporal partition #0

data, address, and select lines are appropriately set in tasks F after they finish execution (tri-stated, OR-ed, or AND-ed as explained in Section 2.2). On the other hand, it should be noted that access of the "g" tasks to the "ML" segments is implicitly arbitrated by the designer. Since each "ML" data segment is assumed to be a single resource, the designer should ensure that not more than one "g" task (or any other task for that matter) can access it at one time.

Temporal partition #1 contained one 4-bit arbiter and partition #2 did not require arbitration. Thus, for the entire 4x4, 2-D FFT, a total of three arbiters were introduced and the design clocked at about 6MHz. Even with this low clocking frequency, the small amount of memory available in each bank, and the rather small size of the processing elements, the RC's hardware execution (4.4sec for a 512x512 image) proved faster than a software execution on a Pentium system running at 150MHz, with 48MB of RAM (6.8sec execution time)! It should be noted that, even without arbitration, the number of temporal partitions produced would have remained the same as well as the clocking speed of the overall design.

Finally, several modifications to the partitioning and synthesis process could be made in order to obtain better results. For instance, providing constraints to the logic and layout synthesis tools could have resulted in faster designs. Also, as discussed above, by not arbitrating tasks "F" and tasks "g", the latency of the design could be reduced since tasks "F" do not have to go through the arbitration protocol in order to access the "ML" memory segments.

6 Conclusion

We have provided an explanation of arbitration in the RC framework and introduced the round-robin arbitration mechanism as a solution. By analyzing the features of the arbitration scheme, we showed how it fits nicely in this environment. A partitioning/synthesis system can freely distribute data segments onto physical memory banks, reuse FPGA pins if required, automatically recognize the need for arbiters and insert them, and ensure proper execution of the design without a substantial loss in area or speed. The Fast Fourier Transform application is a good candidate for such arbitration mechanism and we show how it was synthesized for the Wildforce RC board. With minimal user intervention, the synthesis process produced a solution for a low-end commercial board that was faster than an equivalent software solution executing on a Pentium 150MHz platform. It is noted that without any modifications to the input task-graph, FFT can be synthesized for different architectures using the same set of partitioning/synthesis tools. As future work, it would be interesting to implement different task modification schemes (refer to Section 4.3) in order to decrease the number of clock cycles due to arbiter insertion. Also, preemption techniques could be introduced to ensure that no task is granted access to a shared resource and never relinquishes its request.

References

[1] A. Silberschatz, P. Galvin. *"Operating System Concepts"*. Addison-Wesley, 4th edition, 1994.

[2] A.A.Duncan, D.C.Hendry and P.Gray. "An Overview of the Cobra-ABS High-Level Synthesis System for Multi-FPGA Systems". In *Proceedings of FPGAs for Custom Computing Machines*, pages 106–115, Napa Valley, California, 1998.

[3] Altera Corporation. Reconfigurable Interconnect Peripheral Processor (RIPP10). http://www.altera.com.

[4] Brian Box. "Field Programmable Gate Array Based Reconfigurable Preprocessor". In *IEEE Workshop on FPGAs for Custom Computing Machines*, 1994.

[5] C. Hoare. "Communicating Sequential Processes". In *ACM Communications*, volume 21, pages 666–677, 1978.

[6] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt. "The NAPA Adaptive Processing Architecture". In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

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

[8] Frank Vahid. "Techniques for Minimizing and Balancing I/O During Functional Partitioning". In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 18, January 1999.

[9] I .Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri. "A Unified Specification Model of Concurrency and Coordination for Synthesis from VHDL". In *Proceedings of the 4th International Conference on Information Systems Analysis and Synthesis*, July 1998.

[10] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri. "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures". In *Proceedings of the 5th Reconfigurable Architectures Workshop*, pages 31–36. Springer, March 1998.

[11] IEEE Standard VHDL Language Reference Manual. IEEE Standards Office. New York, NY, 1993.

[12] J. Babb, R. Tessier, A. Agarwal. "Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators". In *Proceedings of FPGAs for Custom Computing Machines*, 1993.

[13] J. Rabaey. *"Digital Integrated Cicuits: A Design Perspective"*. Prentice Hall, 1996.

[14] S. Walters. "Computer-Aided Prototyping for ASIC-Based Systems". In *IEEE Design and Test of Computers*, June 1992.

[15] U. Steinhausen, R. Camposano, et al. "System-Synthesis Using Hardware / Software Codesign". In *International Workshop on Hardware-Software Co-Design*, October 1993.

[16] Wildforce multi-FPGA board by Annapolis Micro Systems, Inc. http://www.annapmicro.com.

[17] Xilinx, Inc. "The Programmable Logic Data Book", 1998.