The General Rapid Architecture Description[1]

Carl Ebeling
Department of Computer Science and Engineering
University of Washington

UW CSE Technical Report UW-CSE-02-06-02

Abstract

This document describes the general Rapid architecture, which generalized the Rapid benchmark architecture developed by the group. Features of the benchmark architecture that are implementation-dependent are excluded or replaced by generic features that are specialized by any particular implementation. The result is the description of a general coarse-grained configurable architecture.

Introduction

Mobile embedded computing devices have become one of the most important components of the emerging information technology infrastructure. As wireless communication bandwidth and the computational capacity of mobile devices increase, they will become even more important, displacing in many instances the PC as the computing platform of choice. The characteristics of these devices are very different from the PC: they are highly optimized to one set of tasks, they are cheap, portable and ubiquitous, and they have a transparent user interface, blending into the environment in contrast to traditional computers. These characteristics will place increasingly high demands on the platforms used to implement these devices. Performance demands are growing rapidly, more rapidly than the underlying circuit technology, driven by multimedia applications, advanced user interfaces that use speech recognition technologies, and high bandwidth wireless communication. These demands must be met within the severe low power and low cost constraints of portable devices, which presents a formidable challenge to the embedded systems architect.

Current embedded systems platforms are based on systems-on-chip (SOC) technology that integrates programmable processors along with application-specific integrated circuit (ASIC) components on a single chip. The processors are used for as much of the application as possible within the performance, power and cost constraints, because they are easier to program and debug, and they can be reprogrammed as the application requirements change. However, processors are at least two orders of magnitude less efficient than ASIC circuits in terms of combined performance, cost and power. Thus, much of the functionality of embedded SOC platforms must be implemented using ASIC components. As performance demands increase, platforms necessarily become more specialized because they must rely more on the ASIC components. This problem is exacerbated by rising NRE costs for chip fabrication that already exceed $1M. It is simply no longer possible to build an efficient, cost-effective embedded computing platform unless it is for a high-volume, narrowly focused application.

Adaptable architectures, sometimes called reconfigurable architectures, have long been proposed as the way to bridge the gap between processors and ASICs. Although they have usually been proposed for performance reasons, in the embedded systems area, it is the combination of power, cost and performance that must be optimized. Although adaptable architectures potentially could have a huge impact on the way embedded system platforms are built, there are several reasons why this has not yet happened.

First, most adaptable architectures have been based on FPGAs. However, FPGA-based architectures are not appropriate for mobile embedded applications because the very high overhead of providing complete generality results in at least a 100x penalty with respect to ASICs in terms of performance for a given cost and power. Coarse-grained adaptable architectures, which are based on efficient custom function units, have much less overhead, and thus are more cost and power-effective. However, research on coarse-grained adaptable architectures has been very limited, in part because of the infrastructure required to implement and experiment with these devices.

[1] This research was supported in part by the National Science Foundation under Grant No. 9901377.

Second, reconfigurable components have typically been viewed as hardware components whose functionality is defined using a hardware description language like Verilog, Handel-C or JHDL. There has been a lot of progress in the high-level synthesis community on synthesizing high-level programming languages like C or Java into hardware. But high-level synthesis assumes that an arbitrary hardware structure can be built, and uses this freedom to find a solution that optimizes some combination of cost and performance. Although this work applies to FPGA-based architectures, it does not translate well to coarse-grained adaptable architectures, which offer a very constrained substrate. That is, the set of function units is fixed and the way they are interconnected is fixed in an adaptable architecture.

This document describes the general Rapid architecture, which is a generalization of the Rapid benchmark architecture. For details of the implementation of the Rapid benchmark architecture, refer to [1, 2, 4]. A separate document describes our proposed approach for compiling programs to coarse-grained adaptable architectures, and specifically Rapid.

Overview of the Rapid Reconfigurable Architecture

Rapid is a coarse-grained reconfigurable architecture intended for applications with large amounts of fine-grained parallelism typical of those found in embedded systems. The goal of the Rapid architecture is to achieve much of the cost and power efficiency of ASICs but with the flexibility of programmable processors. The Rapid architecture achieves this by executing many operations in parallel while minimizing the data movement and control overhead that burns so much power in conventional processor architectures. We envision that components based on coarse-grained adaptable architecture like Rapid will provide much of the functionality currently delivered by DSPs and ASICs in future SOC-based embedded system platforms.

As shown schematically in Figure 1, the Rapid architecture is composed of:

- A set of application-specific function units, such as ALU's, multipliers, shifters and bit-configurable operators.
- A set of application-specific memory elements, including registers and small data memories.
- A set of input and output data ports that interface the datapath to external memory and streaming data interfaces.
- An interconnection data network that connects the function units, memory elements and data ports together using a combination of configurable and dynamically controlled multiplexers.
- A sequencer that generates "instructions" that control the operation of the Rapid datapath.
- An interconnection control network that generates the individual control signals based on the instructions and status signals generated by the function units.

The following sections will describe each of these components in turn and discuss the research questions posed in the context of the general Rapid model. It is assumed that the reader is familiar in general terms with the Rapid benchmark architecture, as described in the paper published in Advanced Research on VLSI conference in 1999.

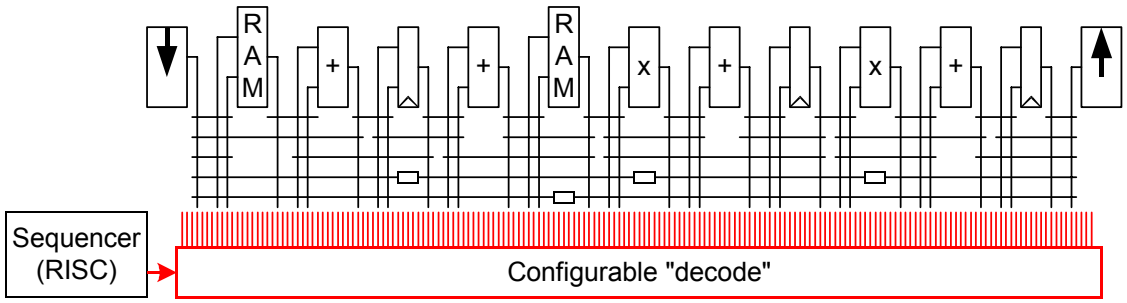

Figure 1 - Abstract view of the Rapid architecture

Application-Specific Function Units

The function units in the datapath provide the operations that are performed by the target applications on the Rapid array. This means that the set of target applications must be known, or that the domain must be well understood. In

addition, it means that the set of computations in those target applications to be implemented must be identified based on their computation requirement and their suitability for implementation on Rapid. For example, computations with little available fine-grained parallelism are generally not suited to Rapid. (By fine-grained parallelism, we mean the ability to execute many closely related operations in parallel in a clock cycle given the constraints of data dependencies and memory bandwidth. In the processor world, this would be called instruction-level parallelism.)

For signal processing applications, the set of function units includes multipliers, ALUs and shifters. For a speech recognition backend, the function units would include a unit to add numbers represented in logarithm format, and units that handle a specialized number representation for probability distributions. For encryption, the function units could be configured to perform the required bit-processing computations.

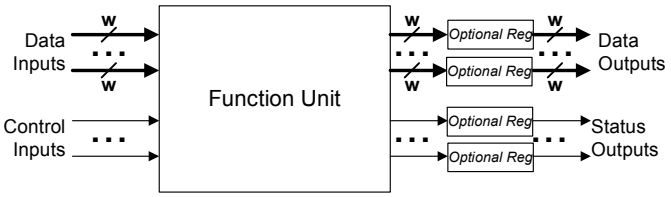

Figure 2 - A Generic Datapath Unit

A generic function unit is shown in Figure 2. In addition to data inputs and outputs that are connected to other function units using the data interconnect network, function units have control inputs that determine the operation to be performed and status outputs that give information about the result. These status signals can be used to modify the data computation or affect the control flow of the program. The control inputs and status outputs are connected using the control interconnect network.

It should be possible to determine the correct mix and type of function units automatically from a set of Rapid programs that describe the entire computation set of interest. Scott Hauck's research group at UW is focused on this problem and has generated some promising results.

Application-Specific Memory Units

Rapid implements dataflow graphs in which results produced by one function unit are forwarded directly to the function units that use them. These values are stored in registers distributed throughout the datapath instead of in a centralized register file or files. The data interconnect itself provides registers for pipelining data transfers that consume more than a clock cycle. Typically, a register is configured onto the output of each function unit, and a clock cycle consists of transferring this value to the next functional unit and performing its operation, whose result is stored in a register. If the register value is fed back to the same operator, then the register becomes used as an accumulator.

These registers provide storage for temporary values but cannot provide all the storage required by a computation. Rapid also provides memories in the datapath, which allow a larger number of values to be stored. These memories act as small explicitly managed caches, but they are almost always accessed in a simple linear pattern, which allows an efficient implementation. We propose that memories be configurable to allow use as general memories using computed addresses and values, and or as variable-length shift registers with perhaps the provision for complex addressing required by applications like Viterbi and FFT.

Rapid does not have registers files because they require too much control in the form of addresses, although nothing in the general model precludes them. There is also nothing to preclude other specialized memory units, for example, CAMs, used to streamline the execution of a particular computation.

I/O Ports

Data enters and exits a Rapid datapath via input and output ports. These ports may be connected to memory associated with the Rapid array, or directly to streaming interfaces connected to sensors or even other Rapid arrays. Ports are typically used to provide streaming access to external data. Data is consumed from an input stream via a specific port read control signal and written to an output stream via a specific port write signal. Ports connected to

memory are associated with an address generator that provides the memory read addresses for read streams and the memory write addresses for write streams. The streams also follow the model of the generic datapath unit as shown in Figure 2, with both control (read or write) and status (empty or full) signals. These status signals may be used to stall the execution of the datapath when the input stream is empty or the output stream is full.

The stream address generators are decoupled from the datapath; that is, they operate autonomously and are allowed to read ahead, for example, on an input stream, or write behind on an output stream, as far as buffering permits. Of course, the decoupling can be reduced or eliminated (almost) by reducing the buffering. The buffering of decoupled streams can be placed on the addresses going to memory, on the data returned from memory, or both. We allow a large amount of buffering in the Rapid emulator since we are using dynamic RAMs and interleaved memory modules. This decoupled memory access reduces the effect of memory latency and allows high bandwidth streaming memory interfaces to large memories constructed using DRAM technology. If the memory latency is very small, then only a small buffer for the addresses is necessary. This means that memory reads and writes are not performed until the stream read or write signal is asserted. It also reduces the problem of memory consistency.

The address generators in the current Rapid can be extracted automatically from a Rapid-C program, or they can be programmed explicitly. Although the first method is convenient at times, we have found it often conceptually simpler to separate the stream addressing from the rest of the program. In the benchmark architecture, the address generator program is loaded at the beginning of execution and cannot be changed by the controller. In the general model, the stream addressing can be initialized and changed by the controller. This allows the program to switch between different stream programs, or to modify a stream program using computed data.

The address generators are currently simple, stack-based, sequencers optimized for executing nested loops. Address generator programs can be quite complex, comprising sequences of deeply nested loops and triangular loops. The address generator produces "address packets" which specify a sequence of addresses with a constant stride. These packets are unrolled to produce the addresses. This allows the address generator design to be relatively simple. This basic design is sound, but provision should be made to allow the sequencer program to easily change the address generator program, or to load a whole new program.

A memory port can also be used in coupled mode, where the datapath provides the read or write address directly. This introduces latency, which reduces performance unless sufficient computation can be scheduled to cover it. If the latency is deterministic, then the memory can be modeled as part of the datapath and explicitly scheduled by the compiler.

Data Interconnection Network

The data interconnection network transfers data between the function units, memory units and I/O ports to implement the dataflow graphs described by a program. The goal of the compiler is to map these dataflow graphs to the Rapid array so that a large number of function units operate in parallel. While it is relatively easy to find operations that can be performed in parallel, it is much harder to manage the transfer of data so that it is available to the function units at the right time. Part of the problem is that a general interconnect structure like a crossbar, which would allow arbitrary connections, is much too expensive for the large number of function units in a reconfigurable datapath. Fortunately, most algorithms that appear in embedded applications do not require such a general network. For example, there are a large number of systolic algorithms for common computations like matrix multiply and 1D and 2D filters that require a relatively sparse interconnect. Thus a more limited, but more scalable, data interconnection network suffices.

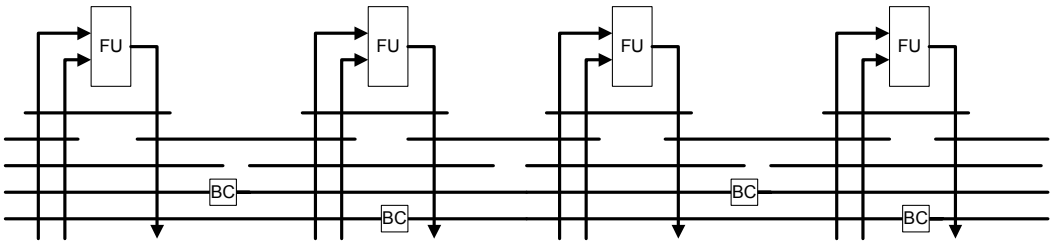

Figure 3 - Data interconnection network for Rapid

The Rapid data interconnection network comprises a set of data busses, each of which can be driven by a subset of the datapath units and read by a subset of units. As shown in Figure 3, the data busses have different lengths that can

be conveniently laid out in tracks as segmented data busses. In some tracks, the data segments can be connected via a bus connector (BC) to make a longer bus segment. The computation dataflow graph must be mapped to the reconfigurable datapath such that the data connections can be implemented using the configurable interconnect network. This mapping in the presence of interconnection constraints is one of the difficult problems that the scheduler must solve.

In the generalized model, locality continues to be an assumption to allow scalability, but the linear array is generalized into a 2-D array in which a linear interconnection scheme can be implemented, and which also allows bypass paths that take advantage of the 2-D nature of the layout. Generalizing the Rapid 1D array to a 2D array allows for shortcut paths in the array as shown in Figure 4. This introduces hierarchy into the bus structure, which can be used both in the rows and columns. That is, bus segments can be driven by other bus segments, in addition to the datapath unit outputs. These extra paths allow dataflow graphs with some long-distance connections to be implemented more efficiently, even if the datapath is pipelined linearly in the usual way.

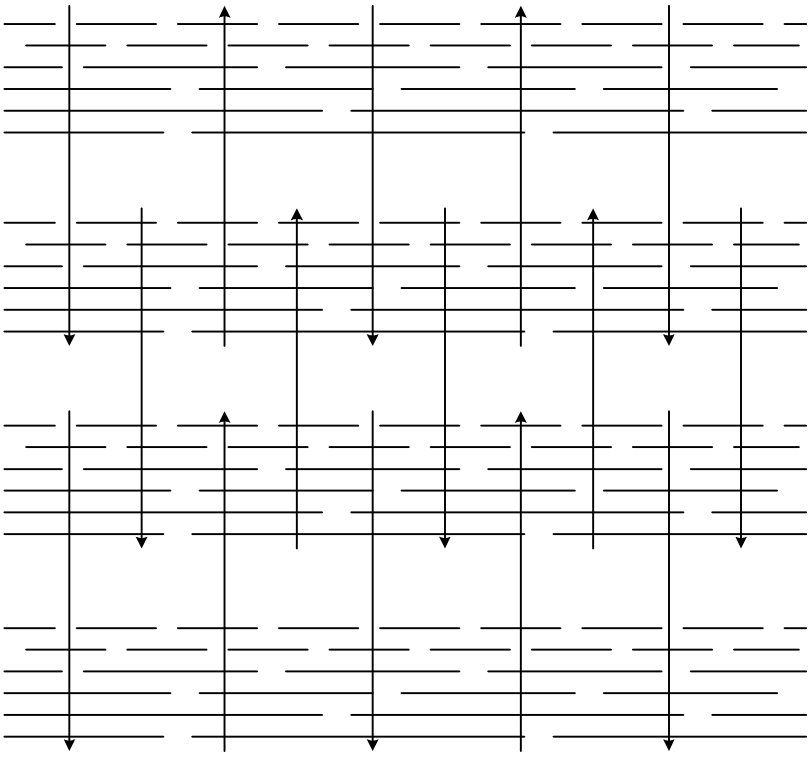

Figure 4 - Data busses in different rows can communicate via bypass busses in the vertical dimesion

The programmable connections in the datapath interconnect can be modeled as multiplexers. For example, each function unit input uses a multiplexer to select one bus from a set of busses, and each bus uses a multiplexer to select from a set of function unit outputs or other busses. Each multiplexer is controlled by a set of signals that determines which input is selected. Thus the datapath control signals comprise the multiplexer control signals in addition to the function unit, memory unit and I/O port control signals. The setting of these control signals on each clock cycle determines the operation of the datapath. Generating these control signals for a large datapath with hundreds of datapath units and thousands of control signals is potentially very expensive in area and power.

Hard and Soft Control Signals

Unlike traditional processor architectures, which generate control signals every cycle by decoding an instruction, adaptable architectures use configuration to define at least some of the control signals. Configuration is done by writing values to memory locations that drive control signals directly. Configurable control is cheap and uses almost no power since only one bit is required for each control signal, and that bit is changed only during reconfiguration, which happens infrequently. However, since configurable control signals cannot be changed while a

program is running, the datapath is forced to execute the same computation on every cycle, which is too constrained to be very useful.

FPGAs use configurable control, and implement dynamic control by configuring the appropriate control circuitry to generate it. In this case, the configurable control is meta-control, and used to generate the actual dynamic control. Unlike FPGAs, coarse-grained configurable architectures separate circuits used for control from those used for computation. This leads to a specialized control architecture for generating dynamic control. (This view of configuration bits as control signals that change infrequently is one that was suggested by the Andre DeHon, but still not accepted by many FPGA types. But it is a useful model for discussing control generation in configurable architectures.)

There have been proposals for time-multiplexed FPGAs that have several sets of configurations that can be changed very quickly. This is orthogonal to the current control discussion – any configurable control could be provided as one of a small set of contexts to allow more rapid switching from one datapath configuration to another.

The Rapid architecture uses a combination of statically configured (*hard*) and dynamically driven (*soft*) control. Dynamic control can be changed every cycle, but it is very expensive since it requires fetching and delivering the control value every clock cycle. As a rule of thumb, a soft control signal costs between 100 and 1000 times as much as a hard control signal. Which control signals are hard and which are soft is an architectural decision, but one that affects the compiler since a program must be mapped to meet the hard control constraints.

The Rapid architecture allows the optimization of soft control signals by allowing them to be *soft-configured* to a constant value just like hard control. This takes advantage of the fact that only a relatively small number of the potentially dynamic control signals actually change during any one computation. Dynamic signals that are soft-configured do not use any instruction memory and are not driven during an application, which saves both area and power. The compiler can increase the utility of this optimization by mapping programs to minimize the number of control bits that change during the program execution.

Control Generation

The control flow of a program is executed by a sequencer, which generates an instruction on each clock cycle that controls the execution of the datapath. The sequencer is implemented as a simple program counter that can execute branches, loops and simple subroutine calls efficiently, along with an instruction memory. Each instruction is executed either by the Rapid datapath, or by a simple RISC datapath, which executes control flow instructions and "random" arithmetic operations associated with the control flow. Most applications that run on the Rapid array execute few, if any, RISC instructions.

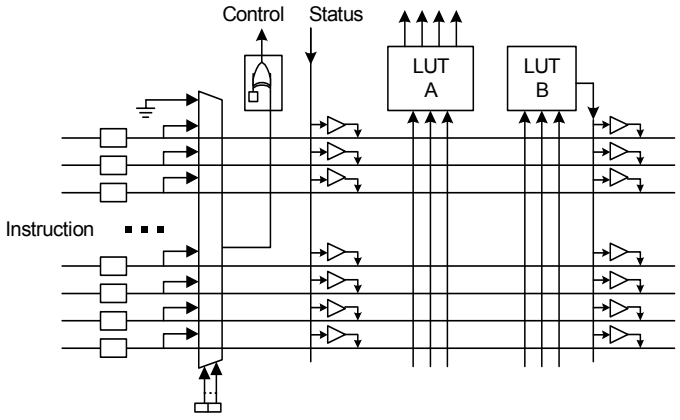

Figure 5 - Control signals are generated by connecting them to the appropriate instruction bit.

The datapath instruction produced by the sequencer is "decoded" using the configurable control network shown in Figure 5 to generate the dynamic control signals. Most control signals are driven directly by selected bits in the instruction. The instruction bit is selected independently for each control signal using configuration data, which acts like meta-control. Thus if two or more control signals take on the same values during the computation, they can be driven by the same instruction bit. This occurs often since there is substantial repetition of control in the execution

of a highly parallel datapath. For example, when a computation phase begins, many datapath elements may need to be initialized and one control signal can often be used to perform this initialization. This effect is magnified if the control generation is pipelined at the same rate that the datapath is pipelined. Some control signals are driven indirectly using lookup tables (LUTs) that allow instruction bit values to be remapped to the values used to drive the datapath. This greatly increases the sharing across control signals. A combination of datapath scheduling and control signal optimization can reduce by at least an order of magnitude the number of instruction bits needed to drive the datapath control signals.

The control architecture defines the number of instruction bits, and the mapping must fit within this constraint. If fewer instruction bits are used, then instruction bits that are not used by an application need not be read from memory or driven. That is, some fairly large number of instruction bits can be provided, only some of which are used by most applications. The extra cost for a longer instruction is the area for the extra instruction memory, wires and control multiplexers.

The LUTs are also used to incorporate status signals into the control functions to enable data-dependent computation. Each status bit can be routed to a LUT and combined with other control signals to produce a data-dependent control signal. For example, a conditional add/subtract operation can be performed based on the sign of a number to implement absolute value or the sum over absolute differences. These LUTs can also implement simple control FSMs that can substantially reduce the complexity of the sequencer program. For example, an FSM can implement a Boolean flag or parity bit that would otherwise introduce control flow into the sequencer program.

The Rapid benchmark architecture uses pipelined control, where the instruction bits are pipelined alongside the datapath. This has the advantage of skewing the control signals along with the pipelined datapath so that more sharing takes place, but has the disadvantage that the datapath has an implicit directionality. That is, the computation tends to run "downhill" from one end of the datapath to the other. While this works well for many pipelined computations, there are others that can take advantage of pipelining in both directions. The control network in the general Rapid architecture allows a limited amount of control pipelining, so that control signals can be offset in time with respect to each other by some constrained amount. This offset may be different for different control signals in the datapath. For example, the range could be less at one end than the other, reflecting the "downhill" bias of the datapath.

There may be by-pass control wires from one row in the array to an adjacent row is the style of the data bypass busses shown in Figure 4, but this would be only for the extra control wires.

The Sequencer and RISC Datapath

The Rapid sequencer produces the datapath instructions, which are decoded as described above. The sequencer executes the control flow part of the control/dataflow graph representation of the program. In the simple case, the program comprises nested loops with some conditional branches and perhaps subroutine calls. Some programs, however, have a complex structure that is more naturally described using multiple, parallel threads. If the nested loops in the different threads do not match up naturally, then the program becomes very complex. This program is really the interleaving of several state machines and in the worst case, becomes the cross product of the constituent state machines with a size that is the product of their sizes. However, if each thread can be run on a different controller and the instruction streams merged, this complexity be avoided. The disadvantage to this scheme is that there are four instruction memories instead of just one.

Rapid-C allows the programmer to write threads that execute in parallel. The synchronization between threads is done via signal and wait statements. When threads are running, however, they run in lock-step so that shared communication and resources can be statically scheduled. Each parallel thread is then mapped to a different controller.

Conceptually, the sequencer can be viewed as a single controller, producing a single instruction, regardless of the number of different controllers there are. This instruction is either a datapath instruction, or a RISC instruction that controls a simple RISC datapath. This RISC datapath is used to execute control-flow related code or code that does not fit well to the Rapid datapath and would complicate the datapath instructions. The Rapid datapath instructions are also produced as "packets" of repeated instructions. That is, if an instruction is repeated, as it might be in an inner loop, an instruction count is used to replicate the instruction. This mechanism is used to cover overhead in the sequencer; that is, it can overlap the execution of RISC instructions with Rapid datapath instructions. This

technique may also reduce power consumption since the controller can be stalled while idle. Stalls can be implemented either by issuing NOPs, that is turning off all register writes, or by turning off the clock.

The instructions generated by the sequencer are either RISC instructions or Rapid instructions. The Rapid instructions contain control signals only for the Rapid datapath, and all sequencing is performed by RISC instructions. These RISC instructions including looping, branching and procedure call instructions. Loops are executed in the same way as the Rapid benchmark architecture: a start loop RISC instruction specifies the loop count and end of loop PC. The loop is then executed without any sequencing instructions by the hardware, which executes the end of loop control autonomously. A loop stack allows nested loops to be executed. An instruction repeat count may also be used to implement very small loops and achieve some overlap between Rapid and RISC code execution.

The Rapid datapath and the RISC operate independently but communicate via data and status FIFOs. Two FIFOs communicate data between the RISC and the datapath, and a third FIFO communicates status values from the datapath to the controller. This allows the controller to execute control flow based on results from the datapath. This feature is not used by many programs, but is necessary for some, and reduces the complexity of others. Branch instructions use conditions generated in the RISC datapath or in the Rapid datapath. The RISC ALU generates the usual condition codes, which can be used for branching. Status generated by the Rapid datapath is communicated via the status FIFO. If this FIFO is empty, then the sequencer must stall until a status value becomes available. It is assumed that there sufficient datapath instructions in the pipeline to generate the status signal. This mechanism avoids having to know exactly the pipeline latency of the computation and allows several status values to be computed and queued to reduce the effective latency. In a typical computation that uses status to break out of a loop for example, a new loop iteration may be started based on the result of the result from one or two iterations in the past. If the control flow is more tightly coupled, then the controller or datapath may stall. Of course, using status produced by a deeply pipelined computation introduces possible deadlock where the datapath can be stalled for instructions while the controller is stalled for status results from the datapath. This must be solved by the compiler, which calculates the maximum latency in the datapath and separates the generate and test instructions by at least this amount, incorporating datapath NOPs if necessary.

Procedure calls are executed by pushing the address of the next instruction on the loop stack. However, there is no stack for the RISC registers. Procedure calls are thus a very simple mechanism to allow code to be shared, but behaves more like inlining. Other control instructions could also be included, such as indirect jumps and multiway branching.

The RISC datapath comprises an ALU and a register file. The size of the register file and the operations supported by the ALU are implementation dependent. A reserved register in the register file is used to interface to the data FIFOs. Writes to this register stores a data item in the FIFO to the Rapid datapath, while a read to this register retrieves the next data value sent from the Rapid datapath. This RISC datapath also interfaces directly to memory: loads and stores are performed directly to memory with no caching. Indexed and indirect addressing is available to allow arbitrary address computations.

The configuration memory of the Rapid datapath and address generators can be mapped into this address space. This allows the RISC instructions to perform reconfiguration directly from memory. The controller program memory can even be mapped into this address space, which allows the controller to load new programs. In this case, a small boot kernel would be placed in ROM that would allow the first program to be loaded.

The instruction width for the RISC and Rapid datapaths should be the same size so that no instruction bits are wasted. It may be possible to keep the instruction size to 16 bits, although 32 bits is a more reasonable goal.

Normally, the controller generates instructions for the Rapid datapath; however, it can execute instructions and even whole programs on the RISC datapath to execute control-dominated, or serial code that is better suited to a RISC. The RISC is not meant to be particularly high-performance, because only a small percent of the instruction cycles will be executed by the RISC. The RISC and Rapid datapath can exchange data, which allows the scalar datapath to use data generated by the Rapid datapath, and allows the RISC datapath to insert data in the Rapid datapath. The RISC datapath also controls the configuration of the Rapid datapath. That is, code running on the RISC processor can directly configure the address generators and streams for the Rapid datapath, as well as write configuration data to the Rapid datapath. Thus, the Rapid datapath can reconfigure itself by fetching a new configuration from memory.

VLIWs and Coarse-Grained Configurable Architectures

Coarse-grained configurable architectures can be seen as very large VLIW processors, with control that is a mixture of configurable and traditional dynamic control. The optimization of control is key, since for a very large datapath, it can consume a large part of the area and power. There is, however, a continuum between traditional VLIW and large-scale coarse-grained configurable architectures that can be implemented in the Rapid-style architecture. A VLIW falls into the Rapid architecture definition if the datapath is constructed using a central, multi-ported register file with sufficient connections between function units, and if all the control is defined dynamically. Thus, part of a Rapid datapath can be make to look like a VLIW, while the rest is implemented using the more general Rapid architecture. This way, programs that do not map well to the configurable datapath can be executed using the less constrained VLIW architecture, but the entire datapath can be used, including the VLIW part, for computations that do map well to a large-scale datapath. This does present some new alternatives for control. The instruction could be divided into control for the VLIW and control for the rest of the datapath, the control could be shared and configured according to the normal control optimization, or perhaps some form of the two could be used.

This use of a VLIW-like architecture for part of the datapath is separate from the RISC datapath used by the sequencer. An alternative is to use a more heavy-duty RISC processor, perhaps even a VLIW, and divide execution between the two parts. This would yield a relatively closely coupled system, but the RISC/VLIW would not run in parallel with the Rapid datapath, except as noted above. If a more loosely coupled system is desired, then a RISC/VLIW should be coupled to the Rapid datapath via streams and run asynchronously. Each would be scheduled separately and controlled by a different sequencer.

References

1. D. Cronquist, P. Franklin, S. Berg, and C. Ebeling. Specifying and Compiling Applications for RaPiD, In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, April 1998.

2. D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths, In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, Atlanta, pp. 23-40 , April 1999.

3. C. Ebeling, L. McMurchie, S. Hauck, and S. Burns. Placement and Routing Tools for the Triptych FPGA, *IEEE Transactions on VLSI Systems*, Vol. 3, No. 4, pp. 473-482, December 1995.

4. C. Ebeling, D. Cronquist, P. Franklin, J. Secosky, and S. Berg. Mapping Applications to the RaPiD Configurable Architecture, In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*, April 1997.

5. C. Ebeling and L. McMurchie. Pathfinder: A Negotiation-Based Router for Routing-Constrained FPGAs, In *Proceedings of the Third ACM Symposium on Field-Programmable Gate Arrays*, Monterey, pp. 111-117, February 1995.

6. C. Fisher, K. Rennie, G. Xing, S. Berg, K. Bolding, J. Naegle, D. Parshall, D. Portnov, A. Sulejmanpasic, and C. Ebeling. "An Emulator for Exploring RaPiD Configurable Computing Architectures," In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications* (FPL 2001), Belfast, pp. 17-26, August, 2001.