Configuration Steering for a Reconfigurable Superscalar Processor

Brian F. Veale
School of Computer Science
University of Oklahoma
veale@ou.edu

John K. Antonio
School of Computer Science
University of Oklahoma
antonio@ou.edu

Monte P. Tull
School of Electrical and
Computer Engineering
University of Oklahoma
tull@ou.edu

Abstract

An architecture for a reconfigurable superscalar processor is described in which some of its execution units are implemented in reconfigurable hardware. The overall configuration of the processor is defined according to how its reconfigurable execution units are configured. An efficient micro-architectural solution to configuration management is presented that effectively steers the current processor configuration toward a configuration that is well matched with the execution unit requirements of instructions being scheduled for execution. The approach first selects the best matched among four steering configurations based on the number and type of execution units required by the instructions. One of the steering configurations is dynamically defined as the current configuration; the other three are statically predefined. Once a steering configuration is selected, portions of it begin loading on corresponding reconfigurable execution units that are not busy. The active configuration of the processor is generally the overlap of two or more steering configurations.

1. Introduction and Related Work

In contrast to a static processor, the architecture of the hardware and/or the instructions supported by a reconfigurable processor can be changed dynamically. This means that the type and quantity of circuitry implementing particular instructions, or functionality, can be changed after fabrication of the processor and even during execution. A main objective of this work is to increase the achieved instruction level parallelism of the processor by best matching the processor configuration to the instructions that are ready to be executed. The particular focus of the paper is on the design of a configuration manager for a reconfigurable superscalar processor.

There are three main paradigms for the design of reconfigurable processors; these paradigms are based on how the reconfigurable logic of the processor is interfaced with other components of the architecture [1]. The three paradigms are: (1) attached processor, (2) co-processor, and (3) functional unit. In the attached processor paradigm, the reconfigurable logic is connected to a host processor via an I/O bus (e.g., a PCI or OPB bus). A host processor controls the operation of the reconfigurable logic via the bus; and/or data is transmitted between the reconfigurable logic and the host processor using the bus [1]. An example of a system that uses the attached processor approach is PipeRench [2].

The co-processor paradigm attaches the reconfigurable logic directly to the host processor in a fashion similar to a floating-point co-processor [1]. One example of this approach is Garp [3].

The final paradigm, the functional unit approach, integrates the reconfigurable logic into the processor as a functional unit; reconfigurable function units are referred to as RFUs in [1]. OneChip98, SPYDER, and PRISC are examples of the RFU paradigm [1, 4, 5]. The architecture considered in this paper is in the RFU paradigm. An advantage of this paradigm is that it closely models the design of a traditional processor and many existing design concepts can be applied to such a processor.

Examples of previous work in the area of applying reconfigurable architectures to general-purpose computing requirements are SPYDER [4] and PRISC [5]. SPYDER uses a single RFU to implement hardware synthesized specifically for a program to be executed on the processor [4]. A C++ to netlist (a hardware description code) compiler that creates the binary configuration code used to configure the RFUs must be run before a program can be executed on SPYDER [6]. Thus, SPYDER requires that source code must be available and recompiled.

PRISC [5] is a reconfigurable processor similar in concept to SPYDER. A main difference between the two is that the reconfigurable resources in PRISC consist of multiple RFUs connected to the data path of the CPU along with static functional units [5]; SPYDER does not specify static functional units. For programs to utilize the reconfigurable resources of PRISC, they must be analyzed by a hardware extraction tool that determines what program code should be executed using the reconfigurable resources [5].

The SPYDER and PRISC processors represent an important step in applying reconfigurable computing to the realm of general-purpose computing; however, they may lack mainstream viability because they are not legacy-compatible at the level of binary code. Consider the vast amount of legacy software and hardware systems that dominate today's market.

Our motivation is to study general-purpose reconfigurable processors that can execute machine code compiled for current or legacy architectures. Research in this direction has already been undertaken in [7], where a general-purpose reconfigurable processor is proposed and modeled. The architecture introduced in [7] is based on a set of predefined configurable modules, each of which defines a different configuration of the functional units available in the architecture. These modules can be dynamically loaded at run-time to best match the needs of the instructions currently being executed by the processor. In order for such an approach to work efficiently, the configuration manager portion of the processor must be able to quickly determine the best configuration at any point in time based on the signature of the instructions in the instruction queue that are ready to be executed.

The work presented in this paper proposes a fast and efficient configuration selection circuit that performs the task assigned to the configuration manager in the architecture proposed in [7]. An overview of the architecture defined in [7] and the modifications and additions that our work assumes are presented and analyzed in Section 2. One aspect of the approach proposed here is that it uses a set of predefined modules, similar to those proposed in [7], and melds them into configurations of the functional units that best match the needs of the system at any given time using partial reconfiguration. The techniques presented in this paper could be applied to other architectures in addition to the architecture of [7] and its modified version proposed here.

2. Overview of the Architecture

Figure 1 shows the partially run-time reconfigurable architecture considered in this paper. This architecture is derived from the architecture introduced in [7]. Because some of the functional units of the processor are reconfigurable, the architecture is within the RFU paradigm discussed in the previous section. A collection of five fixed functional units (FFUs) and eight RFU "slots" are provided as a basis for the architecture in this paper. As the processor executes instructions, it reconfigures RFUs that are not busy to best match the needs of the instructions that are in the instruction queue and are ready to be executed.

The architecture given in Figure 1 includes three predefined configurations for the functional units. The RFUs can be reconfigured independently of each other using partial reconfiguration techniques, thereby allowing the processor to implement a configuration that is a hybrid combination of the predefined configurations. Thus, the current configuration may or may not correspond exactly to one of the predefined configurations. Predefined configurations provide a basis for selecting a steering vector for the reconfiguration.

This approach is an extension of [7], where the use of partial reconfiguration at the level of functional units was not directly addressed. Also, the idea of implementing one of each type of functional unit in fixed hardware was not specified. However, the basic architectural structure assumed in this paper is the same as that assumed in [7].

Each predefined configuration specifies zero or more integer arithmetic/logic units (Int-ALU), integer multiply/divide units (Int-MDU), load/store units (LSU), floating-point arithmetic/logic units (FP-ALU), and/or floating-point multiply/divide units (FP-MDU). Table 1 is a break down of how many functional units of each type are provided by each configuration including the number of each that is provided as a fixed unit. It should be noted that the granularity of the functional units can be generalized to be either finer or coarser than what is assumed here. For the purposes of this work, it is assumed that each instruction is only supported by by one type of functional unit.

In addition to the fixed functional units, other fixed modules of the architecture provide separate instruction and data memories, a trace cache, an instruction fetch unit, an instruction decoder, a register update unit, a register file, and the configuration management unit. The instruction fetch unit fetches instructions from memory and provides them to the configuration management unit, which uses a unit decoder similar to the pre-decoder of [7] to retrieve the instruction opcodes. The instruction opcodes are then used to determine the functional unit resources required. The trace cache is used to hold instructions that are frequently executed. In [7], the trace cache and the pre-decoding unit are used to determine the resources required to execute instructions at run time. Section 3 provides a configuration selection system that matches instructions that are ready to be executed with the functional units they require and (partially) reconfigures the functional units of the processor to match the needs of these instructions. This system can be used (to fulfill the requirements) for the pre-decoders and configuration manager envisioned in [7].

The register update unit collects decoded instructions from the instruction queue and dispatches them to the various functional units configured in the processor. This unit also resolves all dependencies that occur between instructions and registers. A dependency buffer is included in the register update unit that keeps track of the dependencies between instructions and registers. This unit writes computation results back to the register file during the write-back stage of instruction execution. Furthermore, the register update unit allows the processor to perform out-of-order execution of instructions, in-order completion of instructions, and operand forwarding [7].

3. Configuration Selection and Loading

3.1. Configuration Selection

The configuration selection unit is shown in Figure 2. This unit inspects the instructions in the instruction queue that are ready to be executed and chooses one of the four configurations of functional units specified in the architecture. Three of these are the predefined steering configurations; the remaining represents the currently active configuration (see Table 1). The current configuration may or may not correspond exactly to one of the predefined steering configurations because partial reconfiguration is employed when transitioning between configurations. Thus, the current configuration may be a hybrid combination of two or more predefined steering configurations. The configuration selection unit considers the possibility that the current configuration may be better matched to the instructions requesting resources than any of the predefined steering configurations. In fact, achieving a stable and well-matched current configuration is desirable because it implies that the architecture has settled into a configuration state that matches the requirements of the code.

The configuration selection unit consists of four stages: (1) the unit decoders, (2) resource requirements encoders, (3) configuration error metric generators, and (4) a minimal error selection unit. The inputs to the selection unit are the instruction queue and the number of each type of functional units currently configured in the processor. The output of the unit is a two-bit value that indicates which of the four configurations (three predefined RFU configurations or the current configuration) should be configured next.

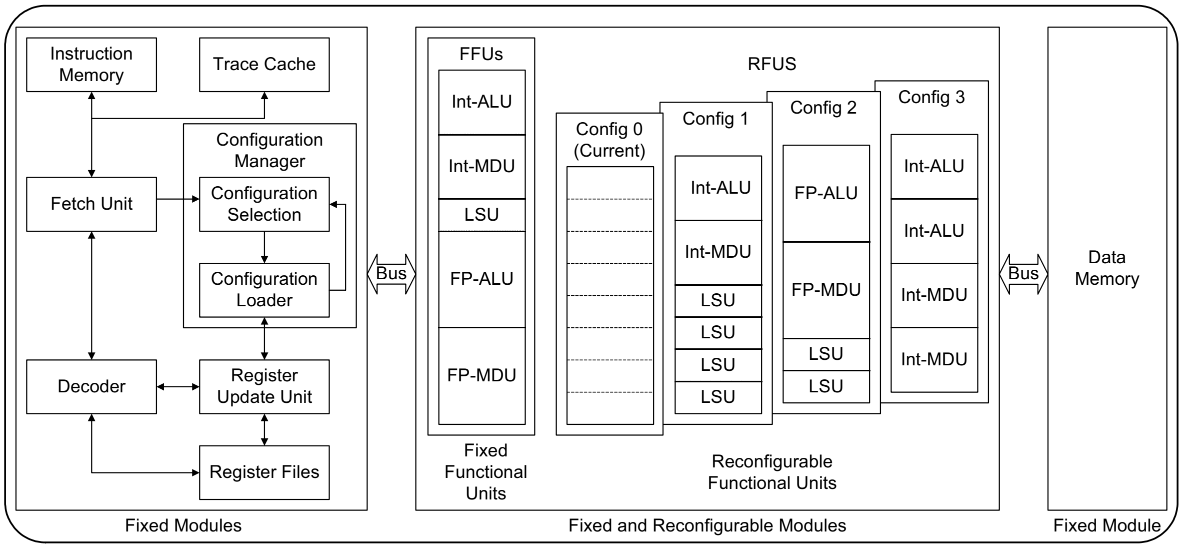


Figure 1. A partially run-time reconfigurable architecture, derived from [7].

Table 1. Number of each type of functional unit provided in the fixed and reconfigurable portions of the processor, and their encodings.

	Int-ALU	Int-MDU	LSU	FP-ALU	FP-MDU
FFUs	1	1	1	1	1
RFUs – Configuration 0 (Current)	0 - 2	0 - 3	0 - 4	0 - 1	0 - 1
RFUs – Configuration 1	1	1	4	0	0
RFUs – Configuration 2	0	0	2	1	1
RFUs – Configuration 3	2	2	0	0	0
Resource Type Encoding, *t*	000_2	001_2	010_2	011_2	100_2

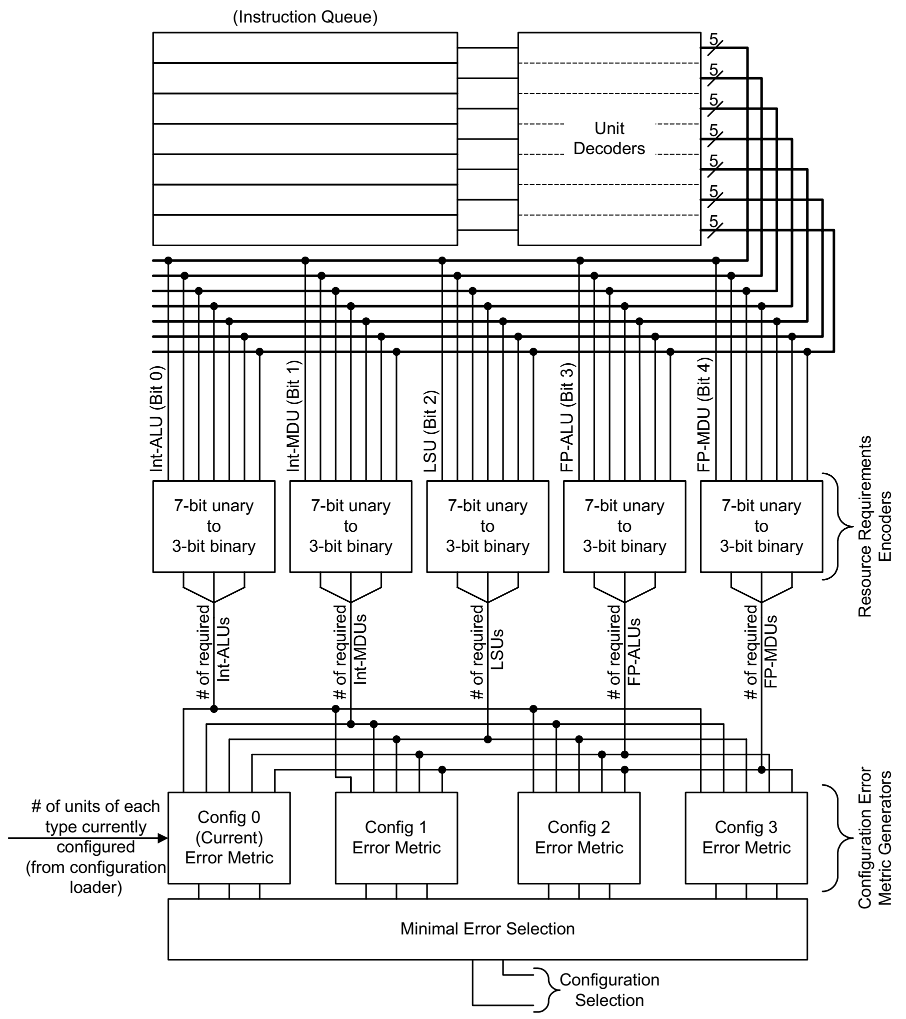


Figure 2. Configuration selection unit.

The unit decoders serve the same purpose as the pre-decoders of the original architecture specified in [7]. These decoders retrieve the opcode of each instruction in the instruction queue that is ready for execution. The output of each unit decoder is a one-hot vector that indicates the functional unit required by the instruction whose opcode the unit decoded. This information is collected from all decoders and transformed into a three-bit binary value by the resource requirements encoder that indicates how many functional units of each type (Int-ALU, Int-MDU, LSU, FP-ALU, and FP-MDU) are required to execute all of the instructions in the instruction queue. The configuration error metric generators then determine how close each of the three predefined configurations and the current configuration are to providing the resources required by the instructions in the instruction queue. Finally, the minimal error selection unit uses the error of each configuration to choose the configuration that most closely meets the needs of the instructions in the instruction queue.

The configuration error metric generators calculate a value that indicates the “closeness” of the number and type of functional units required to execute the instructions in the instruction queue relative to each of the four configurations including the FFUs. The function that each error metric unit implements is defined by the equation given in Figure 3(a).

The configuration error metric (CEM) circuit of Figure 3(b) accepts the quantified configuration resources for the three predefined configurations, as well as the current configuration. The CEM circuit shown in Figure 3(b) implements the equation of Figure 3(a) to produce the error metric for each of the four configurations. The barrel shifters for the three predefined configurations can be arranged with hard-wired shift control inputs to divide by 4, 2, or 1. The shifters for the current configuration use shift control inputs that represent the upper two bits of the quantity of currently configured functional units. Figure 3(c) shows how the upper two bits are treated to approximate division of the functional unit requirement using 4, 2, or 1 as the divisor. A more accurate divider circuit could be implemented, if desired, at the expense of increased complexity and latency. Because the total number of functional units required cannot exceed seven (the instruction queue is assumed to hold seven instructions), three-bit adders are sufficient for summing the total error metric.

The configuration selection unit chooses a configuration that achieves a minimal error by outputting a two-bit binary value that represents the configuration that should begin loading. The novelty in this process is handling the case where an RFU is executing a multi-cycle instruction, which is accomplished by only reconfiguring the RFUs that are not busy.

In cases where the configuration errors are equal, the minimal error selection circuit is designed to identify the configuration that requires the least amount of reconfiguration. Thus, the current configuration is always favored over any predefined steering configuration that has the same error metric value.

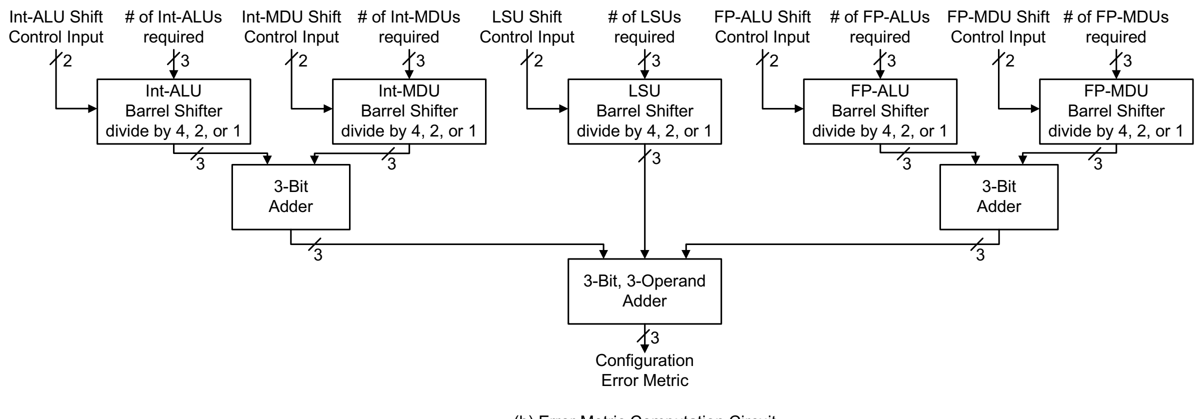
3.2. Configuration Loading

The configuration selection unit of Figure 2 determines the configuration that should be loaded into the processor to execute the instructions in the instruction queue that have not been scheduled. If the configuration selection unit chooses the current configuration, then the system will not reconfigure any of the RFUs. Additionally, the configuration loader tracks what type of functional unit is configured into each slot of reconfigurable logic. This is handled by storing a resource allocation vector that contains this information. Each of the functional unit types supported by the architecture (Int-ALU, Int-MDU, LSU, FP-ALU, FPU-MDU) are given a three-bit encoding, specified in Table 1. Because each functional unit can occupy one or more slots of reconfigurable logic available in the processor, a special encoding is used to indicate that a slot contains a portion of a functional unit that spans two or more slots. The first entry of the resource allocation vector for a unit that spans multiple slots contains that block's encoding, and the following entries contain the special encoding of 111_2.

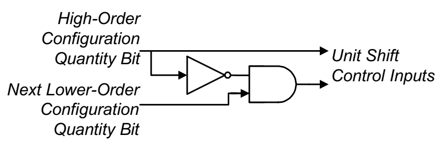
Once a configuration is chosen, the configuration loader will determine which RFUs need to be reconfigured by determining the difference (XOR) between the chosen configuration and the current configuration using the resource allocation vector. The loader will then choose which RFUs to reconfigure on the basis of their availability. If an RFU is executing a multi-cycle instruction, the RFU cannot be reconfigured until the instruction finishes execution and is retired. (And by the time it is available for reconfiguration, a different configuration may have been selected.) To accommodate this approach, each slot has an available port that is asserted when the unit it implements is available. The configuration loader can determine if an RFU can be reconfigured by inspecting this output from the corresponding slot.

$$Error = \left\lfloor \frac{Req'd\ \#\ of\ Int\text{-}ALUs}{Avail\#\ of\ Int\text{-}ALUs} \right\rfloor + \left\lfloor \frac{Req'd\ \#\ of\ Int\text{-}MDUs}{Avail\#\ of\ Int\text{-}MDUs} \right\rfloor + \left\lfloor \frac{Req'd\ \#\ of\ LSUs}{Avail\#\ of\ LSUs} \right\rfloor + \left\lfloor \frac{Req'd\ \#\ of\ FP\text{-}ALUs}{Avail\#\ of\ FP\text{-}ALUs} \right\rfloor + \left\lfloor \frac{Req'd\ \#\ of\ FP\text{-}MDUs}{Avail\#\ of\ FP\text{-}MDUs} \right\rfloor$$

(a) Error Metric Equation


Int-ALU Shift Control Input
# of Int-ALUs required
Int-MDU Shift Control Input
# of Int-MDUs required
LSU Shift Control Input
# of LSUs required
FP-ALU Shift Control Input
# of FP-ALUs required
FP-MDU Shift Control Input
# of FP-MDUs required
2
3
Int-ALU Barrel Shifter divide by 4, 2, or 1
Int-MDU Barrel Shifter divide by 4, 2, or 1
LSU Barrel Shifter divide by 4, 2, or 1
FP-ALU Barrel Shifter divide by 4, 2, or 1
FP-MDU Barrel Shifter divide by 4, 2, or 1
3-Bit Adder
3-Bit Adder
3-Bit, 3-Operand Adder
Configuration Error Metric

(b) Error Metric Computation Circuit


High-Order Configuration Quantity Bit
Next Lower-Order Configuration Quantity Bit
Unit Shift Control Inputs

(c) Current Configuration Shifter Inputs

Figure 3. Configuration error metric generation.

If the unit is available and it must be reconfigured to implement a new configuration, then the configuration loader will reconfigure the RFU to implement the functional unit specified by the chosen configuration. The RFU will not be reconfigured if it already implements the specified functional unit (i.e., the type of the unit currently implemented in the RFU matches the type specified in the chosen configuration). This reconfiguration is performed using partial reconfiguration techniques, such as those discussed in [8].

Due to the possibility that some RFU's may not be reconfigured to implement a functional unit defined by the chosen configuration, certain instructions may not be able to execute for several cycles. This problem would be compounded if FFUs were not provided as a part of the architecture and the processor entered a state where certain functional units were not implemented for long periods of time. Because the FFUs implement units for all instructions, every instruction is guaranteed to execute.

4. Instruction Scheduling and Execution

An integral challenge in the design of a dynamically partial reconfigurable processor is the scheduling, execution, and retirement of instructions. As the processor changes the configuration of its RFUs to best match the instructions being executed, the processor must be able to determine what resources are available to support the execution of instructions. If the processor chooses to schedule instructions for which there are not enough resources, then those instructions' execution can be delayed waiting for the required resources to become available.

To solve this problem, we employ a scheduling approach that uses a wake-up array that allows instructions to "wake up" when the necessary functional units are available and required results from previous instructions are available [9]. This section discusses the basic approach and presents how the availability of RFUs can be dynamically determined. Note that [9] presents a more sophisticated scheduling approach than discussed here; however, our approach can be extended using the same techniques that are employed in [9].

4.1. Scheduling using Wake-Up Arrays

A wake-up array contains information that allows the scheduling logic to match the functional units that are not busy to instructions that are ready to execute. This includes determining if the instruction requires results from any previous instructions and verifying that the results from those previous instructions are available. Specifically, the wake-up array consists of a set of resource vectors that encode which functional unit an instruction requires and the instructions that must produce results before the instruction can be executed [9]. An example of a dependency graph for a set of instructions and the corresponding wake-up array are presented in Figures 4 and 5. Note that there must be a "result required from" column in the array for each row (instruction entry) of the array.

In the example of Figures 4 and 5, the Load instruction (Entry 5) only requires a load-store unit, so only the resource bit for the LSU is set on the row for the Load instruction. Additionally, the Load instruction does not depend on the result of any other instructions, so the column entries for the other instructions in the array are not set. Recall that for the RISC architecture assumed here, an instruction will never require more than one functional unit. The Multiply instruction (Entry 4) uses an integer multiplier (Int-MDU) and requires a result from the Subtract instruction (Entry 3); therefore, the bits for Entry 4 are set in the columns for the Int-MDU unit and Entry 3.

Figure 6 shows the logic associated with the wake-up array of Figure 5 that determines if the instruction represented by each entry of the wake-up array should be considered for release by the scheduling logic. The wake-up logic only determines when an instruction is ready for execution and generates an execution request for those instructions that are ready and does not actually determine if an instruction is scheduled because multiple instructions could require the same resources. This contention between instructions must be handled by the scheduler after multiple instructions that use the same resources request execution.

The "available" lines shown in Figure 6 indicate whether the corresponding resource or the results of the corresponding entry in the array are available; the value of each line is high if the resource/result is available. These lines pass through every entry in the array and enter an OR gate that checks if the resource/result is needed and available [9]. If the resource is not required, then the output of the OR gate must be high in order for the entry to be scheduled when the resources/entries that are required are available. Each of these results are ANDed together to ensure that every resource and entry required is available [9]. The logic required to compute resource availability in a static processor is more straightforward than for a reconfigurable processor, where the logic that determines the availability of a resource must consider not only if the resource is busy but also if the resource is currently configured into the system. This logic is discussed in Subsection 4.2.

The scheduled bit, shown in Figure 6, is required to keep an instruction from requesting execution once it has been scheduled, since instructions may take several cycles to complete [9]. Instruction entries in the wake-up array are not removed until the instruction is retired to keep instructions that rely on the result(s) of the instructions currently being executed from requesting execution too early. After an instruction receives an execution grant, its corresponding available line is asserted at the time that its result will be available. This is handled using a count down timer that is set to the latency of the instruction. If the instruction has a latency of N cycles, the count down timer will be set to $N - 1$; if the instruction has a one-cycle latency, the available line is asserted immediately. An instruction's timer will start once the instruction receives an instruction grant and the instruction's available line is asserted once the time reaches a count of one. Once an instruction finishes execution and is retired, every wake-up array entry associated with the instruction is cleared to keep new instructions that are added to the wake-up array from incorrectly becoming dependent on the retired instruction. This approach also handles the case of an instruction being removed from the array before its dependent instructions are scheduled by allowing these instructions to request execution without considering a dependence on the retired instruction. If an instruction must be rescheduled, then the schedule bit is de-asserted using the reschedule input of the scheduled bit [9].

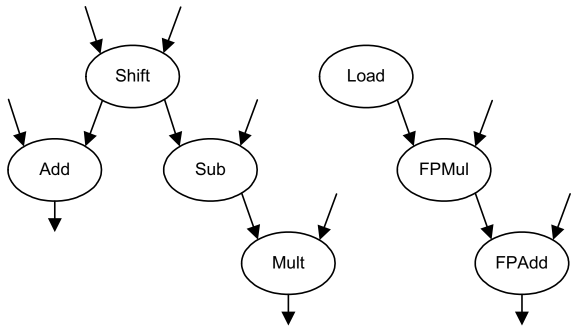


Figure 4. A dependency graph showing the dependencies between entries of the instruction queue, derived from [9].

	Execution Unit Required					Result Required From						
	Int-ALU	Int-MDU	LSU	FP-ALU	FP-MDU	Entry 1	Entry 2	Entry 3	Entry 4	Entry 5	Entry 6	Entry 7
(Shift) Entry 1	1											
(Sub) Entry 2	1					1						
(Add) Entry 3	1					1						
(Mul) Entry 4		1					1					
(Load) Entry 5			1									
(FPMul) Entry 6					1					1		
(FPAdd) Entry 7				1							1	

Figure 5. A wake-up array showing the entries for the instructions of Figure 4, derived from [9].

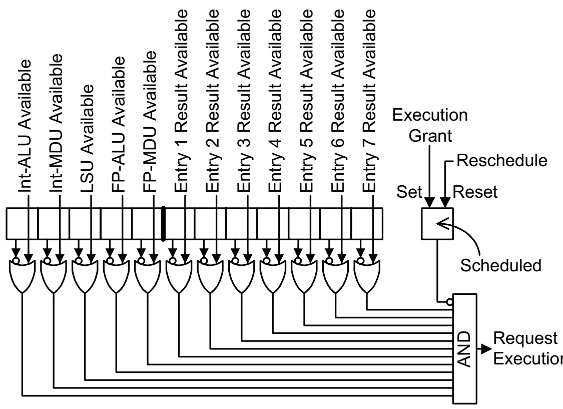


Figure 6. The logic associated with one resource vector of the wake-up array of Figure 5, derived from [9].

4.2. Computation of Resource Availability

In order to use the wake-up array approach to scheduling instructions, the processor must include logic that determines which functional units (resources in the wake-up array) are available. This can be handled by allowing each resource to assert whether it is available. If there are multiple resources of the same type, then their availability assertions must be ORed to ensure that the availability line in the wake-up logic for the resource is asserted. Determining if a resource is available is more difficult in a reconfigurable processor because of the dynamic nature of which resources can be configured into the processor at any given point in time.

The availability of a resource is a function of the allocation of the resource and availability of each copy of the resource that is configured into the processor. The availability of each resource can be determined using a signal from each slot of reconfigurable logic that indicates if the functional unit it implements is busy or available. This availability signal is asserted when the functional unit is available. Equation 1 defines the calculation of an available function that determines if a functional unit of a particular type is available using the availability signal of each slot and the resource allocation vector provided by the configuration loader that specifies the type of functional unit implemented by each RFU and FFU provided in the processor. In Equation 1, *type(i)* refers to the encoding of a functional unit of type *t*, specified in Table 1.

$$available(t) = \sum_{\substack{i \in \text{resource} \\ \text{allocation vector}}} \left(\prod_{b \in [0,2]} \overline{(type(t)_b \oplus type(i)_b)} \right) \bullet availability(i) \quad (1)$$

Some functional units require more than one reconfigurable slot. From Figure 1, we assume that LSUs require one slot, Int units require two slots each, and each type of FP unit requires three slots. If a functional unit spans more than one reconfigurable slot, only one of the entries in the resource allocation vector will contain the encoding of the functional unit and the other entries will contain the encoding 111_2 ensuring that the availability of the functional unit is only considered once in the calculation of the available function. Equation 1 can be realized in hardware using the circuit of Figure 7.

In Figure 7, each bit of the resource allocation vector and the corresponding availability signal are applied to the product, $\left(\prod_{b\in[0,2]} \overline{(type(t)_b \oplus type(i)_b)} \right) \cdot availability(i)$, computed by Equation 1.

5. Conclusions

An approach to configuration management is introduced for a superscalar reconfigurable architecture having both fixed and reconfigurable functional units. The technique proposed matches current requirements with a collection of predefined steering configurations and the current configuration. By employing partial configuration at the level of functional units, the approach effectively steers the current configuration in the direction specified by the best-matched steering configuration.

Designing the predefined steering configurations to be relatively orthogonal to one another may form the basis necessary to permit a large set of actual configurations that are actually realized, perhaps close to the entire set of possible processor configurations. The authors are currently investigating how to formulate an optimal basis, as well as, the separate problem of being able to dynamically reconfigure without using predefined configurations.

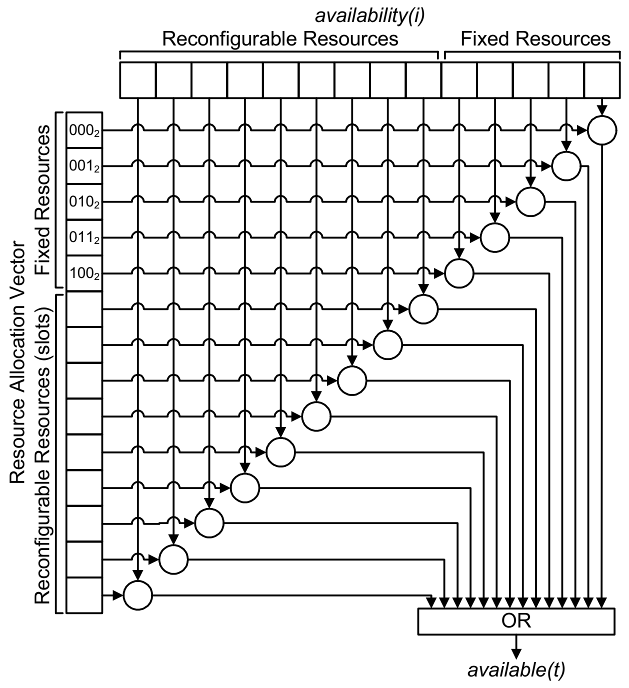


Figure 7. A circuit that computes the availability of a resource of type *t* as specified in Equation 1.

6. References

[1] Francisco Barat and Rudy Lauwereins, "Reconfigurable Instruction Set Processors: A Survey," *Proceedings of the 11th International Workshop on Rapid System Prototyping*, June 2000, pp. 168 - 173.

[2] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R.R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, Vol. 33, No. 4, Apr. 2000, pp. 70 -77.

[3] J.R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *Proceedings of the 5th Annual IEEE Symposium on Field Programmable Custom Computing Machines*, 1997, pp. 12-21.

[4] C. Iseli and E. Sanchez, "Beyond Superscalar Using FPGAs," *Proceedings of the 1993 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1993, pp. 486-490.

[5] R. Razdan and M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994, pp. 172-180.

[6] C. Iseli and E. Sanchez, "A C++ Compiler for FPGA Custom Execution Units Synthesis," *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines*, 1995, pp. 173-179.

[7] Adronis Niyonkuru and Hans C. Zeidler, "Designing a Runtime Reconfigurable Processor for General Purpose Applications," *Reconfigurable Architectures Workshop, in Proceeding of the 18th International Symposium on Parallel and Distributed Processing*, Apr. 2004, pp. 143 - 149.

[8] *Two Flows for Partial Reconfiguration: Module Based or Difference Based,* Xilinx Application Note No. XAPP290, Version 1.2, Xilinx Inc., http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf, Sept. 2004.

[9] Mary D. Brown, Jared Stark, and Yale N. Patt, "Select-Free Instruction Scheduling Logic," *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, Dec. 2001, pp. 204 - 213.