Memory Interfacing and Instruction Specification for Reconfigurable Processors

Jeffrey A. Jacob and Paul Chow
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada M5S 3G4
{jacob,pc}@eecg.toronto.edu

1. ABSTRACT

As custom computing machines evolve, it is clear that a major bottleneck is the slow interconnection architecture between the logic and memory. This paper describes the architecture of a custom computing machine that overcomes the interconnection bottleneck by closely integrating a fixed-logic processor, a reconfigurable logic array, and memory into a single chip, called OneChip-98.

The OneChip-98 system has a seamless programming model that enables the programmer to easily specify instructions without additional complex instruction decoding hardware. As well, there is a simple scheme for mapping instructions to the corresponding programming bits. To allow the processor and the reconfigurable array to execute concurrently, the programming model utilizes a novel memory-consistency scheme implemented in the hardware.

To evaluate the feasibility of the OneChip-98 architecture, a 32-bit MIPS-like processor and several performance enhancement applications were mapped to the Transmogrifier-2 field programmable system. For two typical applications, the 2-dimensional discrete cosine transform and the 64-tap FIR filter, we were capable of achieving a performance speedup of over 30 times that of a stand-alone state-of-the-art processor.

1.1 Keywords

Reconfigurable computer, FPGA, reconfigurable processor, memory interfacing, memory coherence

2. INTRODUCTION

The existence of reprogrammable logic, such as FPGAs, has allowed researchers to develop custom computing machines (CCMs) with the goal of accelerating applications relative to a stand-alone general-purpose processor. One approach is to combine a general-purpose processor and reconfigurable logic into a single system to produce a CCM with the advantages of both resources. The CPU can support the bulk of the functionality required to implement an algorithm, while the FPGA is used to accelerate only the most critical computational kernels of the program.

Previous CCMs that combine the CPU and FPGA resources into a single system first encountered slow interfaces between the processor and FPGA, demonstrating the need for higher bandwidth communication between resources [1][2][3][4]. More recent CCMs closely integrate these two resources, reducing the communication overhead [5][6][7][8][9][10]. However, simply combining the processor and FPGA into a CCM results in a memory bandwidth bottleneck. The original OneChip [7][11] integrated the processor and FPGA, while memory accesses had to go through the processor registers, limiting the memory bandwidth.

In the literature there has also been little discussion of complete methods for specifying configurable instructions, including how they are decoded, how the instructions are mapped to the reconfiguration bits, and how the reconfiguration bit streams are stored.

In this paper, we propose the OneChip-98 architecture, a follow-on to the original OneChip [7][11] that integrated a CPU, FPGA resources and memory on a single chip. OneChip-98 improves upon OneChip by providing a flexible, high-bandwidth interface to memory, including hardware to maintain memory consistency so that the processor and reconfigurable logic can operate in parallel. A scheme for specifying reconfigurable instructions is also proposed that includes a method for storing and locating the corresponding programming bit-streams for the reconfigurable logic. These two techniques require little modification to the programming model of typical processors.

In the remainder of this section, we will outline some of the other current related work in CCMs. Section 3 will elaborate on the architectural issues and give an overview of the proposed system. Instruction issue and specification is detailed in Section 4 and the memory coherence scheme is described in Section 5. A description of our prototype and test applications is given in Section 6, and we conclude and provide suggestions for future work in Section 7.

2.1 Related Work

In this section we provide a brief description of how some of the recent CCM work relates to OneChip-98.

The BRASS research group at UC Berkeley has developed GARP [8], a microprocessor core with a programmable coprocessor, instruction cache, and data cache all on a single chip. GARP addresses the memory bandwidth issue by providing the reconfigurable array with access to the standard memory hierarchy of the CPU. However, their published work seems to have focused more on the organization of the reconfigurable array and how to compile to it, rather than on the details of the interfaces.

The basic NAPA processor [9] is a single-cluster element containing a 32-bit RISC processor, 50k gates of reconfigurable logic, and local memory. The local memory is included as a solution to the memory bandwidth issue, but the reconfigurable logic has no direct connection to main memory. Therefore, data in main memory must be transferred through the CPU to the local memory.

The MIT RAW Machine [10] consists of multiple tiles, where each tile is comprised of fixed logic (registers, ALU), configurable logic, instruction memory, and data memory. The tiles are interconnected in a mesh structure, enabling communication between neighboring processing units. The RAW Machine utilizes a different approach to achieving high memory bandwidth than both GARP and NAPA. By partitioning the memory, each processor has a direct, dedicated connection to its own local memory. Such a system has a radically different programming model from the single instruction stream systems described above.

3. THE ONECHIP-98 ARCHITECTURE

The goal of the OneChip-98 architecture is to take advantage of the limitations of the typical modern processor, which generally operates on data by fetching it into registers, performing an operation and storing the result back in the registers. This limits bandwidth as well as opportunities for parallelism. Most of the available performance improvement will come from applications that can take advantage of large local storage and high memory bandwidth. Such applications are classified as memory streaming applications, which read in a block of data from memory, perform some operation on the data, and write the block of data back to memory. The operation can be very fine-grained, perhaps as simple as inverting each bit, or can be very coarse-grained, such as the 2-dimensional Discrete Cosine Transform algorithm.

For this work, the DLX RISC processor described by Patterson & Hennessey [12] was chosen as the basis processor. The DLX consists of a 32-bit instruction and data path, a five-stage pipeline, and 32 general-purpose registers. Data forwarding should be implemented to reduce read-after-write data hazards. To achieve high performance, the architecture should allow out-of-order issue and perform branch prediction to reduce the number of control hazards.

Figure 1 shows the proposed architecture of OneChip-98. Highlights are used to differentiate between those areas that must be implemented with reconfigurable logic and those that can be implemented with fixed logic.

The FPGA can be subdivided into two distinct regions. One region contains the controller, logic, and local storage needed to execute the specific application. This region must be implemented using reconfigurable logic since it is application dependent. The other region holds the instruction buffer and memory interface, both of which are application independent and can therefore be implemented using fixed logic.

It may be advantageous to use reconfigurable memory, and hence a reconfigurable memory interface, since this will provide the flexibility of either synchronous or asynchronous memory accesses with configurable memory width and memory block size. However, using reconfigurable memory will result in increased area and may adversely affect the performance of the memory system. This trade-off has not been explored in this work, but remains a topic of future research. Further information on reconfigurable memory can be found in [13][14][15].

There are several key areas of the processor-FPGA interface that must be addressed: instruction sequencing and the handling of data dependences, how to specify the FPGA instruction, FPGA dynamic reconfiguration, and data consistency amongst the processor cache, the FPGA memory, and main memory. These will be addressed in the next two sections.

4. INSTRUCTION SEQUENCING AND SPECIFICATION

As CCMs evolve it becomes important to examine more closely how details of the implementation affect the current programming models and hardware required. This section addresses the sequencing and specification of instructions for OneChip-98.

4.1 Instruction Sequencing

Since the CPU is pipelined and the latency of the function implemented in the FPGA is almost certainly greater than one CPU clock cycle, then the question is how the FPGA execution fits in the pipeline. The simple solution is to stall the pipeline while the FPGA executes. However, it would be desirable to allow the processor to continue executing in parallel with the FPGA. To handle this, Tomasulo's algorithm [12] could be used, employing reservation stations to synchronize with the FPGA resource.

The FPGA has an instruction buffer mirroring the FPGA reservation stations, so that when the buffer is full the processor will prevent any further FPGA instructions from issuing. The FPGA will execute the next instruction in its instruction buffer when the FPGA is available to do so. When an instruction is complete, the FPGA will notify the processor. Subsequent instructions in the processor that are stalled due to data dependences on the FPGA instruction are allowed to continue, and the completed FPGA instruction is removed from the FPGA reservation station.

4.2 FPGA Configuration

To integrate FPGA instructions into the instruction stream, there needs to be processor opcodes and a way to indicate how the FPGA is to be configured. For the processor there should be a dedicated opcode, say “111111”, that is set aside for FPGA instructions. If an instruction is targeted to the FPGA (by setting the instruction opcode to “111111”), then the instruction will be forwarded to the FPGA status controller. The FPGA status controller contains the FPGA reservation stations and a Reconfiguration Bits Table. The Reconfiguration Bits Table is used to keep track of FPGA configurations. An example Reconfiguration Bits Table is shown in Table 1. The first column in the table is the FPGA function. Each FPGA function points to a corresponding address in memory where the reconfiguration bits can be found. For example, function “0000” has reconfiguration bits at address 0x5000 in memory, while function “0001” has its configuration bits stored at address 0x6500 in memory. The column labelled *active* in Table 1 indicates whether that configuration is programmed into the FPGA. Only one FPGA function can be active at a given time. In the example Reconfiguration Bits Table the configuration stored at address 0x8000, corresponding to function “0010”, is active.

Dynamically Programmable Gate Arrays (DPGAs) [16][17] allow multiple configurations to be stored on the programmable logic device, while only one configuration (context) is active at any given time. The DPGA in effect acts as a cache for configuration bits. DPGAs offer much lower reconfiguration times when compared to FPGAs since the new configuration bits are already loaded onto the DPGA. Therefore, it is desirable to utilize a DPGA for the programmable logic.

A set of configurations are pre-loaded on the DPGA, setting a flag in the Reconfiguration Bits Table for each loaded FPGA function. These loaded configurations are each assigned a unique DPGA context identifier (ID) in the

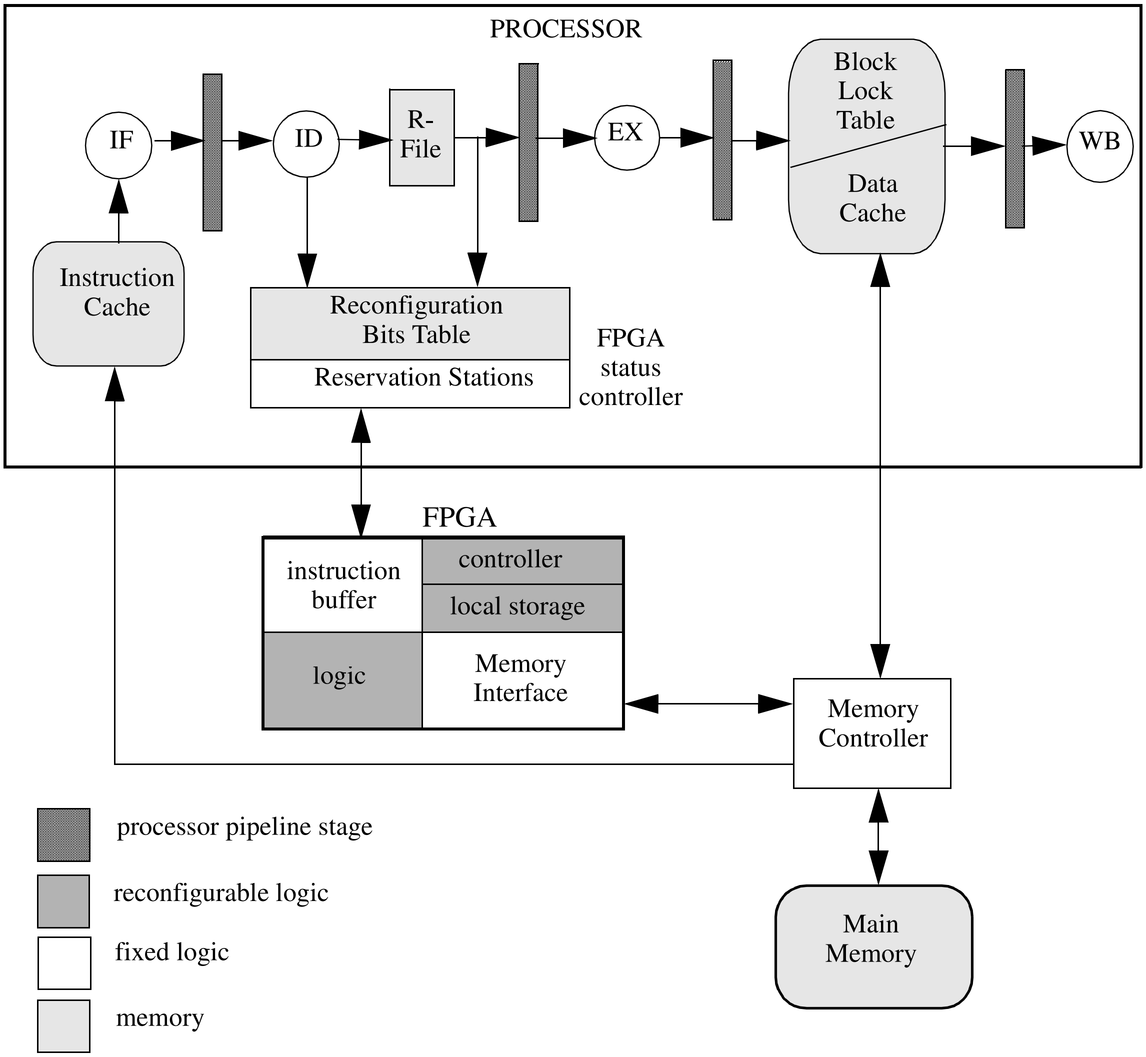

Figure 1: Proposed Architecture of the OneChip-98 System

FPGA function	memory address where configuration is stored	active	loaded (DPGA)	context ID (DPGA)
0000	0x5000	NO	YES	2
0001	0x6500	NO	NO	---
0010	0x8000	YES	YES	3
0011	0x3000	NO	YES	0
0100	0x1800	NO	YES	1

Table 1: Example Reconfiguration Bits Table

Reconfiguration Bits Table. The loaded flag and the DPGA context ID have been added to Table 1 for clarity. In the example Reconfiguration Bits Table, function “0000” is loaded in context ID 2, function “0010” is loaded in context ID 3, function “0011” is loaded in context ID 0, and function “0100” is loaded in context ID 1. Function “0001” is not loaded, and therefore does not have an associated DPGA context ID number. Re-programming the DPGA with a configuration that has already been loaded simply requires selecting the context ID corresponding to the desired configuration. If the configuration is not already loaded, then the configuration is first loaded onto the DPGA and then selected as the active configuration.

Realistically a DPGA can only store a limited number of configurations. Therefore, not all configurations are pre-loaded, instead configurations are dynamically swapped into and out of the DPGA, thus requiring a mechanism for loading new configurations. A new configuration is loaded on demand when the load flag is not set. The disadvantage of this approach is that the FPGA instruction must wait for the configuration to finish loading before proceeding. Alternatively, the new configuration can be pre-loaded using compiler directives [18]. Pre-loading occurs transparently, thereby allowing an earlier FPGA instruction to execute concurrently. Hence, pre-loading is preferred to on-demand loading.

Swapping configurations into and out of the DPGA requires another mechanism for selecting the configuration to replace. The replacement algorithm can be implemented in hardware, utilizing the least-recently-used algorithm. Alternatively, if pre-loading is used then the compiler can select which configuration to replace. This has an advantage since the compiler can statically analyze the code to determine which configurations will be most frequently used, thereby ensuring a more optimal replacement.

The FPGA instruction, which is passed to the FPGA status controller, has a field for the FPGA function. This field is used to select the corresponding row in the Reconfiguration Bits Table to ascertain the DPGA context ID. The context ID is sent along with the FPGA instruction to the FPGA instruction buffer. When the FPGA reads the next instruction in its instruction buffer, the context ID selects a configuration from the pre-loaded configurations, causing the selected configuration to become active.

The format of the FPGA instruction is shown in Figure 2, where the opcode requires six bits, the FPGA function uses four bits, the source and destination registers each require five bits, and the source and destination block sizes are each encoded into five bits. The remaining two bits (labelled *misc.*) are as yet undefined and may be used for miscellaneous tasks or information that needs to be transmitted to the FPGA. The four-bit FPGA function can address 2^4, or 16, different configurations. However, one function is required for setting the memory addresses in the Reconfiguration Bits Table, and another function would be needed for pre-loading configurations in a DPGA. Therefore, 14 different configurations can be addressed, each requiring a single row of storage in the Reconfiguration Bits Table.

The user software program must initialize the Reconfiguration Bits Table with the FPGA configurations that will be needed for the program to execute. For each available FPGA function a memory address is required, indicating where the configuration bits can be found in memory.

4.3 Instruction Format

Recall from Section 3 that the FPGA can offer the most benefit for memory streaming applications. This means that data in a source block of memory is processed and stored in a destination block of memory. The FPGA instruction, as specified in the instruction-set of the processor, needs to indicate the source and destination blocks in memory, in addition to the FPGA function. This implies a memory-memory type of instruction, which does not fit the register-register model of modern processors. The main problem is that there are not enough instruction bits to specify the source and destination addresses and the size of the memory blocks.

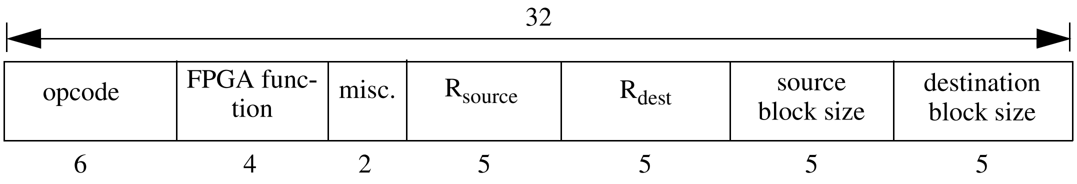

Figure 2: FPGA Instruction Format

The source and destination addresses can be specified by using an indirect addressing mode instead of direct or absolute addressing. The source address and destination address are each stored in registers in the processor register file. The FPGA instruction then references the source and destination registers where the corresponding addresses are stored. When an FPGA instruction is decoded (in the ID stage), the source address and destination address are read from the source register, R_{source}, and destination register, R_{dest}, respectively. These addresses are stored in the FPGA reservation station and are subsequently passed to the FPGA.

The memory streaming application running on the FPGA reads a block of data from the source address, processes it, and writes the result to a block of data at the destination address. The source block size identifies the amount of data to be read beginning at the source address, while the destination block size identifies the amount of data to be written beginning at the destination address.

Both the source and destination blocks contain a power of two number of words (i.e. 2, 4, 8, 16,...), and the corresponding address must be aligned on a block boundary. For example, suppose the source block size is 64 words, then the source address must be a multiple of 64 words. The block size cannot be larger than 2^{32} words, the amount of memory addressable with a 32-bit datapath. Hence, the block size can be encoded in five bits, allowing 2^5 or 32 different block sizes, where each block size is a power of two number of words. This block size restriction limits flexibility a little bit, but makes it easier to check memory coherence, which is described in Section 5.

The five-bit source block size and five-bit destination block size are each included in the FPGA instruction. When the FPGA instruction is decoded, the source block size is decoded into 32 bits and is passed to the FPGA reservation station. The source block size is used to mask the low order bits of the source address, as shown in Table 2, where 'x' is a mask bit. The source address 0x6a20 is shown in the first row of the table. The source block size "00100" in the FPGA instruction is decoded into the 32-bit value shown in the second row of the table. This value specifies the mask bits for the source address. All bits to the left of the set bit in the 32-bit source block size are significant, identifying the source block of addresses. All bits to the right of the set bit (including the set bit) mask the low order bits of the source block address. In Table 2, the fourth bit (counting from bit zero) of the source block size is set, masking bits zero through four of the source address. The third row of the table displays the mask bits, while the fourth row of the table shows the source address after the low order bits have been masked. This masked address signifies that the source data extends from address 0x6a20 to 0x6a3f, and the FPGA reads 32 words of data. The destination block size masks the low order bits of the destination address in a manner identical to the source block size.

The block size may vary depending on the granularity of the application mapped to the FPGA. The application may have a fixed memory block size or a variable memory block size. A fixed memory block size application computes on a predetermined amount of data that is specified by the hardware circuit in the FPGA, while a variable memory block size application performs computations on the amount of data specified in the block sizes of the FPGA instruction. In either case, the block sizes must be included in the FPGA instruction for the purpose of maintaining memory consistency, a topic deferred to Section 5.

Although we have shown a two-operand instruction format, it may also be desirable to have a three-operand format. In this case, one of the block size fields could be used as another register specifier. This means that the memory blocks would all be restricted to the same block size, which is not likely to be a severe constraint.

5. MEMORY COHERENCE

Assuming there are no data dependences between CPU instructions and FPGA instructions the two resources can execute concurrently, thus providing the potential for greater speedup, similar to a multiprocessor system. In OneChip-98, data can be stored in main memory, the processor cache, or the local memory of the FPGA, leading to possible memory coherence problems when data must be shared. Cache and memory coherence in the OneChip-98 system is the topic of this section.

5.1 Situations Causing Memory Inconsistency

The memory consistency problem can manifest itself in six different ways, each situation giving rise to a type of data hazard [12]. The first two situations are caused by read-after-write (RAW) data hazards, the next two situations are caused by write-after-read (WAR) data hazards, and the last two situations are caused by write-after-write (WAW) data hazards. These situations are listed in Table 3. As an example, we will expand upon one situation, but more detailed examples and explanations can be found in [19].

In the first situation shown in Table 3, the processor updates

source address	0000 0000 0000 0000 0110 1010 0010 0000
source block size	0000 0000 0000 0000 0000 0000 0001 0000
source block mask	0000 0000 0000 0000 0000 0000 0001 1111
source address masked	0000 0000 0000 0000 0110 1010 001x xxxx

Table 2: Source address and source block size in the FPGA reservation station

data in its cache. The FPGA then, attempting to access the updated values, reads stale data in main memory. To reduce write stalls and memory traffic, we prefer to use a write-back cache, where data in the cache is updated without updating the memory. To maintain cache consistency in a write-back cache, the processor must flush the cache after the store completes and prior to the FPGA reading data from memory, thereby ensuring that the FPGA reads the correct data.

5.2 Solving the Memory Consistency Problem

The situations described above each necessitate some action to ensure memory coherence. These actions are listed in Table 3. The following discussion presents the mechanisms needed to perform each action.

For Action 1, the source address and source block size of the FPGA instruction are sent to the cache when the source address becomes available. The source block size is used to mask the low order bits of the source address making them into “don’t care bits” when performing the cache lookup. Any dirty locations that match must be flushed from the cache.

The mechanism for invalidating the destination block in the cache is similar to the mechanism for flushing the source block from the cache except that the destination address and destination block size are used. This provides memory coherence according to Action 3 in Table 3.

The mechanisms used to enforce Actions 4, 6, and 8 are more complicated since they must somehow prevent the processor from accessing blocks of main memory. This is performed by using a Block Lock Table. The Block Lock Table is responsible for storing the source address masked and destination address masked for all outstanding FPGA instructions.

A Block Lock Table is shown in Table 4. The table has three columns: *Block Address Masked*, *FPGA Instruction Tag*, and a flag indicating whether the address is a source or destination address. The block address, in conjunction with the block size, is used to determine the range of addresses that will be accessed by the FPGA. This address range is stored using mask bits, or “don’t care bits”, in the column labelled *Block Address Masked*. The number of mask bits is determined by the block size. The FPGA instruction tag identifies which FPGA instruction accesses which block addresses so that the corresponding entry can be removed when the FPGA instruction completes. The table contains two entries for each outstanding FPGA instruction -- one entry for the source block and the other entry for the destination block. The source / destination flag identifies whether the entry corresponds to a source block or a destination block.

The example Block Lock Table in Table 4 shows two outstanding FPGA instructions. The instruction with a tag equal to one has a source address of 0x9000 and destination address of 0x4000, with source and destination blocks each containing 0x400 words. Hence, the source block address ranges from 0x9000 to 0x93ff, and the destination block address ranges from 0x4000 to 0x43ff. The other instruction has a source address of 0x2800, a source block size of 0x200, a destination address of 0x3600, and a destination block size of 0x80. Hence, the source block address ranges from 0x2800 to 0x29ff, and the destination block address

Situation Number	Problem Situation	Actions Taken
1	FPGA read after CPU write	1. Flush FPGA source addresses from CPU cache when FPGA instruction issues 2. Prevent FPGA reads while pending CPU store instructions are outstanding
2	CPU read after FPGA write	3. Invalidate FPGA destination addresses in CPU cache when FPGA instruction issues 4. Prevent CPU reads from FPGA destination addresses until FPGA writes its destination block
3	FPGA write after CPU read	5. Prevent FPGA writes while pending CPU load instructions are outstanding
4	CPU write after FPGA read	6. Prevent CPU writes to FPGA source addresses until FPGA reads its source block
5	FPGA write after CPU write	7. Prevent FPGA writes while pending CPU store instructions are outstanding
6	CPU write after FPGA write	8. Prevent CPU writes to FPGA destination addresses until FPGA writes its destination block

Table 3: Problem situations and actions taken to ensure memory coherence

ranges from 0x3600 to 0x367f.

Block Address Masked	FPGA Instruction Tag	source / destination
0010 100x xxxx xxxx	2	SOURCE
0011 0110 0xxx xxxx	2	DESTINATION
0100 00xx xxxx xxxx	1	DESTINATION
1001 00xx xxxx xxxx	1	SOURCE

Table 4: Block Lock Table

The Block Lock Table is used to prevent the processor from accessing certain blocks of memory according to Actions 4, 6, and 8. Action 6 must prevent the processor from writing to an FPGA source block address. The source address and source block size, which are sent to the cache for flushing, are simultaneously sent to the Block Lock Table. The source/destination flag is set to SOURCE, indicating that writes to this block of addresses are illegal and will cause the store instruction to stall, while reads from this block of addresses are permitted.

To enforce Action 6, all CPU store instructions must be checked in the Block Lock Table to ensure that the store address does not lie within an FPGA source block, otherwise, the store is stalled until the FPGA instruction completes.

Actions 4 and 8, which prevent CPU reads from or writes to an FPGA destination block address, are similar to Action 6 except that the destination address and destination block size are used.

Action 8 requires that CPU store instructions, whose addresses are compared in the Block Lock Table for a match in the source blocks, must also be checked for matches in the destination blocks. Hence, a store instruction may ignore the source / destination flag, since a match in either block would stall the store instruction. Action 4, on the other hand, applies to CPU load instructions whose address only needs to be compared in the destination blocks. Therefore, load instructions must check the source / destination flag -- load addresses that match in the destination block must be stalled, while load addresses that match in a source block can continue (since a read after read, RAR, is not a hazard).

A CPU load or store instruction that stalls due to data dependence on an FPGA instruction can continue when the FPGA instruction completes. Upon completion, the instruction tag is sent to the FPGA reservation station, and the corresponding entry in the reservation station is removed. The instruction tag is also forwarded to the Block Lock Table, and the two entries in the Block Lock Table are deleted -- one entry corresponding to the source block, and the other entry corresponding to the destination block. Since those memory blocks are no longer locked, load and store instructions that stalled due to dependences on the FPGA instruction can then proceed.

Actions 2, 5, and 7 prevent FPGA reads or writes while any issued CPU load or store instruction has not completed. To fulfill this requirement we could prevent the FPGA from issuing as long as there is any pending load or store instruction in the CPU. Alternatively, we could set up an Address Lock Table, similar to a Block Lock Table, that will only prevent the FPGA instruction from issuing when a CPU load or store address is unknown. Once the CPU memory address becomes available, the FPGA instruction could issue (assuming there is no address conflict between the FPGA source and destination block addresses and the CPU load or store instruction address). However, it is difficult to justify the overhead of an Address Lock Table since load and store instructions in the CPU can expect to complete within a short period of time (this delay may be only a few CPU clock cycles). Therefore, we simply prevent the FPGA from issuing while there are any pending load or store instructions in the CPU.

6. IMPLEMENTATION OF THE ONECHIP-98 PROTOTYPE

The OneChip-98 system was mapped onto the Transmogrifier-2 [20] Field Programmable System for prototyping purposes. Once mapped, the architecture of OneChip-98 could be evaluated, and the performance of the system could be extrapolated. The TM-2, however, has several inherent limitations that prevent the OneChip-98 system from being accurately modeled. This section briefly discusses these limitations, and the approach taken to minimize their effect.

6.1 Fixed and Reconfigurable Logic

The OneChip-98 system consists of fixed logic, reconfigurable logic, and memory as shown in Figure 1. Due to the fact that the OneChip-98 system is being mapped to a multi-FPGA system, both the fixed logic and reconfigurable logic portions of the system must be implemented in FPGAs. The disadvantage of this implementation scheme is two-fold: FPGAs do not offer the same performance as fixed logic, and FPGAs deliver a much lower logic density than fixed logic. The advantage is that different approaches can be implemented and evaluated.

The OneChip-98 architecture also stipulates that the processor and FPGA must be highly integrated. However, the TM-2 forces us to partition the design amongst two Altera Flex10K50 PLDs [21] connected with I-cube [22] devices providing a flexible interconnect, as shown in Figure 3. The I-cube chip adds a 20 ns latency to signals, whereas a true realization of OneChip-98 would not incur this delay. Therefore, to accurately model the OneChip-98 system this delay should be neglected.

6.2 Processor Modifications

The MIPS-like RISC processor was implemented in an Altera Flex10K50 PLD. The processor, however, was modified for several reasons:

(a) due to constraints inherent in the structure of the TM-2,

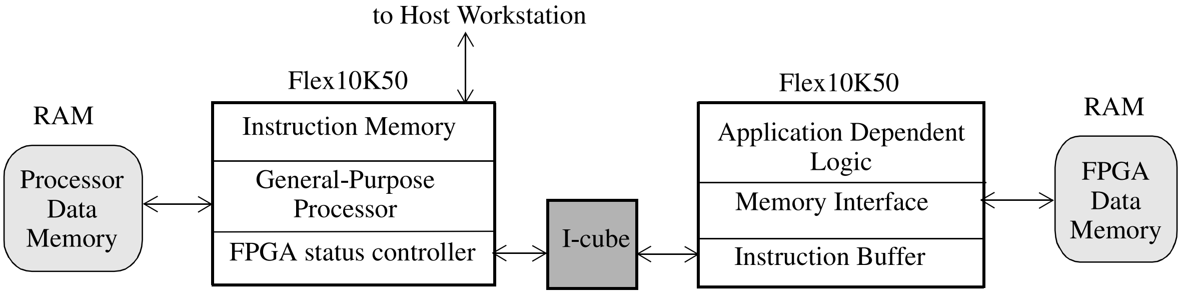

Figure 3: TM-2 Implementation of the OneChip-98 System

(b) certain capabilities of the processor were not essential in modeling, prototyping, and evaluating the system, and

(c) to fit in the Altera device.

Hardware constraints on the TM-2 do not permit an FPGA to reconfigure another FPGA, hence run-time reconfiguration is not possible on the TM-2, so the Reconfiguration Bits Table was not implemented in the processor.

The structure of the TM-2 only permits an FPGA to access data from its neighboring RAMs, preventing multiple FPGAs from sharing a memory port. This additional constraint inhibits memory coherence from being modeled. The Block Lock Table was therefore not implemented in the processor. Alternatively, a tertiary FPGA on the TM-2 could be used as a memory, allowing a memory to be attached to the other two FPGAs (one containing the CPU and the other containing the reconfigurable logic). This would enable us to adequately model memory coherence between the reconfigurable logic local memory and main memory. This however was not implemented and remains a topic of future work.

Tomasulo’s algorithm is not implemented in the processor, with the exception of the FPGA status controller, which uses reservation stations to buffer FPGA instructions. Thus, out-of-order issue cannot be performed.

The FPGA uses a fixed memory block size, accessing a predetermined amount of data. Since the FPGA uses a fixed memory block size and the Block Lock Table is not implemented, the source and destination block sizes are not needed, and are therefore not included in the FPGA instruction. In addition, the FPGA function and DPGA context ID are not included in the OneChip-98 implementation since run-time reconfiguration is not possible.

6.3 Applications and Results

Although we could not test any of the dynamic reconfiguration or memory consistency issues, we did implement two applications to test the basic instruction issue and memory interface. Our philosophy is that the actual prototyping of the these interfaces is the best way to test their feasibility. We chose two applications, the DCT algorithm and the FIR filter, as examples of memory streaming applications.

The HDL code for each application circuit is highly tuned. Hence, it is unlikely that these circuits would be automatically generated. Instead, we envision a library of hardware modules, similar to Hardware Object Technology (HOT) [23], where the programmer simply selects the hardware object from a library of pre-synthesized circuits. The details of the library and the method in which a hardware object is instantiated by the user are topics for future work.

To test the application circuit, the host workstation must communicate with the RISC processor by means of loading instructions into the processor’s instruction memory. The application circuit is targeted by loading an FPGA instruction into the processor’s instruction memory. The application circuit then accesses data from memory, performs some computation on the data, and writes the result back to memory. Thus the functionality of the application circuit is verified by testing the contents of memory before and after the FPGA instruction. The performance of the application circuit is measured using a logic analyzer.

After scaling the actual performance to that of a state-of-the-art implementation, the performance results and speedup of the application circuits are shown below. Table 5 summarizes the performance and speedup of the 2-dimensional DCT implementation, while Table 6 summarizes the performance and speedup of the 64-tap FIR filter.

The results shown in Table 5 and Table 6 are determined on the basis of downloading several consecutive FPGA instructions. These instructions are decoded in the processor and computed by the application circuit running on the FPGA. Hence, this methodology does not account for any software effects, such as instruction scheduling or loop unrolling. To model these effects, one must create a compiler that can schedule instructions to minimize interface bottlenecks between the processor and the FPGA, minimize the amount of context switching in the FPGA, and provide a load balance between processor instructions and FPGA instructions. Modeling these software effects is a topic of future work.

The 2-dimensional DCT and the 64-tap FIR filter are both

Implementation	System	Execution time
Ultra 2 (300 MHz)	actual	17 µs
	scaled by 8 assuming MMX-like **(A)**	2.1 µs
FPGA	Flex10K50-4 (using approximately 47,000 gates)	1.59 µs
	Flex10K50-1 (using approximately 47,000 gates)	0.61 µs
	Flex10K100 (better scheduling) **(B)** (using approximately 64,000 gates)	0.20 µs
Speedup	FPGA vs. Ultra 2 **(A/B)**	10x

Table 5: Performance results and speedup of the 2-dimensional DCT

Implementation	System	Execution time
Ultra 2 (300 MHz)	actual	2.53 µs
	scaled by 4 assuming MMX-like **(A)**	0.63 µs
DSP 56K	50 MHz processor **(B**	1.32 µs
FPGA	Flex10K50-4 (using approximately 37,000 gates)	0.053 µs
	Flex10K50-1 **(C)** (using approximately 37,000 gates)	0.020 µs
Speedup	FPGA vs. Ultra 2 **(A/C)**	32x
	FPGA vs. DSP **(B/C)**	65x

Table 6: Performance results and speedup of the 64-tap FIR filter

memory streaming applications. The 2-dimensional DCT is a coarse-grained application, reading and writing 64 bits every clock cycle. The FIR filter, on the other hand, is fine-grained, reading eight bits and writing ten bits every cycle. Both applications, when implemented on a state- of-the-art PLD, show substantial speedup relative to their software (and DSP) counterparts. This proves that the OneChip-98 reconfigurable computer can speed up memory streaming applications, irrespective of the granularity of the application.

The 2-dimensional DCT and 64-tap filter applications are mapped to an Altera general-purpose programmable logic device. Custom programmable logic may provide higher performance than general-purpose programmable logic, thereby increasing the potential for greater speedup. In addition, the performance results for the hardware implementation are based on the execution time of the application running on the PLD. However, in the OneChip-98 reconfigurable system, the processor can execute concurrent to the PLD. This enables independent instructions to execute in parallel, thereby providing additional speedup. This additional speedup has not been considered in the results above.

7. CONCLUSIONS AND FUTURE WORK

The evolution of custom computing machines has shown that the memory interface must be carefully considered to avoid a serious processing bottleneck. In this paper, we have described the OneChip-98 architecture that tightly couples a fixed-logic processor with reconfigurable logic and memory onto a single chip. The target applications are memory streaming, reading data from memory, operating on it, and storing the results back to memory. In this way, data can be directly accessed by the reconfigurable logic without needing to deal with the overhead of going through the fixed processor.

The proposed architecture introduces potential memory consistency problems. To simplify the programming model, we describe how the hardware can manage the problem. In particular, we introduce the concept of the Block Lock Table, which is used to store the sections of memory that are being accessed. We also describe a means for quickly finding locations to flush or invalidate in a cache by using “don’t care” bits in the tag compare.

A scheme for specifying reconfigurable instructions and managing the storage and access of the programming bits is also proposed. This scheme requires little additional hardware to be added to the typical fixed-logic processor model.

The OneChip-98 architecture enables the processor and the reconfigurable logic to execute concurrently. This not only provides increased performance, but also allows the processor to support small interrupts such as I/O handling. These interrupts do not require much computation and therefore need not affect the reconfigurable logic.

Two applications, one coarse-grained and one fine-grained, were mapped to the OneChip-98 prototype and showed improved performance over a general-purpose processor. Although none of the dynamic reconfiguration or memory consistency mechanisms could be validated, this implementation did demonstrate that the proposed instruction issue and memory interfaces worked well without adding complex hardware to the traditional models. The 2-dimensional Discrete Cosine Transform, a coarse-grained application, obtained a ten-fold speed improvement relative to the stand-alone processor, while the 64-tap finite impulse response filter, a relatively fine-grained application, achieved a performance speedup of over 30 times that of the general-purpose processor. These performance results indicate that the OneChip-98 system can successfully attain speedup for a wide class of applications.

Future work should investigate issues such as whether out-of-order issue provides significant benefit in these systems. More thorough experimentation and benchmarking of applications will provide greater insight into the cost-benefit trade-off. OneChip-98 uses a fixed memory interface. As experience grows with more applications being implemented, it may be observed that a reconfigurable memory interface would be more suitable. The field-programmable system did not allow for the implementation of dynamic reconfiguration or memory consistency. A custom approach could provide these features. Lastly, we have not considered the details of how the programmer actually specifies or designs the hardware that gets mapped to the reconfigurable logic. This is also an important area to study.

8. ACKNOWLEDGMENTS

We would like to acknowledge Altera for helping to develop and build the Transmogrifier-2, for donating components, and for their ongoing support. Additionally, we wish to thank the Natural Sciences and Engineering Research Council of Canada and the Micronet Centre of Excellence for financial support.

9. REFERENCES

[1] R. Jeschke, “An FPGA-Based Reconfigurable Coprocessor for the IBM PC”, M.A.Sc. Thesis, University of Toronto, 1994.

[2] J. Vuillemin et al, “Programmable Active Memories: Reconfigurable Systems Come of Age”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol 4, No 1, 1996, pp. 56-69.

[3] J. Arnold, D. Buell, and W. Kleinfelder, “Splash 2: FPGAs in a Custom Computing Machine”, *IEEE Computer Society Press*, 1996.

[4] P. Athanas and H. Silverman, “Processor Reconfiguration Through Instruction-Set Metamorphosis”, *Computer*, March 1993, pp. 11-18.

[5] R. Razlan and M. Smith, “A High Performance Microachitecture with Hardware Programmable Functional Units”, *Proceedings of 27th Annual International Symposium on Microarchitecture*, Nov. 1994, pp. 172-180.

[6] S. Hauck, T. Fry, M. Hosler, J. Kao, “The Chimaera Reconfigurable Functional Unit”, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’97)*, April 1997, pp 105-116.

[7] R. Wittig and P. Chow, “OneChip: An FPGA Processor with Reconfigurable Logic”, *IEEE Symposium on FPGAs for Custom Compute Machines (FCCM’96)*, March 1996, pp. 126-135.

[8] J. Hauser and J. Wawrzynek, “Garp: A MIPS Processor with a Reconfigurable Coprocessor”, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’97)*, April 1997, pp 24-33.

[9] C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold and M. Gokhale, “The NAPA Adaptive Processing Architecture”, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’98)*, April 1998.

[10] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, “Baring it all to Software: RAW Machines”, *IEEE Computer*, September 1997, pp. 86-93.

[11] R. Wittig, “OneChip: An FPGA Processor With Reconfigurable Logic”, M.A.Sc. Thesis, University of Toronto, 1995.

[12] J. Hennessy and D. Patterson, *Computer Architecture A Quantitative Approach, second edition*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[13] T. Ngai, J. Rose, and S. Wilton, “An SRAM-Programmable Field-Configurable Memory”, *IEEE Custom Integrated Circuits Conference*, 1995, pp. 499-502.

[14] S. Wilton, J. Rose, and Z. Vranesic, “Architecture of Centralized Field-Configurable Memory”, *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 1995, pp. 97-103.

[15] S. Wilton, J. Rose, and Z. Vranesic, “Memory/Logic Interconnect Flexibility in FPGAs with Large Embedded Memory Arrays”, *IEEE Custom Integrated Circuits Conference*, May 1996, pp. 144-147.

[16] A. DeHon, “DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century”, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’94)*, April 1994, pp 31-39.

[17] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, “A Time Multiplexed FPGA”, *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM’97)*, April 1997, pp. 34-40.

[18] S. Hauck, "Configuration Prefetch for Single Context Reconfigurable Processors," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'98)*, 1998, pp. 65-74.

[19] J. Jacob, “Memory Interfacing for the OneChip Reconfigurable Processor”, M.A.Sc. Thesis, University of Toronto, 1998.

[20] D. Lewis, D. Galloway, M. van Ierssel, J. Rose, and P. Chow, “The Transmogrifier-2: A 1 Million Gate Rapid Prototyping System”, *IEEE Transactions on VLSI Systems*, June 1998.

[21] Altera Corporation, *Altera Data Book*, 1996. On-line: http://www.altera.com.

[22] I-cube Corporation, On-line: http://www.icube.com.

[23] S. Casselman, M. Thornburg, and J. Schewel, “Hardware Object Programming on the EVC1 - a Reconfigurable Computer”, *FPGAs for Rapid Board Development and Reconfigurable Computing*, 1995.