INSTRUCTION SET ARCHITECTURE OF AN EFFICIENT PIPELINED DATAFLOW ARCHITECTURE

Guang R. Gao
School of Computer Science VLSI Design Laboratory
McGill University
3480 University
Montréal, Quebec H3A 2A7

René Tio
VLSI Design Laboratory
McGill University
3480 University
Montréal, Quebec H3A 2A7

ABSTRACT

A highly pipelined static dataflow architecture based on an argument-fetching data-driven principle has recently been proposed (as reported by Dennis and Gao [5]). It separates the data-driven instruction scheduling mechanism from the actual instruction execution unit, avoiding the unnecessary overhead of data token movement that exists in other proposals of dataflow architectures. This paper describes the work carried out at McGill University on the instruction set design and machine program format. The implementation of long-latency operations — the structure memory operations and interprocessor communication operations — are discussed, as is the implementation of FIFO buffers. The design of an assembler and instruction set interpreter for the new architecture is briefly outlined.

1 INTRODUCTION

A highly pipelined static dataflow processor architecture has recently been proposed, as described by Dennis and Gao [5]. This new architecture is based on an argument-fetching data-driven principle: the arguments of an instruction never “flow” in the new architecture, but instruction scheduling remains data-driven. The new architecture does not keep data information (values) and control information (addresses) bound together in packets as in previously proposed dataflow architectures. Instead, it separates the data and signaling roles of the information packets [5]. An instruction in the new architecture behaves much like one in any conventional von Neumann processor, fetching its own arguments from and storing its result to a data memory. This avoids the problem of high data traffic volume caused by the flow of data tokens.

A static argument-fetching dataflow processor consists of two parts: a pipelined instruction processing unit (PIPU) and a dataflow instruction scheduling unit (DISU). The PIPU is similar to a conventional pipelined instruction processor except that it does not have a program counter. The DISU plays the role of the program counter in the sense that it maintains a pool of addresses for instructions to be executed next. A data-precedence signal graph of the program is operated on by the DISU to determine which instructions are enabled based on the dataflow firing rules.

The data-driven scheduling mechanism guarantees that no instruction is initiated unless it has no data conflict with other instructions in the pipeline of PIPU. Therefore, it can completely avoid the problem of pipeline gaps in conventional architectures caused by conditional branches and data dependence hazards. This paper describes recent work carried out at McGill University on the instruction set design and machine program format for a static argument-fetching dataflow architecture. The implementation of long-latency operations — such as the structure memory operations and interprocessor communication operations — are discussed, as is the implementation of hardware-supported FIFO data buffers.

2 THE ARGUMENT FETCHING PRINCIPLE

2.1 The Architecture

A static argument-fetch dataflow processor is shown in figure 1. It contains a pipelined instruction processing unit (PIPU) and a dataflow instruction scheduling unit (DISU). The role of the DISU is to execute the signal graph of the dataflow program and maintain a pool of instructions that are ready for execution. The role of PIPU is to actually execute these instructions. A more detailed description of the architecture can be found in [5].
2.2 The Program Tuple

A program tuple is used to represent the program graph \( G \) to be executed on the new architecture, such that

\[
G := \langle P, S \rangle
\]

where \( P \) is a set of PIPU instructions and \( S \) a signal flow graph. Each actor in the dataflow program graph is represented by a pair of entries in \( P \) and \( S \).

The instruction graph \( P \) is a list of instructions where:

\[
P := \langle p\text{-inst-list} \rangle
\]

\[
\langle p\text{-inst-list} \rangle :=
\langle p\text{-instruction} \rangle \\
| \langle p\text{-inst-list} \rangle
\]

\[
\langle p\text{-instruction} \rangle :=
\langle opcode \rangle \langle op\text{-address} \rangle \langle op\text{-address} \rangle \\
\langle result\text{-address} \rangle
\]

\[
\langle op\text{-address} \rangle, \langle result\text{-address} \rangle := \\
\langle mode \rangle \langle address \rangle
\]

The p-instructions are three address instructions similar to those in conventional architectures and are executed by the PIPU. The address mode is used to calculate the effective address of operands. Currently supported addressing modes include immediate, direct and indirect modes.

The signal graph \( S \) consists of a set of signal nodes:

\[
G := \langle s\text{-node-list} \rangle
\]

\[
\langle s\text{-node-list} \rangle :=
\langle s\text{-node} \rangle \\
| \langle s\text{-node-list} \rangle
\]

\[
\langle s\text{-node} \rangle :=
\langle signal\text{-count} \rangle \langle signal\text{-list} \rangle \\
\langle enable\text{-count} \rangle \langle reset\text{-count} \rangle
\]

\[
\langle signal\text{-list} \rangle :=
\langle u\text{-address-list} \rangle \langle t\text{-address-list} \rangle \langle f\text{-address-list} \rangle \\
\langle u\text{-address-list} \rangle, \langle t\text{-address-list} \rangle, \langle f\text{-address-list} \rangle := \\
\langle s\text{-address-list} \rangle
\]

\[
\langle s\text{-address-list} \rangle :=
\langle s\text{-address} \rangle \\
| \langle s\text{-address-list} \rangle
\]

\[
\langle s\text{-address} \rangle :=
\langle address \rangle \langle mode \rangle
\]

The three address lists in each s-node entry correspond to the unconditional, true and false signaling lists, which are used in the implementation of conditional expressions [5]. Each address in the address list has an associated mode field which may be used to implement procedure calls and signal list sharing.

2.3 Instruction Execution

In the static argument-fetching architecture, instruction execution is divided into two phases: the PIPU execution phase and the DISU scheduling phase. The PIPU execution phase starts when a firing signal for an instruction is sent by the DISU to the PIPU. A firing signal is simply the address of a p-instruction. The PIPU will then retrieve and execute this instruction in a conventional pipelined manner. The p-instruction address is carried with the operation as it traverses the processor pipeline, and, when the operation completes, the address is released as a done signal. Note that the pipeline does not exhibit strict FIFO behavior regarding the release of done signals, since it may be delayed...
Fetah Pi (done. Pi. CC) to scheduling operands

The DISU scheduling phase begins by processing a done signal from the PIPU. A done signal contains both the address of the instruction which has just finished its PIPU phase and a condition code returned by the execution of the instruction. This conditional code is either T or F. When a done signal is received, the s-instruction at the address specified by the done signal is retrieved. Each s-instruction contains three address lists designated the true, false and unconditional list, and if the condition code returns T (F), a count signal is sent to each address in the true (false) list. In any case, the addresses in the unconditional signal list are always signaled. The DISU then retrieves the status information of the s-instruction specified by the count signal and decrements its enable count field. The instruction is identified as enabled when this count reaches zero. At that time, the count field of the enabled s-instruction is returned to the value of reset, and finally, the DISU chooses an enabled instruction and sends a fire signal to the PIPU. Since there may be more than one enabled instruction, the DISU uses a fair scheduling mechanism to determine the order in which the instructions are fired.

The phases of the instruction execution mechanism are illustrated in figures 2 and 3.

3 SPLIT-TRANSACTION MEMORY OPERATIONS

At McGill university, we are experimenting with split-transaction memory operations as an extension of conventional dataflow operations to support accesses in structure memory.

In the instruction set design, Load and Store operations are used to transfer scalar datum between data memory and structure memory. Array operations are translated into a pair of instructions: an L-INDEX and LOAD for a SELECT operation, and an S-INDEX and STORE for APPEND (or UPDATE). The index instructions are used to compute the
actual element address from the base address of the array \( A \) and the index \( i \) for the element. The element address is then used by LOAD or STORE to perform the actual memory operations. The split-transaction style of memory operations provides a way where memory latency can be tolerated in a dataflow computer. The format of the structure memory instructions is discussed in [7].

We are currently investigating efficient implementation schemes for structure memory operations. In the following, we outline one suggested implementation.

The Structure Memory Unit (SMU) performs several functions for array accesses. For the index operations, the SMU calculates the absolute address of the array element in structure memory given \( A \) and \( i \), and also manages the parking and unparking of delayed done signals for the index operations. For LOAD/STORE operations, the SMU receives an absolute address and fetches the data at that address from structure memory.

Each location in structure memory has a corresponding entry in the SMU parking store. Each of these entries has a valid bit field and an address field. The valid bit, when set, signifies that the corresponding structure memory location contains valid data. When reset, the valid bit indicates that any L-INDEX instruction that references the location should be parked. The address field is simply a pointer to a list in the unformatted area of parking store memory. This list contains all the parked done signals of other L-INDEX operations awaiting the same array element.

The parking mechanism of the SMU operates in the following manner:

- When an L-INDEX operation is executed by the SMU, the absolute address of the array element is calculated. The validity of the array access is then checked by reading the valid bit of the location and, if set, the done address is released as normal to the following stage (the operand store stage). The operand store stage then stores the absolute address in a result register (to be used by the following LOAD instruction) and the instruction address is released as a done signal to the DISU.

- If the valid bit is reset, the done signal (which is simply the p-instruction address) for the L-INDEX operation is inserted in the list of delayed done signals in the SMU parking store (fig. 4). However, the operand store stage will continue as normal, storing the absolute address for later use. The difference is that the instruction address, which normally accompanies the instruction through the pipeline, is specially marked, so the PIPU will not release a done signal when the operand store is completed.

If an L-INDEX operation is parked, it must await the firing of an S-INDEX operation which references the same absolute address. The unparking mechanism operates in the following manner:

- The S-INDEX operation calculates the absolute address from \( A \) and \( i \) and sets the valid bit of the corresponding parking store entry. The absolute address of the array element is then released with the operation address to the operand store stage, which will proceed as normal (i.e., store the address and release the done signal.)

- As the valid bit of the parking store entry becomes set, the list of delayed dones is retrieved and each address is released back into the PIPU pipeline (fig. 5). The operand store stage performs no action upon receiving these delayed dones, but simply releases them as a done signal.

- The done signal for the S-INDEX operation immediately causes the firing of the corresponding STORE
using the "short-cut" firing mechanism described in [7]. This ensures that no LOAD instructions, which must go through the PIPU-DISU-PIPU cycle, will be fired before the STORE has begun.

As an added measure of safety, the done signal for the S-INDEX operation is released before any of the delayed dones. This further guarantees the STORE will be "short-cut" fired by the S-INDEX done signal before the delayed dones can even begin enabling any LOADs.

4 PARKING STORE AND INTERPROCESSOR TRANSACTIONS

Dataflow architectures eliminate the software overhead needed for interprocessor synchronization since no interrupts, context switches, or busy waits are required for such purposes. In this paper, we outline the "parking" mechanism in the Interprocessor Communication Unit (IPC Unit) based on the interprocessor argument-fetching principle. In this scheme, a non-local argument address is specified by an addressing mode in one (or both) of the input operands of a p-instruction.

When an instruction fires, its operands are normally decoded and fetched by the two operand fetch stations (OpFetch1 and OpFetch2) before continuing on to the execution unit. If the instruction contains at least one remote argument, the entire instruction will be parked after OpFetch2 regardless of which argument requires the remote data. This allows the IPC Unit to park the instruction just once even if it has two remote operands.

When either OpFetch1 or OpFetch2 detects a remote argument, it constructs an IPC request

\[ <I, s, n> \]

where \( I \) is the p-instruction address, \( s \) the remote argument spec, and \( n \) a tag indicating whether the request is from the first or second argument of the instruction (i.e., is either 1 or 2,) and sends this request to the IPC Unit. The instruction is then tagged as an IPC operation. If the first argument is remote, the instruction continues on to OpFetch2, which will decode and fetch the second operand, or initiate another IPC request. In any case, the instruction is parked after OpFetch2 has completed either the fetch or the IPC request for the second operand.

The IPC Unit services requests on a FCFS basis. As a request \( <I, s, n> \) is received, it performs an associative search on \( I \) and attempts to find a parking store entry with the same tag. If the search fails, a free entry with tag \( I \) is allocated for the instruction (this method may be modified so that allocation is immediately performed on \( <I, s, 1> \) requests while the search is performed only on \( <I, s, 2> \) requests.) Each entry has a status count field which is incremented each time a request is received for that entry, and also a storage field for keeping a parked instruction (i.e., an opcode, two data fields for the instruction arguments, and a result spec.) The IPC Unit then sends \( <I, s, n> \) (and possibly some routing information) as a fetch request to the remote PE (fig. 6).

Upon receiving a fetch request \( <I, s, n> \), the remote IPC Unit decodes \( s \), performs the fetch from its operand memory, and returns \( <I, d, n> \), where \( d \) is the remote data. Upon receiving the returned package, the data is written into the \( n \)-th argument field of the parked instruction whose tag is \( I \), and the status count field of that entry is decremented and checked for zero. If zero, the parked inst-

\[ \text{Figure 6: An I/O Parking Operation} \]

\[ \text{Figure 7: An I/O Unparking Operation} \]
struction, assuming that it has already been received from OpFetch2, is released to back into the PIPU pipeline (fig. 7).

The structure of the IPC Unit parking store differs from the SMU parking store that the IPC Unit may only park a limited number of instructions. This is due to the high cost of embedding associative memory into a processor. Therefore, an abort mechanism is needed in the argument-fetching processor, where if a free parking store entry cannot be allocated for a particular instruction, that instruction is aborted and will be re-fired at some later time.

5 SUPPORT FOR DATAFLOW FIFO BUFFERS

The amount of parallelism that can be achieved in a dataflow graph has been shown to be dependent on the degree of balancing of the graph [6]. Thus, FIFO buffers for graph balancing are essential in a dataflow machine. These FIFO buffers are implemented as ring buffers in data or structure memory and provide detection of full, empty and non-full (or non-empty) conditions.

A FIFO buffer is represented in a p-instruction as an addressing mode either in one of the input operand specifications or the result operand specifications. A producer instruction p1 can write and continue writing to a FIFO buffer if it is not full and should be suspended when the buffer becomes full. Similarly, the consumer instruction p2 may continue reading as long as the buffer is non-empty and should be suspended when the buffer becomes empty. Additionally, p1 should re-activate p2 if the buffer is no longer empty, and p2 should re-activate p1 if the buffer is no longer full.

These conditions can be met by using the signaling capabilities of each actor in a dataflow graph. However, the normal two state condition code returned by the PIPU will not be enough, so an extended Z state is introduced for unconditional-list only signaling. The FIFO buffer returns:

- T if the buffer was empty before write (for n1)
  or if the buffer was full before read (for n2)
- F if the buffer is non-full and non-empty
- Z if the buffer is full after write (for n1)
  or if the buffer is empty after read (for n3).

Accordingly, n1, the signal node of p1, should signal only itself when the FIFO returns F and both itself and n2 if the FIFO returns T (so n1 will be notified that the buffer is no longer empty.) Similarly, n3 should signal itself only on a F condition code and both itself and n1 if the FIFO returns T. A Z received by either n1 or n3 will suspend the node since it will no longer signal itself to continue (fig. 8).

![Figure 8: A FIFO Buffering Example](image-url)

The condition codes returned by a FIFO buffer will override those returned by the nodes it is connected to. Therefore, it may not be directly connected to any node which uses its own condition code for conditional signaling, nor can more than one FIFO be connected directly to a node. In these cases, identity actors must be provided as a buffer between these instructions and the FIFO buffer.

6 THE ASSEMBLY LANGUAGE AND INTERPRETER ENVIRONMENT

As part of the continuing research of the Advanced Computer Architecture and Program Structures group at McGill University, an assembly language, an assembler and an instruction set interpreter for the new architecture [11,12] have been designed and implemented. Using the assembly language a-code, a user may directly specify the program tuple format of a dataflow graph. An a-code program may be assembled into a load image to be used by any interpreters or simulators for the argument-fetching processing element.

An a-code program is based on the following elements:

- Within a dataflow program, there are actions to be performed on data, the data itself, and the synchronization between the actions.
- An action may be a group of operations, such as the S-INDEX/STORE transaction, and is represented by a group of p-instructions collected in a node template.
- The synchronization between the instructions is represented by the list of signal arcs a particular node must wait for before it is able to fire. This list of signals is further separated at start-up time to the ones that the node must wait for before firing for the first time, and all the other signals it will also require for subsequent firings.
- The actions and synchronization of the actions together make up the basis of a program.
The Inner Product function in a-code
forall i in 0,ARRAYSIZE do
returns array of A(i) * B(i).

The data of a program may reside in either the operand memory or the structure memory of the PE.

Accordingly, an a-code program is composed of a collection of three types of segments: the PROG segment, which contains the node templates and its signal arcs, and the DATA and STRUC segments, which represent operand memory and structure memory of a PE.

After the design of a-code was completed, an integrated assembler/instruction set interpreter was constructed. The instruction set interpreter will be used to analyze program performance by generating parallelism profiles for that program and therefore may also be used to measure the effectiveness of the compiler or optimizer that generated the program. The interpreter also has rudimentary debugging capabilities such as the setting of breakpoints, the initialization and examination of memory locations, and execution traces. These capabilities may be used to debug user programs in the absence of an actual processing element on which to run the program.

Figure 9 shows a sample dataflow program graph and a portion of the a-code assembler representation of the graph.

7 SUMMARY

At McGill University, we are currently conducting a wide range of research on the static argument-fetching dataflow architecture. A number of preliminary results are reported in a recent workshop held at Montreal [9]. The structure memory and IPC operations will be investigated through the interpreter and its extensions. Dataflow compiler technology is also being studied in our group — in particular the software pipelining techniques of dataflow program graphs. Finally, a preliminary feasibility study of VLSI realization of the architecture is under consideration.

8 ACKNOWLEDGEMENTS

We thank Dr. Dennis for generously spending his time through the course of the past year, discussing the project with our research group during his trips to Montreal. We also thank the members of the ACAPS group at McGill and Herbert Hum at Centre de Recherche Informatique de Montreal for many interesting discussions.

REFERENCES


