CMU-CS-86-161

Compilation for a Processor Containing Pipelined Register Files

Alan Sussman
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

October 1986

Abstract

Pipelined register files are a novel form of processor storage found, for example, in the Link and Interconnection Chip (LINC) designed at CMU. The WarpJr processor uses the LINC pipelined register files as a major part of its intermediate storage capability. The processor is a microcoded machine that contains many resources that must be managed in every microinstruction, such as a floating point arithmetic unit, a data memory, LINC pipelined register files for several datapaths, and the LINC crossbar switch for routing data between datapaths. Past experience with programming such machines has shown the benefits that can be achieved from managing these resources with a compiler.

We concentrate on the code generator for the WarpJr processor, which manages all the low-level resources in the processor and attempts to generate efficient microcode quickly. The problems the code generator must solve are related to effectively utilizing the capabilities of the intermediate storage in the processor, namely the pipelined register files. Based on a set of benchmark programs, we present empirical data to demonstrate that we have achieved a satisfactory solution to all these problems, and can generate high quality microcode that effectively uses the pipelined register files in the WarpJr processor.

The research was supported in part by Defense Advanced Research Projects Agency (DOD), monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and Naval Electronic Systems Command under Contract N00039-85-C-0134, and in part by the Office of Naval Research under Contract N00014-80-C-0236, NR 048-659. The author was also supported by a grant from the General Electric Research and Development Center.

1. Introduction

The availability of VLSI implementation to universities and research laboratories has fostered experimentation with new processor architectures, and some of these architectures employ novel schemes to implement storage. Pipelined register files are an example of a novel storage structure. A pipelined register file is similar to a queue that allows read-access to all elements. Only the tail register of the queue can be written, and write operations have a side-effect on all registers. A write operation shifts the contents of all registers towards the head of the queue (and thereby overwrites the old value of the head register). Pipelined register files are used in custom VLSI architectures or available as off-the-shelf components.

All processor architectures contain local or temporary storage in the form of buffers, registers, queues, or latches. Experience with compilation of high-level language programs for such processors has shown the benefits that can be achieved from managing those resources with a compiler.

In this paper, we discuss the impact of pipelined register files on compilation for the WarpJr processor (and on code generation in particular). The WarpJr processor is an implementation of the Warp processor architecture designed at CMU [4], so it is designed to be a component of a systolic array. We present algorithms, along with their observed performance characteristics, to compile efficient microcode for the processor.

Pipelined register files are an attractive implementation of temporary or local storage. Since there is only one location that can be the destination of a write-access, a pipelined register file needs less control circuitry than a conventional general-purpose register file. This allows compact layouts and results in significant space savings. On the other hand, pipelined register files do not have the disadvantages of (strict) queues, which only support read-access to a single location. Pipelined register files allow random access to all elements in the file.

Most of the difficult tasks involved in compiling code for the WarpJr processor are a direct consequence of the presence of pipelined register files. Pipelined register files demand additional support from a language translator. A value remains in the register file only for a finite number of cycles. Each write-access to the the register file shifts the value one more position towards the head, and after a fixed number of write-accesses the value is lost. That is, the lifetime of a value in a register is not controlled explicitly by the compiler (as in a conventional register file), but is controlled implicitly by the number of write operations to the register file. As a consequence, the order of instruction evaluation is important; the compiler must attempt to reuse operands while they are still in

the register file.

The paper first gives an overview of the WarpJr processor architecture. Then we discuss the LINC architecture and the problems that are created by its capabilities. A complete treatment of the code generation problems encountered and the algorithms used to solve them follows. We present our empirical data on the performance of the code generator, and then shift our attention to compacting the LINC microcode. Finally, we discuss several issues related to integrating the WarpJr processor into a complete computer system and finish by presenting our conclusions on the impact of pipelined register files on compilation.

2. WarpJr architecture

The WarpJr processor is an implementation of the Warp processor architecture [4] that has been designed at CMU. As in Warp, the processor is designed to be a computational element in a systolic array. We call the processor WarpJr to suggest that it is a small, fast implementation of the Warp architecture. The processor architecture attempts to greatly reduce the chip count of the original Warp processor and is centered around the Link and Interconnection Chip (LINC), a custom VLSI chip designed at CMU.

A major goal of the design project is to investigate the effectiveness of LINC in a Warp-style architecture, by providing a small, reliable implementation of such an architecture. The datapath of the processor consists of a floating point arithmetic unit, 4 copies of LINC and a local data memory. Each input to the ALU is supplied with operands from one LINC pipelined register file, and the results from the ALU are fed back into the LINC to be switched to any desired datapath in the processor, including either input to the ALU. There are two major datapaths flowing through the processor, called X and Y, in addition to the address and systolic control code paths. A block diagram of the datapath of the WarpJr processor is shown in Figure 2-1. Although the word size of the processor is 32 bits, most physical datapaths in WarpJr are 16 bits wide, so 32 bit data transfers must be performed in two clock cycles.

The processor is similar in design to the Warp processor, with several important exceptions. The four copies of LINC [8] provide all datapath control and intermediate storage in the processor, replacing both the register queues and the PALs that implement the crossbar in the Warp processor. The LINC architecture is described in the next section. Another major difference from the Warp processor is that the floating point processor in WarpJr is a single ADSP-3220 [3], which performs a floating point (or integer) ALU operation in one clock cycle of approximately 100ns. Therefore, data pipelining in a WarpJr processor comes from the two stage pipeline in LINC, not from the floating point unit. The

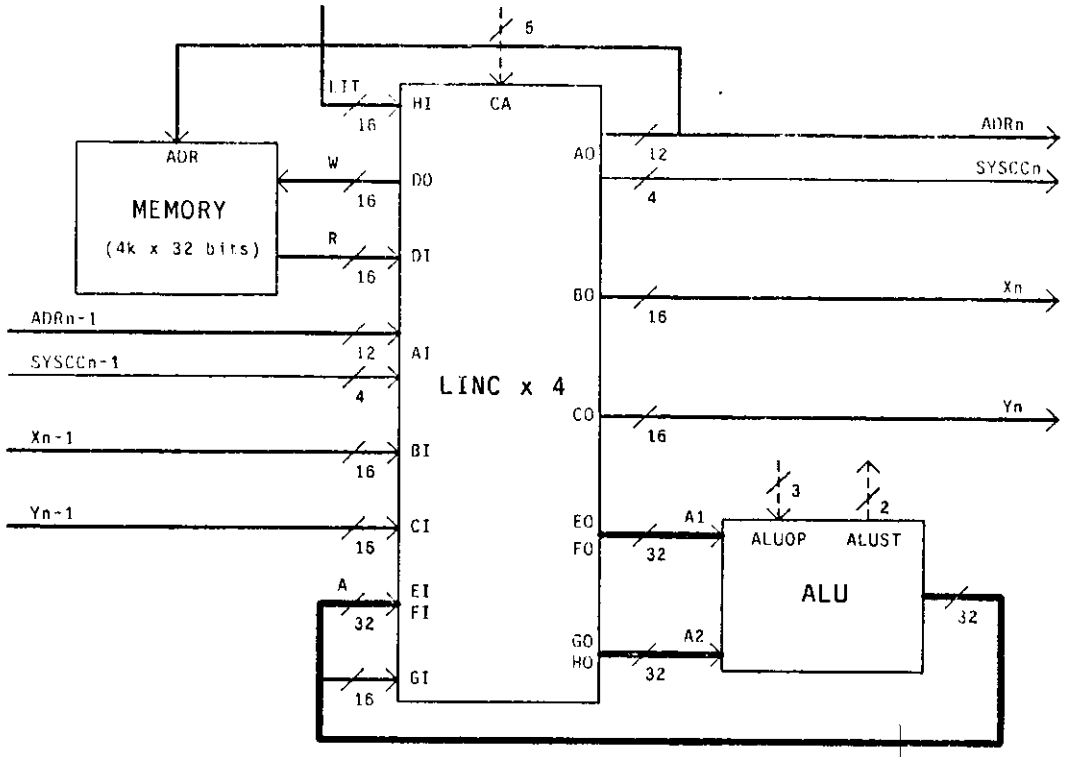

Figure 2-1: WarpJr Datapath

Warp processor contains the Weitek floating point chip set [12], consisting of a floating point multiplier and a floating point ALU, each of which is used as a five stage pipelined, 200ns clock cycle chip. The last major part of the datapath, which is the same in both processors, is a 4K×32 bit local data memory. This explains the need for the address (ADR) path through the processor. Overall, the major difference between the WarpJr and Warp processors is not in their architecture, but in the amount of hardware required to implement them. A Warp cell requires approximately 250 commercial components on a 15"×17" board. On the other hand, a WarpJr cell requires only about 75 chips, mounted on a small 6"×9" VME board.

Just as in a Warp processor, a WarpJr processor contains an AM2910 series microcontroller [2]. However, in WarpJr a microcode word is only 40 bits, less than 1/3 the size of a microcode word in Warp. This is slightly misleading though, because five bits of a WarpJr microcode word address a 64 bit LINC control pattern (the address is the same for each of the four copies of LINC in the processor). The best way to think of the WarpJr microcode is as a two-level microstore. The lower level consists of the internal LINC control patterns, and the upper level contains the cell microstore. Each instruction in the upper level includes an address for a LINC control pattern. The WarpJr writable control store contains 4K microcode words. These 4K WarpJr microcode words and the 64 word

control store on LINC provide the target for our code generation efforts for the machine. For a more detailed description of the WarpJr processor design, including application examples, see the WarpJr overview by Nishizawa [10].

Because the Warp and WarpJr processors are similar in so many ways, it seems reasonable to attempt to share as much work as possible in designing compilers for the machines. By the time WarpJr had been designed, much work had already gone into designing a high level language (W2 [6]) and building a compiler for the Warp processor (and the entire systolic array machine to be built with it). Fortunately, the compiler had been designed to generate intermediate code in a form suitable for code generation for any processor with capabilities similar to those of Warp, including WarpJr. Therefore, we decided to use the W2 compiler for Warp as a front end for the WarpJr compiler, and work on generating microcode from the intermediate code. This proved to be a good decision, because we were able to build our compiler in a reasonable amount of time and collect statistics on its performance, particularly on the performance of the LINC pipelined register files, without going through the difficulties of language design and building a new front end for the compiler.

3. LINC architecture and problem areas

Four copies of LINC provide all of the datapath control and the intermediate storage capability in the WarpJr processor. LINC is a custom VLSI chip designed at CMU whose sole function is to serve as an efficient link between system modules, like the ones in WarpJr. LINC allows system functional elements to communicate with each other through a crossbar switch, and can buffer each of its inputs in a FIFO or programmable delay and buffer each of its outputs in a pipelined register file. Interconnection configuration and register access control are performed by control patterns stored on the chip, which are addressed externally. In WarpJr, the LINC control patterns are addressed by the processor microcode.

The complete LINC architecture is discussed by Hsu [8], but we will give a short overview to provide enough information to understand the issues involved in compilation for a processor containing LINCs. The novel features of LINC include the integration of storage and switching on a single chip and the provision of an on-chip control store, both of which provide many interesting possibilities for new computer architectures. LINC has eight 4-bit datapaths, consisting of eight FPDs (FIFO and/or programmable delays), an 8×8 crossbar and eight pipelined register files. The datapath is a two stage pipeline, so there is a minimum delay of two 100ns clock cycles before an input can appear at an output port. A block diagram of LINC appears in Figure 3-1.

LINC has a data buffer at every input port, which can be configured as part of one of two FIFOs (of

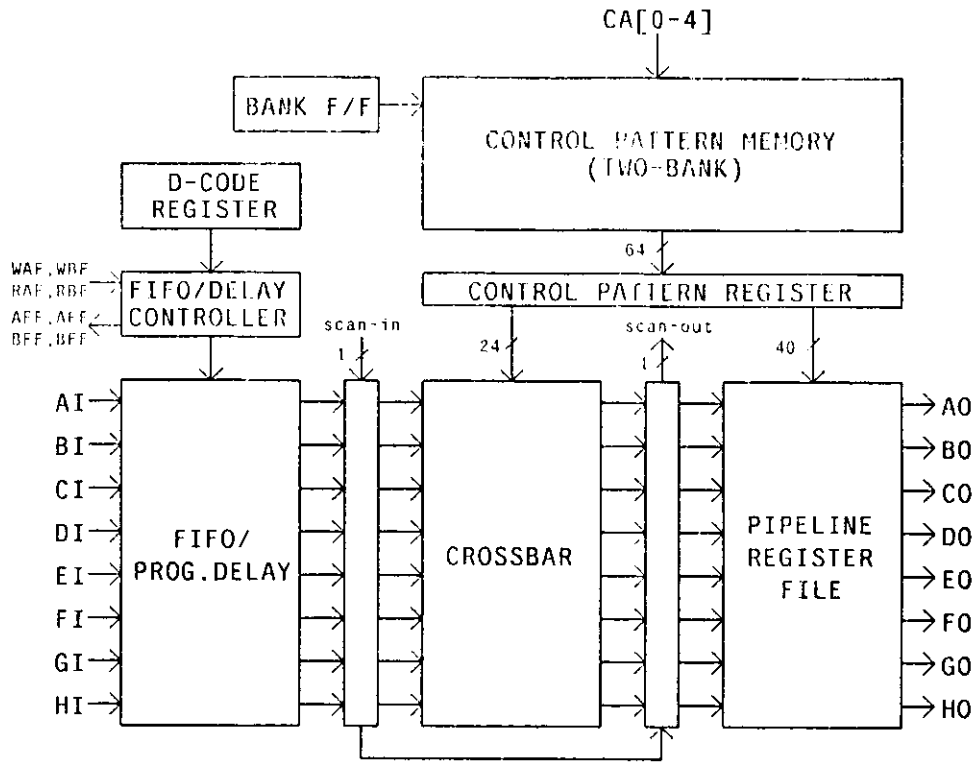

Figure 3-1: LINC datapath block diagram

maximum depth 31) or as a programmable delay of from 1 to 31 cycles. The configuration is stored in the d-code register depicted in Figure 3-1, and is intended to remain stable during program execution. The FIFOs are controlled by off-chip read and write signals, and similarly provide off-chip status signals (full and empty). While LINC can support two FIFOs, WarpJr only provides microcode control for one FIFO because each FIFO requires several bits of microcode every cycle. In the default configuration for WarpJr, all input buffers are configured as programmable delays of length one, except for the buffer corresponding to LINC input GI (see Figure 2-1) which is a programmable delay of length two.

In WarpJr, LINC input GI can be used in conjunction with input EI for sending 32 bit ALU outputs to 16 bit processor outputs X or Y over two clock cycles. This is accomplished by first routing the high-order bits from input EI through the crossbar to the desired crossbar output and on the following clock cycle routing the low-order bits from input GI to the same output. The additional delay of one cycle through the programmable delay associated with input GI allows the high and low order parts of a 32 bit result to both be routed to the same crossbar output even though the crossbar can transmit only one 16 bit value to an output in a single clock cycle.

An 8x8 crossbar connects the LINC input FPDs to the output pipelined register files. The crossbar is unidirectional, allowing each output port to select the input port it wishes to read on each cycle. The design allows broadcasting, since multiple output ports can read the same input port on a given cycle. Crossbar control is provided by the control pattern register, which is loaded from the control pattern memory.

Between the crossbar and each LINC output port is a 14-deep pipelined register file. A pipelined register file is similar to a queue that allows read access to all elements. Only the tail register of the queue can be written, and write operations have a side effect on *all* registers. A write operation shifts the contents of all registers towards the head of the queue, thereby overwriting the old value of the head register, as is shown in Figure 3-2.

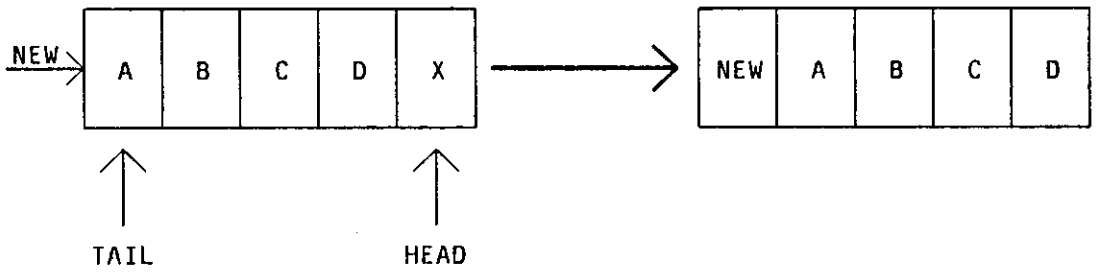

Figure 3-2: Write operation for pipelined register file

Another example of a pipelined register file is the AMD 29520 register file, which has a depth of two [2]. Any number of these parts can be cascaded to form a pipelined register file of the desired depth. The polycyclic architecture designed at ESL also incorporates a variant of pipelined register files in its interconnection elements [11].

Each LINC pipelined register file uses one bit in the control pattern register to decide whether to write the current crossbar output into the register file. Each pipelined register file output is specified by a four bit field in the control pattern register, and also serves as a LINC output port. The field specifies one of the fourteen pipelined register file stages, the crossbar output, or high-impedance. Therefore, LINC provides a *bypass mode* for its pipelined register files, because it is possible to route a value directly from the input port to the output port of the register file. The bypass path implements a "hot spot" - the input is valid for only one cycle. Of course, it is possible to use the bypass path *and* write the value into the pipelined register file.

The LINC control pattern memory and control pattern register determine the operation of the crossbar and pipelined register files. The memory contains 64-bit control patterns, configured in two banks of

32 words each. This means that one bank can be loaded with new control patterns while the other bank is controlling the system data flow. The limitations of 64 total control patterns and the bank size of only 32 control patterns are the source of two of the problems we must solve in compiling programs for the WarpJr processor. The control pattern memory requires a five bit off-chip address, provided in WarpJr by the processor microcode, which selects the control pattern from the bank currently controlling the chip. The control pattern register contains the pattern addressed in the previous clock cycle and controls the LINC behavior in the current clock cycle.

The work presented in the rest of the paper is mainly concerned with solving the problems presented in using the LINC pipelined register files effectively. Therefore, we must justify this effort by providing reasonably compelling reasons for using pipelined register files. A k-deep pipelined register file requires one bit of write control and a $\log_2 k$ bit read address every computation cycle. On the other hand, a k-deep general purpose register file requires two $\log_2 k$ addresses every cycle for both a read and a write address. In LINC, the savings in control logic was crucial to implement the design on a single chip. Since LINC has eight pipelined register files, each fourteen deep, we save 24 bits of control every cycle. The advantages of pipelined register files thus include requiring less control pattern memory, a lower control bus bandwidth, and a simpler control structure for the register cell than for a general purpose register file. While pipelined register files are not as powerful as general purpose register files, we shall see that it is possible to efficiently generate good microcode from a high level language for a processor containing pipelined register files (namely WarpJr).

Several major problems in compiling programs for the WarpJr processor are related to using the capabilities of LINC. These problems include using the pipelined register files efficiently and dealing with the limited size of the LINC control store. To use the pipelined register files efficiently, the code generator must use the pipelined register file bypass mode as often as possible to minimize the number of operands that must be written into the register files. Minimizing the number of operands written into the pipelined register files also implies that we need a good algorithm for assigning operands to the ALU input register files, because we don't want to write many operands into *both* ALU input register files. Another goal of the code generator is to minimize the number of pipelined register file overflows. A pipelined register file overflows if the value lost during a write is needed by a subsequent computation. Therefore an overflow is an expensive event, requiring us to recycle a value back into the pipelined register file that overflowed so that the computation can be performed.

The second major problem in using LINC in WarpJr is minimizing the number of control patterns that must be stored in the LINC control memory. Since the WarpJr microstore contains many more microinstructions than does the LINC control memory, we must attempt to use a LINC control pattern

for more than one WarpJr microinstruction. The LINC control memory is partitioned into two banks, one of which can be loaded while the other is controlling the system dataflow. Therefore, the basic requirement for managing the LINC control store is that no WarpJr program should need more than 32 LINC control patterns in any given sequence of microinstructions (the sequence length is determined by how long it takes to load a bank of the LINC control store with new control patterns). More formal statements of the pipelined register file problems and the LINC control store problem, along with the solutions we have implemented for WarpJr, are presented in later sections.

LINC pipelined register files have one important additional consequence in compiling programs for the WarpJr processor. The pipelined register files make it impossible to perform many global optimizations during code generation, because pipelined register files do not allow explicit register allocation. The major difficulty that pipelined register files introduce is that a value cannot remain in a fixed location in the register file for an indefinite length of time. A write to a pipelined register file is not to a register chosen by the programmer, but is always to the tail of the register file. In addition, the value in the head register of the pipelined register file is *always* lost by performing a write operation.

The compiler cannot do global register allocation, or many other global optimizations that require explicit control over registers. This limitation can be easily observed by looking at preserving values in pipelined register files across a basic block that is the body of a loop. Since the block can be executed many times (however many times the loop iterates), any write to a pipelined register file scheduled for the block is done multiple times, once for each loop iteration, resulting in the loss of values stored during previous blocks in the pipelined register file. A simple example of this behavior for a pipelined register file (called *R*) is shown in Figure 3-3.

```
store x into R
for i = 1 to N do
  begin

    other statements in loop

    store a[i] into R
  end
read x from R
```

Figure 3-3: Example of global register allocation problem

In the example, the position of *x* in the pipelined register file *R* for the execution of the read depends on *N*, the number of iterations of the loop. If *N* is greater than the number of registers in the register file, the value *x* is lost from the register file so cannot be read at all. More generally, calculating the position to read from after execution of a loop requires both counting the number of times the loop is

executed (which cannot be know at compile time for a *while* loop) and analyzing the pattern of writes done to the pipelined register file in the body of the loop. Therefore the only safe place in WarpJr to store data, so that it can be explicitly written to and read from a fixed address, is each processor's local data memory. Unfortunately, it is more expensive (in number of microinstructions executed) to access the data memory of a WarpJr cell than it is to access its LINC pipelined register files. We can still perform global optimizations that only require transformations on the intermediate code, such as copy propagation, code motion (to move invariant computations out of a loop), common subexpression elimination, etc. [1], but such transformations are outside the scope of this work.

4. Basic code generation/scheduling

The high level problem to be solved is to generate good microcode for the WarpJr processor. The input to the code generator (also called the scheduler) is a set of basic blocks with flow control between blocks embedded into the block information. The computation that the block performs is represented by a directed acyclic graph (a *dag*), with nodes corresponding to operations to be performed and edges representing the dependency relationships between operations (in our dags, a parent depends on the results of its children). The generation of the basic block structure and the computation dags for the blocks is performed by the front end of the W2 compiler [7], which performs local common subexpression elimination, constant folding, and several other local optimizations on the intermediate code (as represented by the dags for the basic blocks) before it is input to the WarpJr scheduler. As we have discussed, the presence of pipelined register files in the processor makes many global optimizations impossible because pipelined register files cannot store a data item indefinitely. The implicit register management strategy for pipelined register files compel us to generate code separately for each basic block. Any data that must pass between blocks is stored in the local data memory of the processor.

Three major problems must be solved to generate microcode for the processor. The first problem we call *register file assignment*, because we must decide which ALU input register file will receive each operand of each ALU operation. The problem occurs because the two ALU inputs in WarpJr are connected to different LINC pipelined register files, so if an ALU operation requires two operands they must reside in *different* ALU input register files. This problem must be resolved prior to scheduling the dag nodes for the basic blocks, so that during scheduling we can decide to which pipelined register files (and more specifically, to which ALU input register file or files) we must route operands. The second problem is to *schedule* the nodes in the dag for each block. The solution to this problem requires generation of a sequence of microinstructions (and LINC control patterns) for the block, and then the insertion of sequencing information at the end of each block to generate the correct flow

control between blocks. The last major code generation problem occurs in handling pipelined register file *overflows*. A register file overflow takes place when the value lost because of a register file write is needed as an operand by a operation that has not yet been evaluated. We are scheduling operations for a single basic block at a time, so we can tell during code generation when an overflow occurs, simply by noting that a required register file read depth is greater than the actual maximum depth of a pipelined register file (in the case of LINC the depth is 14). We will now more fully describe these problems and present the algorithms used to solve them.

4.1. Assigning operands to pipelined register files

In the WarpJr processor, each input to the ALU is associated with a different register file. A simple block diagram, showing only the ALU, its input pipelined register files and the switch for the feedback path, is shown in Figure 4-1, to further clarify the problem we must solve. We see that for operations requiring two operands, the ALU must receive inputs from two different pipelined register files, and the ALU result must be routed back through the switch to the appropriate register file(s) if the result is needed for further ALU operations.

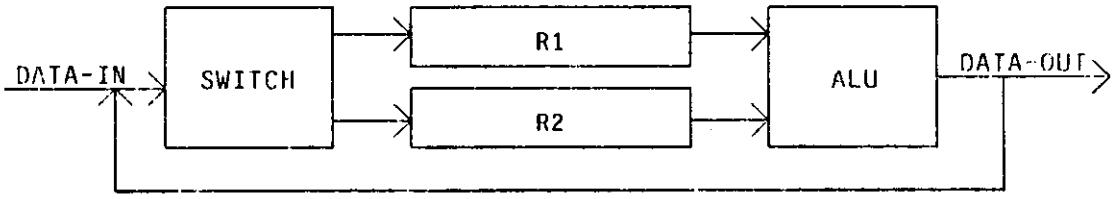

Figure 4-1: Simple processor model

We therefore have a register file assignment problem. We must assign each operand to an appropriate register file so that the two operands of a binary operation reside in different register files. For some sequences of operations it may be necessary to store a value in more than one file. Figure 4-2 shows a code sequence with its computation dag that requires a node to be allocated to both register files (in the dag edges point from top to bottom, and a parent node depends on the results of its children): Node X must be stored in both register files (or A1 and B1 must both be stored in both files).

One solution to the register file assignment problem is to simply direct all operands to both pipelined register files that are connected to the ALU inputs. However, this is not practical since the pipelined register files are only of limited depth. If too many operands are shifted into a pipelined register file, the oldest ones are lost. Therefore it is important to minimize the number of operands that must be

```
X ← A1 − B1;
A2 ← A1 * X;
B2 ← X * B1;
```

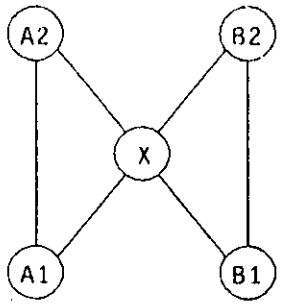

Figure 4-2: A dag that needs an operand in two pipelined register files

directed to both input pipelined register files.

> **Register file assignment problem:** Given a dag that must be evaluated on an ALU with two pipelined register files. Find an assignment of operands to register files that minimizes the number of operands that must be stored in both files.

The problem is equivalent to 2-coloring a graph, if we represent each operand as a node in the graph, and insert an edge between each pair of nodes that are operands of the same ALU operation. Then, if we can 2-color this graph, we can assign the operands to pipelined register files so that no operand is assigned to both register files. There are polynomial time algorithms to decide whether or not a graph is 2-colorable [5], but this is not sufficient for our needs. We require not only a 2-color decision algorithm, but also a scheme for assigning nodes to register files regardless of the result of the decision algorithm. The decision algorithm only tells us whether it is possible to 2-color a graph, not how to do it.

Good heuristics for coloring graphs are known [9], but are more complex than necessary to solve our particular problem. We will justify the choice of a simple algorithm later, in the performance analysis section of the paper. We have implemented an algorithm which traverses the graph recursively from the roots of each basic block, assigning nodes to register files following the simple heuristic of doing its best to assign the two children of a node to different register files. An operand may have multiple parents because of the existence of common subexpressions, so a node is visited exactly once for each of its parents. Therefore, the running time of the register file assignment algorithm is proportional to the number of edges in the computation graph for the program.

The recursive function **color**(*node*) is called for each root of each basic block. In the algorithm, the requirement is to color the two children of a single node opposite from one another.

Definition 1: Two nodes are colored *opposite* from one another if either they have been assigned different colors, or at least one of the nodes is assigned both colors.

Algorithm 1: Register file assignment.

Input. Set of computation dags, one for each basic block in the source program.

Output. The same computation dags as in the input, with all nodes colored so that for any node with two children, the children are colored opposite from one another.

Method. Recursive function **color**(*node*).

1. Count the number of children *node* has.

2. If *node* has no children **return**.

3. If *node* has 1 child and it is not already colored, color it arbitrarily, recursively visit the child **color**(*child*), and **return**.

4. *Node* has 2 children. There are 3 cases:

   a. If neither child is already colored, color *child1* and *child2* opposite from each other, recursively visit both children, **color**(*child1*) and **color**(*child2*), and **return**.

   b. If one child, say *child1*, is already colored, color *child2* opposite from *child1*, recursively visit child2 **color**(*child2*), and **return**.

   c. If both children are already colored, and they are not already colored opposite from each other, color one of the children both colors (thus assigning it to both register files), and **return**.

4.2. Scheduling

The scheduler generates microcode for one basic block at a time. The major goal of the scheduler is to generate good microcode for the WarpJr processor, mainly by using the bypass mode of the LINC pipelined register files as often as possible and doing its best to minimize the number of overflows from the LINC pipelined register files. The scheduler uses the flow control information provided in the intermediate code representation to generate the necessary sequencing control at the end of each basic block.

In general, generating optimal code for an arbitrary dag is a difficult problem. However, list scheduling has proved to be a good heuristic scheduling strategy, and we describe briefly our implementation. Scheduling a dag node is accomplished by allocating the resources the node requires. In the WarpJr processor, resources are entities such as the ALU, the various paths through the LINC crossbar, the LINC pipelined register files and the LINC output ports, and accesses to the data memory. Resources are represented by fields in the WarpJr microinstructions and in the LINC control patterns. To implement the list scheduling algorithm, we maintain both a ready list, R, and an

almost ready list, A, of nodes to be scheduled.

Definition 2: Dag node lists *ready* and *almost-ready* for the list scheduling algorithm.

The *ready* list, R, contains those nodes currently ready to be scheduled (ordered by some cost function).

The *almost-ready* list, A, contains the nodes n such that

- all children n_j of n are already scheduled
- the result of (at least) one child n_i is not yet available because of the internal delay of the processor.

Algorithm 2: Instruction scheduling: variant of list scheduling.

Input. Set of computation dags, one for each basic block in the source program.

Output. Sequence of microinstructions and LINC control patterns for the WarpJr processor that performs the computation specified by the input dags.

Method. List scheduling of the nodes in the dags, using lists A and R as defined above.

1. Select the node n in R with minimum cost; delete n from R. (If $R = \emptyset$, issue a no-op, update list R as in step 4, and try again.)

2. Reserve all the resources the node n requires.

3. Schedule the node n. If node n uses a result computed by child n_j, check whether bypass mode may be used. If not, compute the position of n_j in the pipelined register file at the time n is scheduled. If n_j has overflowed from the register file, fix the overflow using one of the methods described in the next section.

4. Check all the parents of n to see if any now have all their children scheduled. If so, add the parent to the almost-ready list A.

   Check the nodes of A and move a node p from A to R if the results computed by the children of p are available (i.e. all pipeline delays in the processor have been satisfied).

5. If there are still nodes left to be scheduled, go to step 1.

During the list scheduling algorithm we must dynamically keep track of the contents of the pipelined register files, so that we may generate the correct register file reads for the operands of a particular operation. Fortunately, because we are only scheduling for a single basic block at a time we can depend on linear flow of control, so we can be certain that the microinstructions for a block are executed in sequence. Therefore, to determine the depth of a pipelined register file read for parent

node n of its child node m, we find the microinstructions in which n and m were scheduled, call them μ_n and μ_m respectively, and count the number of shifts into the pipelined register file from μ_n to μ_m. That is the depth of the register file read for the operand.

We delay the decision to shift an operand into the register file until the parent nodes of the operand are scheduled, therefore we must update pipelined register file read depths when we decide to shift an operand into a register file. This is done by incrementing all reads of the pipelined register file in microinstructions *after* the new shift that are to data items in the register file shifted in *before* the new shift (thus they are pushed deeper into the pipelined register file by the new shift). Of course, in doing these increments we may detect an overflow (by incrementing a read past the maximum depth of a register file), and then we must recycle the lost value back into the register file.

Two key features make the list scheduling algorithm work particularly well in scheduling the basic block dags for WarpJr. The first feature is that even though the cost function determines the order in which nodes on the ready list are scheduled, the resources a node requires are allocated at the earliest possible microinstruction, after the node's operands have become available, that has all the free resources needed by the node. The second feature is that the bypass mode of the pipelined register files is used as often as possible, minimizing the number of writes to the register files. We can do this because we do not shift a result into a pipelined register file when scheduling the node that computes the result, but only shift in a result when an operation that *uses* the result is scheduled and cannot use bypass mode (i.e. the operation has been scheduled some time after the result arrived at the input port to the register file).

4.3. Overflows

The limited depth of a pipelined register file can be exhausted by the evaluation of some dags.

Definition 3: A pipelined register file *overflows* if the value lost in doing a write to the register file is needed by a node that has not yet been evaluated.

Overflows are a major problem for users of pipelined register files, since there is no way to maintain values in a pipelined register file after a fixed number of writes (and that number depends solely on the depth of the pipelined register file). Whenever an overflow occurs, the lost value must be recycled back into the register file, through one of the several mechanisms we will soon discuss. Thus we can state the overflow problem more formally:

> **Overflow problem:** Given a dag D and two pipelined register files of depth n connected to the inputs of a single ALU. Find a legal evaluation for the dag that

minimizes the number of overflows.

An overflow is detected during the list scheduling algorithm when a pipelined register file read is generated for an item that has been lost from the register file before the microinstruction containing the read (i.e. the calculated read depth is greater than the actual maximum depth of the pipelined register file). We have designed the register file assignment algorithm and the list scheduling algorithm to attempt to minimize the number of overflows from register files, because an overflow is costly in terms of the quality of the generated microcode. Unfortunately, finding a legal evaluation for a computation dag that minimizes overflows is not easy. The problem contains the register sufficiency problem, which is known to be NP-complete [5]. The best we can therefore currently hope for is that our heuristic solution will work well. As will be shown later in the performance statistics for the algorithms, our heuristics do indeed work well for the benchmark programs, generating very few overflows during the scheduling algorithm.

In WarpJr, there are at least three possible ways to handle the occurrence of an overflow during list scheduling of a dag representing a basic block. One solution, which is the most generally applicable solution to the overflow problem, is to modify the dag so that the node which overflowed from the pipelined register file is also written to WarpJr's local data memory, and then insert local memory reads into the dag so that the overflowed node is again read into the register file from which it overflowed. This requires that we throw away all the work that we have already done in scheduling the block, and start over again with the new, modified dag. The solution is general, because it will work in all instances where we detect an overflow, but can be expensive because of the need to restart our scheduling algorithm from scratch on the dag each time an overflow is detected.

A second method for handling an overflow is to attempt to backtrack and write the node that overflowed to the local data memory. This can be done by routing the node to the memory data input pipelined register file when the node arrives at the LINC crossbar and generating a memory write in the appropriate microinstruction. We can then insert a memory read between the microinstruction that writes the node to memory and the microinstruction that does the register file read that overflowed. Inserting these local data memory reads and writes into our already generated microcode requires that we find microinstructions containing enough free resources to implement the overflow solution. Since such resources are not always available in the restricted set of microinstructions in which we may allocate them, this solution suffers from the fact that it may fail, and if the solution succeeds it will cost two complete local memory cycles.

Yet another solution to the overflow problem is only easily applicable to overflows from the register files connected to the ALU input ports. We can recirculate the node that overflows through the ALU,

by performing an effective null operation on the value (i.e. add 0.0 to the value), and then route the result back to the same register file. We must schedule the recirculation ALU operation in some microinstruction between the one that initially shifted the node into the pipelined register file (so that the value is already in the register file to be recirculated) and the one that does the register file read that causes the overflow. We can then reschedule the operation that caused the overflow (the original register file read) and use the recirculated result as the operand in the register file that overflowed.

The major difficulties with the recirculation solution to the overflow problem are in finding microinstructions to schedule both the ALU operation for the recirculation, and in generating the second operand for that ALU operation (for WarpJr this would mean scheduling a literal with value 0.0 for an addition in the ALU). This is the solution that we have implemented in the code generator for WarpJr, and it has worked quite well in practice. The implemented strategy should not impose a great performance penalty on the generated code, because we only use resources that are unused by all previously generated code. However, fixing up an overflow uses resources that may have been used in scheduling subsequent nodes in the list scheduling algorithm, so may in reality somewhat affect the quality of the generated code. To extend the solution to pipelined register files that are not directly connected to the ALU input ports, such as the local memory data and address input pipelined register files, we would have to route the overflowed nodes to an ALU input register file and then use the same algorithm described above to recirculate the node back to the register file that actually requires the node. This would involve additional writes into the ALU input port register files, so probably is a poor solution to the general overflow problem.

5. Pipelined register file performance

We demonstrate that pipelined register files are an attractive implementation of local storage in a processor such as WarpJr. Because we cannot analytically prove any performance bounds on how well pipelined register files work, we will use a set of benchmark programs for which pipelined register files exhibit good behavior on the WarpJr processor, and argue both that these results are generally applicable to many programs compiled using our algorithms and that the results extend to other processor implementations using pipelined register files. For each benchmark, static performance data were collected either from the dags representing the computation or from the microinstructions and control patterns generated by the WarpJr scheduler. Dynamic performance data were obtained from examining the actual execution path of each benchmark program on a simulator, and determining how many times each microinstruction and control pattern were executed.

There are many issues involved in measuring the performance of the pipelined register files in the WarpJr processor. The performance of the register file assignment algorithm tells us how often operands are directed to both ALU input register files. A good register file assignment algorithm will assign few dag nodes to both register files. The scheduling algorithm attempts to use the bypass mode that LINC provides as often as possible, to save writes into pipelined register files. Therefore, measuring how often bypass mode is used should give us an idea of the utility of providing such a facility in a system that uses pipelined register files. The performance of the register file assignment algorithm and the use of bypass mode tell us how well we are minimizing the number of operands that are written into the register files.

While minimizing the number of operands to be written into pipelined register files is an important goal, the main issue we are interested in is the performance of pipelined register files when they are actually used to provide operands in evaluating computation dags. The *depth* of a pipelined register file read of node *n* can be determined by counting the number of values written into the pipelined register file from the microinstruction node *n* is written until the microinstruction node *n* is read (this includes the write of node *n*). There are many ways of evaluating the performance of the pipelined register files in WarpJr. The average and maximum depths of a pipelined register file read tell us how deep pipelined register files must be to successfully execute programs without excessive numbers of overflows. Another interesting measure of pipelined register file performance is the liveness of the nodes that are written into the register files. More specifically, we would like to find out for a given data item that is written into a pipelined register file how long it needs to remain available to be read (i.e. how deep in the register file the *last* read of a given value is). The last, and probably the most significant, performance measure for pipelined register files is how many overflows occur. This is really the true measure of how well pipelined register files perform, because until an overflow occurs a pipelined register file is just as powerful as a general purpose register file of the same size, the only difference being the implicit nature of a pipelined register file write location vs. the explicit decision to write to a given general purpose register.

We have selected a set of several medium-sized benchmark programs written in the high level block-structured language W2 [6] to measure the performance of the pipelined register files in the WarpJr processor. These include several low-level vision algorithms (binop, colorseg, pgen, rgbnorm and sobel), several signal processing algorithms (fft, conv and lup) and a program to compute a Mandelbrot set (mandel). In all that follows, static information is taken directly from the compiled microinstructions, without reference to program execution, while dynamic information refers to the microinstructions fetched during program execution.

5.1. General properties of the benchmark programs

To give an idea of the complexity of the benchmarks, we provide data on several properties of each program. The first complexity measure is the number of lines of W2 source code. The next set of data is information on the output of the front end of the W2 compiler, which generates the computation dags for each basic block in the source W2 program. The number of basic blocks and the total number of dag nodes required to describe the computation of the benchmark should give some idea of the size of the job that the code generator/scheduler for WarpJr must handle. Finally, we provide data on the number of microinstructions and LINC control patterns that the scheduler generates from the computation dags. This tells us the size of the actual program that runs on WarpJr. Table 5-1 contains the data on the general properties of the benchmark programs.

	W2 program (# lines)	# basic blocks	# dag nodes	# μinsts	# control patterns
binop	44	11	76	140	27
colorseg	106	31	227	374	58
conv	177	61	438	580	57
fft	34	13	91	154	37
lup	63	27	118	252	32
mandel	56	6	92	120	38
pgen	59	10	140	202	52
rgbnorm	112	51	248	426	25
sobel	144	79	278	580	31

Table 5-1: Properties of benchmark programs

5.2. Register file assignment

To measure how well our register file assignment algorithm works, we will compute both static and dynamic information from our register file assignment algorithm. The static numbers show the number of dag nodes that are assigned to both ALU input register files, as a percentage of the total number of dag nodes that must be assigned to the register files. The dynamic numbers show how many microinstructions route operands to both ALU input pipelined register files during actual program execution, as a percentage of the total number of microinstructions executed. Table 5-2 displays these measurements for the benchmark programs.

As can easily be seen from the table, very few nodes are assigned to both ALU input register files by our register file assignment algorithm. Also, during program execution, not many values are directed to both ALU input register files because of the register file assignment algorithm. This is a promising result, as it says that we are not routing many values to more than one pipelined register file, even

	static	dynamic
binop	2.9	0.0
colorseg	5.1	1.9
conv	0.0	0.0
fft	0.0	0.0
lup	2.0	0.0
mandel	5.4	6.2
pgen	13.2	1.8
rgbnorm	5.0	0.2
sobel	1.0	0.4

Table 5-2: Performance of register file assignment algorithm (in %)

though we do not have an optimal assignment algorithm. The good observed behavior of our assignment algorithm should enhance our pipelined register file performance in later measurements.

5.3. Bypass mode for pipelined register files

Realistic implementations of machines containing pipelined register files provide for a direct path from the input port to the output port of the register file. The purpose of this path is to allow direct forwarding of a value that is used only once, without shifting it into the register file. That is, the register file can be *bypassed*. However, the input is valid at the output port for only one machine cycle (the same one at which it arrives).

We would like to measure how well our list scheduling algorithm is able to take advantage of the presence of bypass mode in the LINC pipelined register files. There are two interesting performance measures that we can display for showing the effectiveness of bypass mode in minimizing the number of operands that are written into pipelined register files. The first is to measure how often bypass mode is used, both statically in the code generated by the list scheduling algorithm, and dynamically during the execution of the program generated by list scheduling. The relevant performance measure is the number of pipelined register file reads that use bypass mode, as a percentage of the total number of reads from the WarpJr pipelined register files. The second measure of how useful bypass mode is, is to look at how often the use of bypass mode actually saves a shift into a pipelined register file, again both statically and dynamically. Using bypass mode for a particular operand does not always save a shift into a pipelined register file, because the operand may be needed for *another* operation that cannot be scheduled without shifting the operand into the pipelined register file. Here, the relevant performance measure is the number of pipelined register file writes that are *not* done when bypass mode is used, as a percentage of the total number of uses of pipelined register file bypass mode. Table 5-3 shows these measurements for the benchmark programs.

	Bypass reads (as % of total reads)		Saved writes (as % of bypass reads)	
	static	dynamic	static	dynamic
binop	43.5	39.8	99.4	100
colorseg	36.3	35.9	96.5	93.7
conv	28.8	26.2	100	100
fft	38.9	41.0	98.0	98.0
lup	38.2	36.9	98.0	98.3
mandel	36.9	34.7	95.5	91.2
pgen	38.3	40.7	94.4	97.2
rgbnorm	37.3	36.6	99.6	99.7
sobel	40.5	41.8	100	100

Table 5-3: Bypass mode utilization (in %)

We see that bypass mode is used over 1/4 of the time, both statically and dynamically, in all the benchmark programs. This means that frequently a data item can be used as soon as it becomes available at the input to a pipelined register file. Even more importantly, in all the benchmark programs, over 90% of the uses of bypass mode (and frequently much greater percentages) actually save a write into a pipelined register file, thereby prolonging the liveness of all the data items already in a register file. This result indicates that few common subexpressions are found in the dags for the benchmark programs, because a common subexpression is used more than once as an operand, so must be written into a pipelined register file. The statistics provide a strong case for providing a bypass mode in any system containing pipelined register files, because bypass mode enhances the utility of pipelined register files by greatly decreasing the number of data items that must be written.

5.4. Depth of pipelined register file reads

The depth of a pipelined register file read for a particular data item tells us how many other values have been written into the pipelined register file from the time that the data item was written until the time that it is read. Therefore, the average depth of a pipelined register file read for a given processor provides information on how deep the actual register files should be so that as few overflows as possible occur. The average depth allows us to make an informed decision on trading off silicon area invested in register files vs. the need to minimize the number of overflows that occur. In addition, information on the maximum depth of a pipelined register file read for a particular register file in a processor provides us with worst case information as to how deep the pipelined register file would have to be to completely remove all overflows from a sample program.

We display the worst case depth information for the three major pipelined register files in the WarpJr

processor - the ALU input register files (grouped together), the register file that provides the address to the local data memory (called ADR-out), and the register file that provides input data to the local memory (called MEM-out). The data should *not* be taken as stating that if the worst case pipelined register file read is greater than the depth of a register file in WarpJr, we could not generate correct code for the program. The measurements were taken by disabling the overflow correction algorithm just to see what the worst case behavior of the scheduling algorithm would be.

Table 5-4 shows the average pipelined register file read depth, both statically and dynamically, over all the register files in WarpJr for the the benchmark programs, with overflow correction enabled. Overflow correction allows us to collect statistics on programs that can actually run on the WarpJr processor, because no pipelined register file overflows occur. The statistics do not include register file reads that use bypass mode, and therefore only measure register file reads that actually access the data from the registers in the pipelined register file. Table 5-5 shows the worst case pipelined register file read depth for the three register files in WarpJr described above, with overflow correction disabled as described above.

	static	dynamic
binop	1.85	2.04
colorseg	2.37	2.49
conv	1.60	1.65
fft	1.57	1.54
lup	1.29	1.26
mandel	1.45	1.37
pgen	2.27	2.12
rgbnorm	1.20	1.18
sobel	1.65	1.70

Table 5-4: Average pipelined register file read depth, not including bypass reads

Table 5-4 indicates that the average depth of a pipelined register file read, both statically and dynamically, is only about 2 deep in the register file. Once again, this is a significant result, because it says that we are almost always reading values that have not resided long in the pipelined register files, so are highly likely not to have overflowed. In addition, it means that we do not have to make the actual pipelined register files in the processor very deep, because we do not access elements deep in the register file often. The maximum depth results of Table 5-5 indicate that most programs *never* perform reads that are deep into the pipelined register file. The data provides additional evidence for the observation that most operands are not used often, and if an operand is used often it is used within a short sequence of microinstructions. The observation is directly analogous to the concept of

	ADR-out	MEM-out	ALU-in
binop	2	2	5
colorseg	13	2	16
conv	7	4	6
fft	2	1	7
lup	2	2	6
mandel	2	2	5
pgen	4	5	11
rgbnorm	1	1	3
sobel	3	1	4

Table 5-5: Maximum pipelined register file read depth

locality of reference in user programs, and is also a consequence of our local code generation strategy.

5.5. Liveness

To reinforce our argument that most operands are only used within a short sequence of microinstructions, we have computed the liveness properties of data items written into the WarpJr pipelined register files. In our terminology, liveness of an operand in a pipelined register file is measured by looking at the *last* time a particular data item is read from a pipelined register file relative to the time it was written into the register file, both in terms of the number of microinstructions executed since the item was written to the register file, and in terms of the number of writes to the register file since the item was written.

The liveness measurements provide us with information on the depth of the last read of a data item from a pipelined register file and on the number of microinstructions executed between the time a data item is written into a pipelined register file and the time it is last read from that register file. For these measurements, static numbers give liveness results as if each microinstruction were executed exactly once (thereby weighting each microinstruction equally in the average liveness measurement), while the dynamic numbers are obtained from the actual execution path of the benchmark program (which effectively weights each microinstruction by the number of times it is executed). Table 5-6 shows the average liveness of a data item in the WarpJr pipelined register files, both statically and dynamically, for the benchmark programs.

The measurements show that data items are not live for a long time in the WarpJr pipelined register files. On average, the last microinstruction in which a data item is read from a pipelined register file is shortly after the microinstruction in which the data item was written into the register file. In addition,

	Static		Dynamic	
	# writes	# μinsts	# writes	# μinsts
binop	1.4	4.0	1.5	5.0
colorseg	2.2	7.3	2.5	8.5
conv	1.4	2.9	1.6	3.3
fft	1.5	4.6	1.5	4.5
lup	1.4	5.0	1.2	2.9
mandel	1.4	3.0	1.6	2.8
pgen	2.2	8.9	2.1	9.5
rgbnorm	1.1	1.6	1.0	1.3
sobel	1.1	1.8	1.2	2.2

Table 5-6: Average liveness of a data item

there are few writes into the pipelined register file in that time interval. The measurements provide strong support for our contention that data items are usually used within a short sequence of microinstructions. The data also indicate that the scheduler for WarpJr does a good job of scheduling operations to use data items shortly after they arrive at a pipelined register file. Overall, the liveness data reinforce the notion that pipelined register files are an attractive form of register storage for a processor such as WarpJr, because pipelined register files, while not able to store operands indefinitely, are able to store operands long enough so that the operations that need them can be scheduled.

5.6. Overflows

Our major concern through all the performance measures on pipelined register files has been their impact on the number of overflows that will occur during program execution. We wish to minimize the number of overflows that occur because an overflow forces us to waste resources that are not explicitly required to perform the computation specified by the program. All of our algorithms were explicitly designed to minimize overflows, and now we will present the results of our efforts. Table 5-7 displays for the benchmark programs both the static number of overflows that occurred, as a percentage of the total number of microinstructions generated, and the dynamic number of overflows that would have occurred had we not eliminated them through our overflow resolution scheme, as a percentage of the total number of microinstructions executed.

The evidence is strong that overflows simply are not a major problem in the WarpJr processor with 14 deep pipelined register files. Overflows occurred in only one program (colorseg), and in that program all the overflows were resolved using the strategy described earlier, which, although not a general solution to the overflow problem, detracts very little from the performance of the compiled microcode.

	Static	Dynamic
binop	0	0
colorseg	1.1	1.9
conv	0	0
fft	0	0
lup	0	0
mandel	0	0
mandel	0	0
rgbnorm	0	0
sobel	0	0

Table 5-7: Overflow occurrences (in %)

With this evidence from the pipelined register files in WarpJr, and the maximum depth statistics stated above, we can say that pipelined register files do not have to be extremely deep to provide adequate support for compilation of high level language programs to microcode. More specifically, the 14 deep pipelined register files in the LINCs in WarpJr provide more than adequate support for basic block at a time microcode generation, while providing great savings in silicon area and register control logic.

6. Microstore compaction

When we speak of compacting the microcode for the WarpJr processor, we mean that we wish to minimize the number of control patterns for the LINCs in the processor. This is because there are only 64 control pattern locations on a LINC, while there are 4K control store words directly accessible to the microcontroller which, in addition to providing an address for the LINC control pattern, provides control and sequencing capability for the entire processor.

In the WarpJr design, it is intended for LINC control patterns to be used many times. The control pattern memory in LINC is divided into two banks of 32 words each, which allows us to control the datapath with one bank while at the same time we can load patterns into the other bank. This implies that at any particular time during execution of a program, the program should have a working set of at most 32 control patterns, so that we can be preparing for future computations by loading another set of 32 control patterns into the other bank of the control memory. Switching between control banks frequently is not desirable, because each switch costs one cycle, during which no control pattern is accessed. This means that for each switch between control banks we lose one computation cycle in the WarpJr cell, essentially waiting for LINC to switch control memory banks. Therefore we would like for any program loop to execute completely from one control bank of 32 patterns, to avoid having to switch banks at every iteration of the loop.

We consider LINC control pattern compaction to be a postpass phase of our code generation algorithm, because it would be difficult to attempt both to minimize the number of LINC control patterns and generate overall good microcode at the same time. We have therefore taken the approach of first generating the microcode, including the control patterns, and then attempting to reduce the number of control patterns. In general, for all the sample programs we have seen, the control patterns generated by the compiler are sparse, meaning that there are many fields which contain default values. We call these *don't care conditions*, because it does not matter to the user what the values of these control pattern fields are, since they will not affect the results of the computation to be performed. The sparseness of the control patterns generated is what will allow us to often merge multiple patterns into a single pattern to be used by many microinstructions. We will now describe the problem more formally and describe our algorithms and results.

6.1. The LINC control pattern compaction problem

The control pattern compaction problem can be stated formally as:

> **Compaction problem:** Given a set of control patterns, and a criterion for determining whether two control patterns may be merged, find the optimal sequence of merges to minimize the number of control patterns.

The LINC control patterns provide datapath control for the user program. A LINC control pattern in WarpJr consists of three fields for each of the eight datapaths in the processor. The three fields are:

- a crossbar instruction - to select for each crossbar output the input to be directed to that output,
- a shift instruction - to decide whether or not to shift the crossbar output into the pipelined register file for the datapath,
- a pipelined register file output instruction - to select the register in the pipelined register file which is read and directed to the LINC output port.

The crossbar and pipelined register file output fields often receive default values in the generation of microcode (including control patterns) during execution of our list scheduling algorithm. These default values may therefore be changed to any arbitrary value, and will not affect the correctness of the microcode. If a crossbar field is set to the default value, this means that the value is neither sent to the pipelined register file output nor shifted into the register file. Therefore, we can set the field to any desired value without affecting the computation being performed. If a pipelined register file output field is set to a default value, the output data is never used once it leaves the LINC output port. Again, we can set the field value to whatever we desire, without affecting the computation being performed. In contrast, a pipelined register file shift field can never receive a default value, because

the decision to shift a register file must be explicitly made for every datapath in every microinstruction to ensure that the contents of the register file remain in the correct registers for later reads from the register file. The key notion is that a pipelined register file shift affects the position of all values in the register file. Therefore, we can never change the value of a pipelined register file shift field in a LINC control pattern, because if we do we may execute an erroneous computation.

Given the requirements we have just discussed, we can state the conditions under which two LINC control patterns, call them *pat1* and *pat2*, may be merged into one pattern. The conditions for merging two patterns are as follows:

- The pipelined register file shift fields of *pat1* and *pat2* must match exactly (either both shift or both not shift) for each datapath.
- The crossbar fields and pipelined register file output fields of *pat1* and *pat2* must match for each datapath. In this case, a match occurs if either both patterns contain the same value for the field, or at least one pattern contains a default value for the field.

The new pattern *pat3* will contain:

- For the pipelined register file shift field, *pat3* will get the same value for the field that both *pat1* and *pat2* contain, for each datapath.
- There are two cases for the value of the crossbar and pipelined register file output fields in *pat3*. If *pat1* and *pat2* both contain the same value for a field (including the default value), *pat3* gets that value for the field. If one of *pat1* and *pat2* contain the default value for the field, *pat3* gets the non-default value for the field from the pattern not containing the default value.

Of course, if we merge *pat1* and *pat2* into one new pattern *pat3*, any WarpJr microinstructions which address either of the two merged patterns must be updated to address the new pattern. If two control patterns satisfy the above conditions so that they may be merged, we call them *compatible*. From the discussion above, we can see that if we only merge compatible LINC control patterns, we will not affect the computation performed, but will decrease the total number of control patterns by one each time we merge two control patterns.

6.2. An algorithm for merging control patterns

Given a set of control patterns, we would like to find the sequence of merges which minimizes the total number of control patterns and provides the same functionality as the original set of patterns. Unfortunately, the problem is equivalent to the clique partition problem for graphs, which is NP-complete [5]. This is easily seen by modeling each control pattern as a node in a graph, and inserting an edge into the graph between each pair of control patterns that are compatible. Then the problem

of minimizing the number of control patterns is equivalent to finding the minimum number of cliques which cover the graph. Since it is highly unlikely that we will find an optimal algorithm that runs in polynomial time to solve the control pattern merging problem, we must find a good heuristic for deciding which control patterns to merge.

The algorithm that we have implemented is quite simple. It makes a single pass through the array of control patterns, and for each pattern p attempts to find an already considered (and possibly merged) pattern q to merge with. In the original array, each control pattern is used by exactly one microinstruction, because that is how the list scheduling algorithm generates the microcode. The search for a pattern q is done in reverse order, from pattern p to the first control pattern in the array. This helps to ensure that a pattern gets merged most often with ones that were close to it in the original set of control patterns. The algorithm runs in time $O(n^2)$, where n is the number of control patterns we start with (and also the number of microinstructions). This ensures reasonable running times for the algorithm on programs compiled for WarpJr, which has at most 4K microinstructions. The results of applying this simple algorithm to the set of benchmark programs described in the section on pipelined register file performance are presented in Table 6-1.

	initial # of patterns	compacted # patterns	average times pattern used
binop	140	27	5.2
colorseg	374	58	6.4
conv	580	57	10.2
fft	154	37	4.2
lup	252	32	7.9
mandel	120	38	3.2
pgen	202	52	3.9
rgbnorm	426	25	17.0
sobel	580	31	18.7

Table 6-1: LINC control pattern compaction

The statistics clearly support our contention that the control patterns generated by the list scheduling algorithm are sparse, allowing us to merge a large number of initial patterns into many fewer patterns that must be stored in the LINC control pattern memory. For the benchmark programs, we were able to compact the patterns generated by the list scheduling algorithms so that they always fit into the 64 control patterns (in two banks) provided by LINC, even though the benchmarks include sample programs of fairly large size, generating several hundred microinstructions. The results imply that our compaction algorithm should work well even on much larger programs than the benchmarks,

because the algorithm appears to be doing a good job of minimizing the number of control patterns needed to execute programs.

Another concern we have discussed with respect to compacting control patterns is that no inner loop should require more than 32 LINC control patterns, because of the expense incurred in switching between banks during execution of a loop. The easiest way to look at the performance of the compaction algorithm in this respect is to look at the maximum number of merged control patterns used by *any* basic block in a compiled program. In general, the inner loops of a program perform most of the computation so they should generate the most microinstructions, and hence use more control patterns than any other basic block. In any case, looking at the worst case block can give us no better results than only looking at inner loops. Table 6-2 shows the greatest number of control patterns needed by a basic block for each of the benchmark programs.

	Max. # patterns
binop	19
colorseg	42
conv	45
fft	25
lup	27
mandel	18
pgen	23
rgbnorm	16
sobel	24

Table 6-2: Greatest number of control patterns for a basic block

In general, the results are encouraging. In all cases except two (colorseg and conv), the control patterns for the longest basic block in each program fit into one bank of the LINC control memory. The apparently costly exceptions for the benchmark programs colorseg and conv are really not as great of a problem as it may seem. The patterns for all basic blocks in colorseg and conv, except those for the inner loop, *do* fit into one bank of the LINC control pattern memory. The inner loop control patterns for both programs, which do not fit into one bank, do fit into both banks easily. The inner loop of colorseg requires 98 microinstructions to execute and the inner loop of conv requires 128, and we can partition the control patterns for either loop into two sets so that we only have to switch between banks once during execution of the inner loop. Therefore the overhead for switching banks in both cases is approximately 1%, so that we really do not pay that much of a penalty for not fitting all the control patterns into one bank.

The results of our compaction algorithm are quite good, and we do not think the effort and computational resources required to obtain better results will pay off in much better compacted sets of control patterns. The only blocks for which we have seen that our compaction algorithm fails to adequately minimize the number of control patterns, so that they fit into one bank of the LINC control memory, require many microinstructions. Therefore, so long as the patterns fit into *both* control pattern banks, the penalty paid for having to switch banks in the middle of a basic block is not all that great.

7. System issues

While the WarpJr processor is designed to be an element in a systolic array, we have only discussed problems related to compiling programs for a single processor. We have been able to use this approach because the high-level source language W2 is designed to allow the programmer to partition his problem onto the cells of a systolic array by writing a program for *each* cell (or perhaps using the same program on several cells). The language and its compilers do not attempt to solve the difficult problem of partitioning an arbitrary algorithm onto a parallel machine solely from its sequential description in a high-level language. There are, however, several problems related to compiling W2 programs for a systolic array machine that we must address to be able to generate working microcode The problems include generating local cell data memory addresses efficiently, generating cell loop control, and synchronizing data transfers between cells.

The Warp systolic array machine, from which the WarpJr machine was derived, consists of ten Warp processors connected in a linear array, a host machine which provides the user interface to the array, and an interface processor between the host and the processor array which provides data caching and address generation capabilities . The WarpJr processor was designed to interface cleanly to the same system configuration, and uses the same interface processor as Warp with only minor hardware modifications. The W2 compiler for Warp generates a host program, an interface unit program and cell programs for each processor in the array from a single complete W2 program.

We use the Warp W2 compiler to generate the program for the WarpJr interface unit, using the information the WarpJr scheduler produces from scheduling the code for the WarpJr processors. The interface unit provides both local data memory addresses and loop control information for the cells in the WarpJr array. Local memory addresses are not generated by the WarpJr processor cells themselves because the processors do not contain an integer arithmetic unit, and it is expensive to calculate addresses in floating point arithmetic and convert to integer addresses. Address generation by the interface unit is also efficient for algorithms that step through large data arrays on many cells,

because often an address can be used by more than one cell as it passes through the processor array. This saves calculating the same address several times, once for each cell. Loop control is done by the interface unit because it is expensive to maintain a loop counter in the cell, and loop control and address generation are closely related functions. The interface unit does have integer arithmetic capability, and can easily calculate the time at which loop control is necessary from the information provided by the WarpJr code generator.

The mechanism by which the interface unit program is created is quite simple. The WarpJr code generator, in addition to generating microcode for the WarpJr processors, generates the framework for the interface unit program by specifying several characteristics of the compiled microcode, including the size of each basic block (in number of microinstructions), the microinstructions in which local memory addresses are required (and the variable name associated with the access), and the microinstructions in which loop control are required. The Warp W2 compiler then uses this information to assign variables to data memory locations and produce an interface unit program that both generates memory addresses at the correct times (the ones at which the cell program expects them) and also generates the systolic control codes necessary to decide when the cell program should terminate a loop. The loop control scheme can be simple because W2 only allows simple counting loops, so the compiler can tell in advance how many times the loop will execute. With that information, and knowledge of the length of all basic blocks (in microinstructions), the Warp W2 compiler can easily create the interface unit program to generate both local data memory addresses and cell loop control.

The last major system problem we must address involves synchronization of data transfers between WarpJr cells. In the Warp machine, data queues provide the buffering capability between cells necessary to synchronize data transfers. The queues provide an ordering relationship between the data sends from one cell and the receives in the corresponding cell. In other words, the receiver acquires data in the same order that the sender transfers the data [4]. In WarpJr, the FIFOs in LINC provide properties similar to the queues in Warp. The FIFOs are written under control of the sending processor, and read under control of the receiving processor, which then routes the data through the LINC crossbar to the desired datapath and its corresponding pipelined register file.

Unfortunately, the WarpJr processor does not provide microcode control for the three FIFOs needed to implement the Warp architecture (X, Y, and address/systolic control), but only for one FIFO. In addition, the normal composition of the LINCs in WarpJr configures all the input buffers as fixed length programmable delays. Programmable delays force the relationship between a data send and the corresponding receive to be perfectly timed, in that the receiving cell *must* know the exact time

the data will arrive. Perfect timing is necessary so that the data can be routed correctly through the crossbar when it leaves the programmable delay a fixed number of cycles after it arrives. Programmable delays are not a viable solution to the problem of transferring data between cells, because they restrict the code generation problem so much that it becomes virtually impossible to compile efficient, correct code for multiple cells [4]. With programmable delays, every data receive must be exactly synchronized with the corresponding send, allowing the code generator little flexibility in scheduling operations efficiently. FIFOs are a much more desirable buffering mechanism, because of their flexible read/write capabilities. The reason for the differences in buffering capability between the Warp and WarpJr processors is that there were design changes in Warp, after the WarpJr design was complete, that were not carried over to WarpJr.

Another major problem with implementing the Warp architecture on the WarpJr machine is that each LINC only provides two FIFOs, instead of the three required. Currently, each LINC in WarpJr is programmed with exactly the same control patterns, with each of the four LINCs in a processor containing four bits of each sixteen bit datapath. The approach is simple, but not powerful enough to implement the desired architecture with three FIFOs. To successfully implement the architecture with FIFOs, the datapaths must be physically distributed across LINCs, with the three FIFOs realized by programming the d-code registers on each LINC differently and also suitably programming the control patterns in each LINC to conform to the datapath distribution. None of these difficulties would be apparent to the W2 programmer, but would complicate the task of the assembler which generates the actual microcode to be run in the processors.

There is another minor change to WarpJr that would help the compiler to generate efficient microcode. In the present design, the only method for implementing blocking data transfers between cells, using the LINC FIFOs, is for the programmer to check the FIFO full and FIFO empty status signals LINC provides and block accordingly. A hardware solution to the problem would be to automatically block a processor that tries to send to a full FIFO (or receive from an empty FIFO), until the FIFO is not full (or not empty). This hardware change would significantly simplify the problem of programming the WarpJr machine, both for humans and for the compiler, since the program would not have to explicitly check for the relatively infrequent occurrences of FIFO error conditions. The printed circuit board version of the Warp machine will provide this hardware capability.

8. Conclusion

Overall, we have established that pipelined register files supply good support for basic block at a time code generation for the WarpJr processor. In addition, we can state that global optimization during microcode generation would be difficult in any processor containing pipelined register files, because of the lack of explicit control over register allocation. We cannot perform global register allocation, which provides performance enhancements in many compilers, but we have shown that we can generate efficient code using only local pipelined register file allocation and a random access data memory for more explicit memory allocation. Pipelined register files exhibit many characteristics that make them easy for a compiler to manage, but also display features that cause great problems in compiling efficient code.

We have shown that pipelined register files are an attractive implementation of local storage. Pipelined register files create many constraints that the WarpJr code generator must enforce, but we have succeeded in finding reasonably simple heuristics to solve the problems pipelined register files create. We have effectively minimized the number of operands that must be written into pipelined register files, which greatly enhances their performance. This has been accomplished both by using the bypass mode of the LINC pipelined register files and by using a good heuristic to assign operands to register files. Our algorithms generate code with almost no overflow corrections, so that the pipelined register files are not losing large amounts of live data. Also, our LINC control pattern compaction algorithm does a good job of minimizing the number of control patterns that must be stored in the WarpJr processor LINCs, thus the small size of the LINC control memory is not a major difficulty in compiling programs. We can therefore state that while many of the abstract problems that arise in generating good code for the WarpJr processor are difficult, our heuristic solutions have proved to be powerful enough to generate efficient microcode.

References

[1] Aho, Alfred V., Sethi, Ravi, and Ullman, Jeffrey D.
Compilers: Principles, Techniques, and Tools.
Addison-Wesley, 1986.

[2] Advanced Micro Devices.
Bipolar Microprocessor Logic and Interface Data Book.
Advanced Micro Devices, Inc., 1985.

[3] Analog Devices.
High-Speed 64-bit IEEE Floating Point Multiplier and ALU
Analog Devices, 1985.

[4] Annaratone, M., Arnould, E., Gross, T., Kung, H.T., Lam, M., Menzilcioglu, O., Sarocky, K. and Webb, J.A.
Warp Architecture and Implementation.
In *Proceedings of the 13th Annual International Symposium on Computer Architecture*. June, 1986.

[5] Garey, M.R., and Johnson, D.S.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
W.H. Freeman, 1979.

[6] Gross, Thomas and Lam, Monica.
A Description of W2.
November, 1985.

[7] Gross, T., and Lam, M.
Compilation for a High-performance Systolic Array.
In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*. ACM, June, 1986.

[8] Hsu, F.H., Kung, H.T., Nishizawa, T., and Sussman, A.
Architecture of the Link and Interconnection Chip.
In Fuchs, Henry (editor), *Proceedings of the 1985 Chapel Hill Conference on Very Large Scale Integration*, pages 439-462. Computer Science Press, 1985.

[9] Leighton, Frank Thomson.
A Graph Coloring Algorithm for Large Scheduling Problems.
Journal of Research of the National Bureau of Standards 84(6):489-506, November-December, 1979.

[10] Nishizawa, T.
Overview of WarpJr.
April, 1986.

[11] Rau, B.R., Glaeser, C.D. and Picard, R.L. .
Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support.
In *Proceedings of the Ninth Annual Symposium on Computer Architecture*, pages 131 - 139. ACM, 1982.

[12] Woo, B., Lin, L. and Ware, F.
A High-Speed 32 Bit IEEE Floating-Point Chip Set for Digital Signal Processing.
In *Proceedings of ICASSP 84*, pages 16.6.1-16.6.4. IEEE, 1984.