A Variable Instruction Stream Extension to the VLIW Architecture

Andrew Wolfe and John P. Shen
Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
(412) - 268 - 3601

Abstract

A Variable Instruction Stream processor architecture called XIMD is proposed. The XIMD structurally resembles a VLIW and shares many of the beneficial characteristics of VLIW; however, the XIMD architecture can dynamically partition its resources to support the concurrent execution of multiple instruction streams. The number of streams can vary from cycle to cycle to best suit each portion of the application. The XIMD concept and a comparison with other traditional architectures based on state machine models of control paths are presented. Several program examples further illustrate the capabilities of XIMD. A brief description of an XIMD prototype machine is included; details of this implementation are presented in another paper.

1. Introduction

1.1. VLIW Processors

Horizontally microcoded computing engines have been used for many years as special purpose processors for well characterized, compute intensive applications such as digital signal processing. Very Long Instruction Word architectures, especially the ELI-512 project at Yale [Fisher83], extended and formalized these architectural concepts. By using advanced compilation techniques, fine-grain parallelism can be extracted from a wide range of general purpose and scientific applications. This opened the possibility of using horizontally microcoded processors as general purpose computers. The Multiflow TRACE/7 [Colwell87] was the first commercially available VLIW processor. These processors share several characteristics:

Multiple Functional Units: The processor can simultaneously execute a number of independent operations, which may include arithmetic operations, memory accesses, control operations and data movement.

Direct Execution of Application Code: The processor contains only a single level of control store. The functional units are explicitly controlled by independent fields in each wide instruction. The application is coded using these detailed instructions.

Single Code Stream: Although they perform many operations per cycle, these are essentially SIMD processors [Flynn66]. A single global controller selects a single instruction for execution each cycle.

These processors use straightforward, non-interlocking hardware to execute parallel operations. Pipelining is often used as well to enhance performance. Data and resource dependencies are always resolved prior to execution and explicitly controlled by the instructions. Extra hardware can be devoted to parallel data paths and multiple functional units rather than to large amounts of control and synchronization logic.

RISC processors have demonstrated excellent performance by implementing a computing model well suited to compiler optimizations. Similarly, VLIW processors adopt a number of features which simplify compilation for parallel architectures.

- Load/Store Architecture
- Register to Register, 3 Address Operations
- Large Global Register File
- Completely Orthogonal Instruction Fields
- Fixed Length Pipelines
- No Hardware Synchronization or Interlocking
- Architecture Completely Exposed to the Compiler

This inclination towards a fully deterministic, compiler-friendly architecture distinguishes VLIW processors from traditional horizontally microcoded processors. A powerful compiler is used to detect fine-grain parallelism and explicitly schedule multiple operations per instruction.

The term VLIW is commonly used to describe a variety of architectures which demonstrate many or all of the characteristics described above. Figure 1 is a model of a typical VLIW architecture. The datapath consists of a number of functional units connected to a global, multi-ported register file. Every functional unit is controlled by an independent set of instruction bits which also specify register addresses for the operands and results. The instruction also controls a separate instruction sequencer. The sequencer generates the address of the next instruction based on its current instruction fields and stored condition codes.

Although this model describes a typical VLIW architecture, significant variations have appeared on commercial and research machines. The Multiflow TRACE and Cydrome's Cydra-5 [Rau89] have implemented alternatives to a single global register file. Functional units may be homogeneous

such as in the machine proposed by Ebcioglu at IBM [Ebcioglu88] or they may be heterogeneous as in the TRACE.

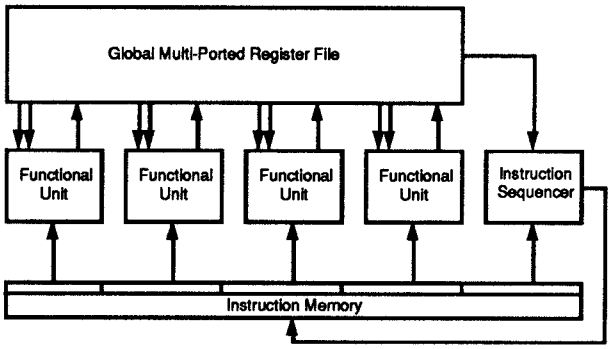

Figure 1: A Basic VLIW Model.

Several new microprocessors have extended RISC concepts to include VLIW-like operations as well. The Intel i860™ microprocessor [Intel89a] has a dual-instruction mode in which the chip acts as a small VLIW processor. The LIFE processor from Philips Research Labs [Labrousse90] is a more conventional VLIW processor on a single chip. Superscalar processors share many of the architectural features of VLIW machines such as multiple functional units and large register files, however they use complex instruction decoding and issuing hardware to extract operations which may proceed in parallel from a sequential instruction stream. Although these techniques first appeared in mainframe architectures [Tomasulo67], they have recently been applied to microprocessors as well [Intel89b, Bakoglu89].

1.2. Advantages of VLIW Processors

VLIW architectures exhibit several features which facilitate the use of powerful compilation techniques.

- The architecture is highly regular and instructions have few side effects. There are few restrictions on access to processor resources. This allows the schedule of operations to be manipulated with a great deal of freedom, thus allowing the compiler to explore a large number of possible optimizations in a short time.
- The effects of processor instructions are highly deterministic, including the timing of each instruction. The entire processor operates synchronously from a global clock. This permits the compiler to accurately predict a great deal of information about the run time execution of a program. This information allows the compiler to resolve data and resource dependencies at compile time, thus scheduling parallel operations without requiring run-time synchronization.

Instruction level parallelism, supported by multiple functional units, dense interconnect, and high instruction bandwidth, provide the high performance of VLIW processors. However, it is the powerful compilation techniques which take advantage of these features that allow VLIW processors to outperform other architectures for many applications.

In addition to the compilation techniques used in optimizing compilers for conventional RISC and CISC architectures [Aho86] and the vectorizing techniques used for supercomputers, a number of VLIW specific compilation techniques have been demonstrated. Trace Scheduling [Fisher81] was the first technique applied to scheduling code beyond basic blocks on VLIW processors. Percolation Scheduling [Nicolau85] is another technique proven effective for scheduling multiple operations per cycle from a variety of scalar codes. Software Pipelining [Ebcioglu87, Lam88] uses the semantics of program loops to tightly schedule repetitive operations.

1.3. Disadvantages of VLIW Processors

Although VLIW architectures perform well on many applications, their performance can degrade rapidly when faced with factors which decrease run-time predictability.

Control Flow Dependencies

The control flow of many programs is influenced by run-time data. This means that the actual schedule of operations is determined at run time through conditional branches. As data operations are removed from the critical path through parallel scheduling, control operations may begin to dominate execution time. Since a VLIW processor only contains a single program counter and branch mechanism, only one control operation can be executed each cycle. Attempts to eliminate or combine control operations can result in exponential code growth and can require prohibitively complex branching hardware.

Unpredictable Processor Interfaces

Processors often receive information from external sources. This interaction is often beyond the control of the compiler and thus unpredictable. This involves both truly asynchronous events such as external interrupts as well as bounded but still non-deterministic responses from memory and peripherals. A VLIW processor must always take a pessimistic, worst-case approach to unpredictable memory and peripheral behavior. This may involve stalling the entire processor when only one operation is delayed. Architectures which can dynamically schedule operations may respond better to these unpredictable occurrences.

Both of these issues limit the effectiveness of VLIW architectures on some problems. Both issues are also sensitive to increased parallelism within the processor. The goal of this research is to address these issues without forfeiting the advantages of VLIW architectures.

1.4. An Extension to the VLIW Model

This paper proposes an extended model of VLIW architecture called XIMD. XIMD stands for Variable Instruction Stream, Multiple Data Stream Processor. XIMD retains the capabilities of a VLIW architecture to effectively exploit fine-grain parallelism while adding control mechanisms to address the weaknesses of the VLIW model. XIMD can potentially exploit medium-grained and coarse-grained parallelism as well. Unlike a VLIW processor, which can only execute a single instruction stream, XIMD can split its functional units among one or more instruction streams. Furthermore, the number of instruction

streams can vary dynamically at run time based on a scenario generated by the compiler.

XIMD retains the ability of VLIW architectures to employ data parallelism while adding mechanisms to facilitate control parallelism as well. XIMD adds these additional capabilities with a minimal increase in hardware over a VLIW computer and still exhibits a clean, regular implementation. The XIMD Processor structurally resembles a VLIW processor and displays many of the same characteristics, such as a large global register file, multiple functional units, direct execution of instructions, and a fully exposed architecture.

Figure 2 shows a basic model of an XIMD machine. Note that the instruction sequencer has been duplicated for each functional unit. Condition code information must also be distributed to each of the sequencers. The individual control path sections may either operate identically, emulating the behavior of a VLIW architecture, or they may operate independently, as in an MIMD processor. There are also many interesting variations between these two extremes. By simply replacing the instruction address distribution hardware of a VLIW with duplicate sequencers and slightly increasing the complexity of the condition code selection hardware, we can significantly enhance the capabilities of the machine.

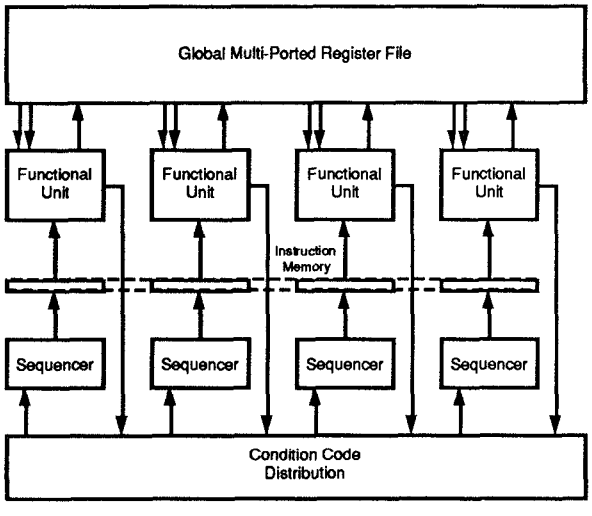

Figure 2: A Basic XIMD Model.

The addition of a separate sequencer to each functional unit essentially transforms each functional unit into an individual processor. Therefore, although the XIMD processor is structurally similar to a VLIW machine, it must be technically classified as a very tightly coupled MIMD processor. However, an XIMD processor can mimic either type of processor in operation. Additionally, an XIMD machine can vary its characteristics dynamically in a manner unlike either traditional architecture.

The ability to dynamically vary the number and functional unit width in each of the executing instruction streams from cycle to cycle differentiates the XIMD architecture from traditional MIMD architectures. XIMD can be implemented either as a shared memory machine or as a distributed memory machine. The distinction is not significant in understanding this architecture as memory is not used as a primary synchronization mechanism. XIMD uses fully distributed control with shared access to condition code information. There is no central controller and there are no master/slave relationships among processors. Each processor can provide information to other processors through data dependent or explicit management of condition codes; however no processor can control another processor. Since these condition codes are used to provide information to other processors about the state of a process they may be construed as a message passing mechanism, albeit one of very limited information capacity.

The XIMD architecture functionally behaves much more like a machine with centralized control than one with distributed control. The XIMD architecture operates under the assumption that all code is generated by an intelligent compiler, or at least written based on a set of well defined rules used by the compiler. The compiler either resolves dependencies at compile time or generates code to resolve dependencies between instruction streams using explicit synchronizations. Since the compiler is creating code for a generally deterministic exposed architecture, it implements synchronizations through cooperative behavior between instruction streams. In this manner the compiler itself acts as a central controller for the XIMD processor by limiting the run-time options available to each instruction stream. By forcing cooperation between processors, the compiler can maintain control without impacting run-time critical paths in the same manner as a centralized hardware controller. This permits efficient operation using limited hardware resources. Other architectures which include SIMD and MIMD modes of operation have been discussed by [Siegel81] and [Zaafrani90].

A limited precedent exists for adding multiple sequencers to VLIW architectures. The proposed Multiflow TRACE/500 architecture [Colwell90] contains two sequencers, one for each set of 14 functional units. The two sequencers can execute in lock-step or independently. This allows two processes to run concurrently when neither requires more than half of the machine. XIMD is a generalization and formalization of this concept, allowing very flexible variation in the number of instruction streams within a single program. These formal relationships are illustrated more clearly in the following section.

2. Architectural Models

2.1. State Machine Models

The functional model of XIMD is a straightforward extension of the classical state machine models of traditional architectures. Analyzing these models together, the relationships between them become clear. XIMD is a superset of several traditional architectures and thus can emulate the functionality of each.

A processor is customarily modeled as a union of control path and data path as shown in Figure 3. The control path can be modeled as a finite state machine which generates control signals for the data path and reacts to status signals from the data path. Modeling of the different types of control path structures can be used to differentiate classes of architectures.

The basic Moore machine is found in the control path of the model of a classical microprogrammed SISD uniprocessor shown in Figure 3. The functions and variables shown in the

diagram correspond to basic structures and mechanisms found in the processor.

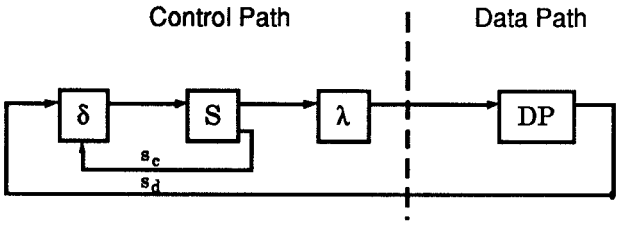

Figure 3: SISD Model.

S ↔ Micro-Program Counter

λ ↔ Microcode Memory (Data Path Control Fields)

δ ↔ Sequencer & Microcode Memory (Control Path Fields)

s_c ↔ Control Path State

s_d ↔ Data Path State (Condition Codes)

The function λ (the output function) depends only on state variable S, thus for a given value of the μPC, a given instruction will execute on the data path. The function δ (the next-state function) depends on the data path state s_d, the control path state s_c, and external inputs. Formally this function is described in terms of the complete control path and data path states. In practice, it is often adequate to use a simplified abstraction of these states, for example, a limited set of condition codes to represent the state of the data path.

A similar control path is shown in Figure 4, the state machine model of a VLIW processor. The VLIW processor has multiple functional units, each corresponding to the data path of an SISD processor. The VLIW model control path contains a separate output mapping function λ1 ... λn for each functional unit in the data path. The next state function δ must consider the state of each of the functional units, thus s_{d1} through s_{dn} as well as s_c are all inputs to this function. In an actual implementation this implies that condition code and status information from each functional unit feeds back to the instruction sequencer.

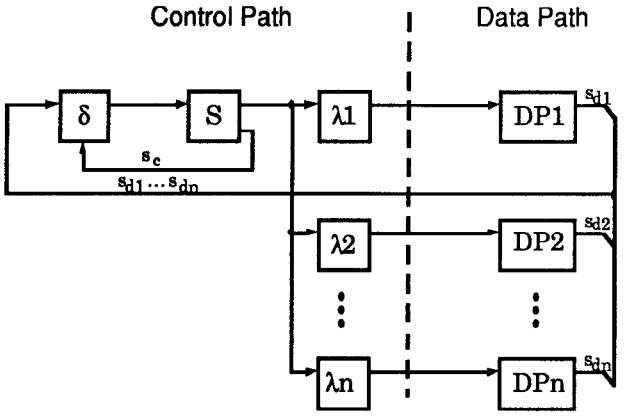

Figure 4: VLIW Model.

The traditional SIMD processor model is a simplification of the VLIW model. A traditional SIMD would distribute the output of a single function λ to each functional unit. It is easily shown that VLIW is a functional superset of SIMD. If for a given program the functions λ1 ... λn are identical and equal to the function λ of a corresponding SIMD machine, then the two machines are functionally equivalent. This implies that we can program a VLIW processor to emulate the functionality of an SIMD processor.

Figure 5 shows the model for an XIMD processor. When compared to a VLIW processor, the output functions, λ1 ... λn, and the functional unit data paths, DP1 ... DPn, are unchanged. The remaining portion of the control path has been duplicated for each functional unit. This results in separate state variables S1 ... Sn representing separate program counters. Also, separate next address functions, δ1 ... δn, exist to represent separate address generation and sequencing hardware. Just as the amount of state relevant to next address generation increased when additional data path units were added, the number of inputs to the δ functions must increase to include the state of each section of the control path.

Like the VLIW model, XIMD is capable of executing unique data operations on each functional unit. Additionally, XIMD may execute a unique control operation for each functional unit. This provides a greater range of functionality than the simpler architectural models. If for a given program, the functions δ1 ... δn are identical and the initial values of the state variables S1 ... Sn are identical, then the XIMD machine will be the functional equivalent of a VLIW machine.

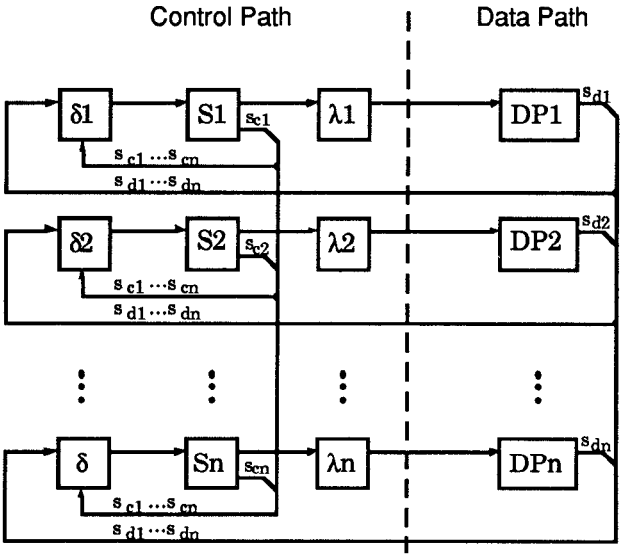

Figure 5: XIMD Model.

As before, it is often adequate in practice to only distribute an abstraction of the control path state rather than the actual values of all of the control path state variables. Furthermore, in an XIMD architecture a different abstraction of a state variable Si may be distributed to the local next address function δi than to the other next address functions.

Figure 6 shows a model of an MIMD processor. By selecting functions for δ1 ... δn which disregard the state of other functional units, XIMD can be a functional equivalent of this MIMD model as well. In a way XIMD can be viewed as containing the most general and capable control path design. Section 3 describes some of the unique operating modes of an XIMD architecture.

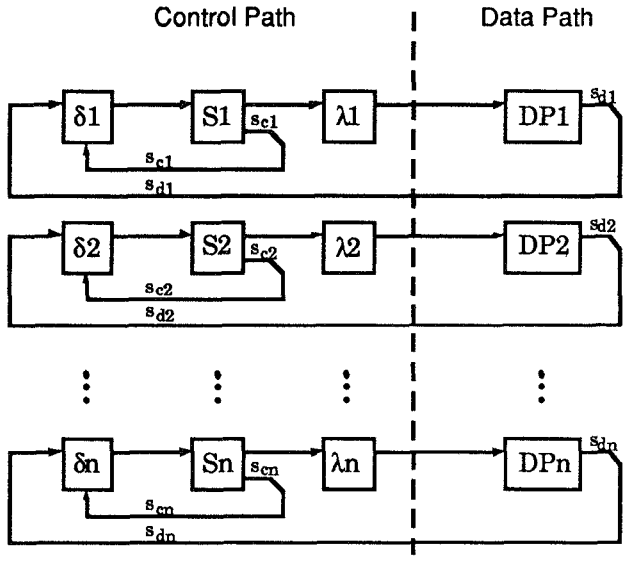

Figure 6: MIMD Model.

The XIMD model given is rather general. Various implementation decisions affect the range of capabilities available to the programmer. More specifically, three factors affect the complexity of the atomic control operations which can be executed by a given XIMD implementation.

- The number of values each control path state variable s_{c1} ... s_{cn} can assume.
- The number of values each data path state variable s_{d1} ... s_{dn} can assume.
- The complexity and programmability of each function δ1 ... δn.

The first two factors affect the amount of information which can be shared among instruction streams while the third factor determines how effectively one instruction stream can differentiate different situations occuring in other functional units.

2.2. XIMD Architectural Model

A baseline design of the general XIMD model has been selected for investigation. Even a minimalist XIMD design provides many interesting research topics and a variety of opportunities for new implementation and programming methods. This research model, called XIMD-1, is used for analysis and simulation in order to gain a broader understanding of the issues involved in a variable instruction stream architecture. It will continue to serve as a basic model for hardware implementation and further performance measurements.

The model contains 8 homogeneous universal functional units. These functional units can perform a wide variety of operations on multiple data types. Each functional unit is essentially capable of performing all of the operations of a RISC type processor, including loads, stores, and branches. Each functional unit can execute one data operation and one control operation (branch) per cycle. The control signals for each functional unit are supplied by a unique portion of the instruction memory. Each of these unique portions of the processor instruction is called an instruction parcel and is individually selected by a separate program counter. Since the functional units are homogeneous and have equal access to all processor resources, the compiler has complete flexibility in assigning operations.

Data Path

The data path consists of a global register file, eight data operators, and eight condition code registers. Each functional unit can perform a 3-address, register to register operation or a memory operation every cycle. All data operations complete in one cycle. Two data types are supported, 32-bit float and 32-bit integer. The register file simultaneously supports two reads and one write per functional unit for a total of 16 reads and 8 writes per cycle.

Each data operation consists of an opcode and three operands. These four fields describe the range of the output functions λ1 ... λ8 in the XIMD model. The three operands may be registers or constants. They are:

Operand	Symbol	Description
srca	a	Source Operand A
srcb	b	Source Operand B
dest	d	Destination Operand

Therefore an instruction of the form:

iadd srca, srcb, dest

Performs the operation:

$$a + b \rightarrow d$$

Figure 7 shows several examples of defined instructions. In addition to those shown, the common integer and floating point arithmetic, logical, and compare instructions are available. For the complete instruction set see [Wolfe89].

Integer Arithmetic

Opcode	Function
iadd	a + b → d
isub	a - b → d
imult	a * b → d
idiv	a / b → d

Memory Operations

Opcode	Function
load	M(a + b) → d
store	a → M(b)

Figure 7: - Example Instructions

One data operation can be scheduled for each functional unit every cycle without any restrictions. Each functional unit also contains one condition code register, CC_i. This register can hold one of two values, TRUE or FALSE. Compare operations set or clear the condition code register corresponding to the functional unit which executes the operation. Other operations leave the condition code register unchanged. The value of the condition code register CC_i represents the variable s_{di} in the XIMD model.

Control Path

Each functional unit contains a separate program counter, PC_i. This register corresponds to the state variable Si in the XIMD model. Each PC is used to select the instruction parcel for the corresponding functional unit each cycle. This instruction parcel is comprised of the data path control fields described above and the control path control fields shown in Figure 8. The next state function δi is implemented by the control path control fields, the branch target selection multiplexer, and the condition code evaluation function, implemented here as a PAL [Palmer90].

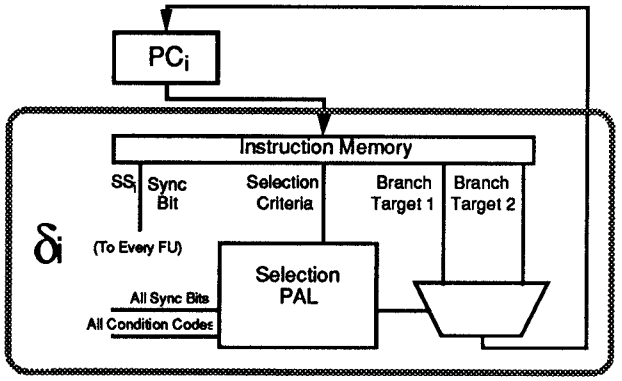

Figure 8: Control Path for each FU.

The control path control fields include two branch targets, T1 and T2, allowing the next instruction to be chosen from two explicit choices. There is no incrementer for the PC. The synchronization signal SS_i represents an abstraction of the state variable Si. It is distributed to the other functional units for use in process synchronization. It is a two valued signal. The values are arbitrarily named BUSY and DONE. The condition selection criteria field determines how to combine and evaluate the condition codes and synchronization signals from all of the functional units to choose from the two branch targets. Several simple control operations are defined for XIMD-1. This set of available control operations basically describes the next state function δi shown in the XIMD model. Two unconditional control operations have been included:

Target 1	next instruction address = branch target 1
Target 2	next instruction address = branch target 2

A number of conditional control operations have also been included. If the specified condition is true then the next instruction is executed from the address at branch target 1, otherwise branch target 2 is used.

Branch on (CC_i == TRUE)	Branch on one condition code
Branch on (SS_i == DONE)	Branch on one Sync signal
Branch on $\prod_i (SS_i == DONE)$	Branch on ALL Sync signals
Branch on $\sum_i (SS_i == DONE)$	Branch on ANY Sync signal

2.3. Scope of Current Research

For the current research, an idealized model is assumed for processor memory. A shared memory model is used. Each functional unit can read or write to memory every cycle. All ports use a single shared address space. Memory operations complete in one cycle. Multiple writes to the same location in one cycle are undefined.

The current XIMD research model, XIMD-1, is intended for use as a special purpose processor for compute intensive applications. As such, several critical issues which affect all general purpose parallel processors have not yet been addressed. These include interrupt and exception handling, virtual memory, and multiple user operation. In addition, a modern implementation will require data path pipelining and a simplified memory model to permit a short cycle time. These issues must be addressed prior to implementation. The simplified model presented is useful as a research tool to study the potential of XIMD architecture.

2.4. Definition of XIMD Terms

The interactions among functional units in an XIMD machine can become quite complex. The behavior of the processor changes dynamically among many processing styles. In order to precisely describe some of the unique features of XIMD it is necessary to define some specific terms and notations.

FU: Functional Unit.

Instruction Parcel: The set of instruction fields which control each FU. This includes the fields for the control path, data path, and synchronization signals for each FU. Each instruction parcel is independent. Eight instruction parcels comprise one instruction, whether or not they were issued from the same physical address.

SSET: A Synchronous Set of Functional Units or SSET describes a set of one or more XIMD functional units which are currently executing a single program thread. An SSET of functional units is indistinguishable from a VLIW processor of the same size. Formally, two functional units are in the same SSET at time t, if given the program and the control state of one FU, the control state of the other FU can be uniquely determined.

Partition: An XIMD processor can be operating as one or more SSETs. This separation of the complete set of functional units into SSETs is called a partition. An XIMD machine can change from one partition to another during program execution. This may or may not change the number of executing threads.

{}{}: A set notation is used to describe a partition. The functional unit numbers of all members of an SSET will be placed in braces. The current SSETs are listed sequentially. For example: {0,1,2,3,4,5,6,7} describes the partition containing a single SSET and thus executing one instruction stream. {0,1}{2}{3,6,7}{4,5} describes a partition containing four SSETs and thus executing four instruction streams or program threads. {0}{1,2,3}{4}{5,6,7} describes a different partition also containing four SSETs.

3. XIMD Program Execution

3.1. Sequential Programs

The XIMD processor can function in a number of different modes of operation, duplicating the behavior of several traditional architectures as well as exhibiting some new behaviors. The programmer and the compiler select the best execution model for each section of the compiled application.

For sequential programs, that is programs without explicit parallel constructs or iterative loops, the VLIW execution model is an excellent selection. VLIW compilation techniques such as Trace Scheduling and Percolation Scheduling can extract instruction level parallelism from scalar code. This produces an efficient schedule of operations for a VLIW processor. This VLIW style program can then execute just as efficiently on the XIMD as on a VLIW machine.

TPROC

```
tproc(a,b,c,d)
  int a,b,c,d;
{
  int e,f,g;
  e = a + b;
  f = e + c * a;
  g = a - (b + c);
  e = d - e;
  return (a + b + c) + d + e + (f + g);
}
```

	FU0	FU1	FU2	FU3
00:	-> 01: iadd a,b,e	-> 01: imult c,a,f	-> 01: iadd c,b,g	-> 01: nop
01:	-> 02: iadd f,e,f	-> 02: isub a,g,g	-> 02: iadd e,c,a	-> 02: isub d,e,e
02:	-> 03: iadd a,d,a	-> 03: iadd f,g,g	-> 03: nop	-> 03: nop
03:	-> 04: iadd a,e,a	-> 04: nop	-> 04: nop	-> 04: nop
04:	-> 05: iadd a,g,f	-> 05: nop	-> 05: nop	-> 05: nop

Example 1: Scalar XIMD Code

Example 1 shows a small fragment of scalar code. A Percolation Scheduling compiler [Breternitz90] has been used to compile this procedure. The resulting schedule is shown in the code listing. Only four functional units are displayed for clarity. In order to execute a VLIW style program on the XIMD, the control path instruction fields must be duplicated in each instruction parcel, so that each functional unit will execute the same control. This requires duplicating the sequencer instruction, all branch targets, and all condition code selection criteria, as shown.

The fully synchronous VLIW-style execution model is applicable for many traditional vectorizable problems as well. Livermore loop 12, shown below, is an example of such a program. Software Pipelining can be used effectively to schedule multiple iterations of this loop in parallel.

```
      DO 12 k = 1,n
12      X(k) = Y(k+1) - Y(k)
```

Livermore Loop 12

Format Note: The XIMD instruction is displayed in several columns, one for each functional unit. Each row of boxes represents the instruction parcels stored at one instruction address. As shown in Figure 9, the top section of each box contains the control path operation for each functional unit, either an unconditional or conditional branch. Unconditional branches are coded in the form (-> 05:) meaning branch to address 05:. Conditional Branches are coded in the form (if cc1 02: | 03:) meaning if cc1 has the value TRUE then branch to address 02: else branch to address 03:. The bottom section contains the data path operation. Note that although instruction parcels for different functional units appear at the same address, each functional unit has a separate sequencer and thus they might not execute from the same physical address at the same time. Assume that in every example program, all functional units begin execution together at address 00:.

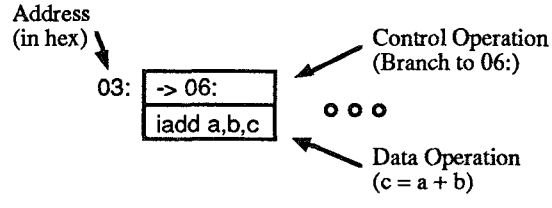

Figure 9: Example Code Format

3.2. Loops with Branches - Fork/Join Constructs

Program loops containing branches are difficult for VLIW compilers to schedule and VLIW processors to execute. XIMD can use a variety of multiple instruction stream methods to address these problems. One of the simplest cases is a program where all branch paths within the loop have the same length. The program in Example 2 can be forced into this form by padding short paths with `nop` such that every branch path has the same length.

The program shown in Example 2 searches the integer array `IZ()` for the minimum and maximum value. These two searches

MINMAX

```
   max = minint
   min = maxint
   DO 99 k = 1,n
     IF (IZ(k).LT.min) min = IZ(k)
     IF (IZ(k).GT.max) max = IZ(k)
99 CONTINUE
```

	FU0	FU1	FU2	FU3
00:	-> 01: load #z,#0,tz	-> 01: iadd #1,#0,k	-> 01: lt n,#2	-> 01: iadd n,#0,tn
01:	if cc2 08: I 02: lt tz,#maxint	if cc2 08: I 02: gt tz,#minint	if cc2 08: I 02: nop	if cc2 08: I 02: isub tn,#1,tn
02:	-> 03: nop	-> 03: nop	if cc0 04: I 03: eq k,tn	if cc1 04: I 03: nop
03:	-> 05: load #z,#k,tz	-> 05: iadd #1,k,k	-> 05: nop	-> 05: nop
04:			-> 05: iadd tz, #0, min	-> 05: iadd tz #0, max
05:	if cc2 08: I 02: lt tz,min	if cc2 08: I 02: gt tz,max	if cc2 08: I 02: nop	if cc2 08: I 02: nop
08:	-> 0a: nop	-> 0a: nop	if cc0 09: I 0a: nop	if cc1 09: I 0a: nop
09:			-> 0a: iadd tz, #0, min	-> 0a: iadd tz, #0, max
0a:	-> 0a: nop	-> 0a: nop	-> 0a: nop	-> 0a: nop

k, n, tn, tz, min, and max are register names.
z, minint, and maxint are constants

z is the address of IZ(1)
minint is the smallest representable integer
maxint is the largest representable integer

Example 2: Implicit Barrier Synchronization

BITCOUNT1

```
B[0] = 0;
for (i = 1; i < n; i++) {
  b = 0;
  while (D[i]) {          /* Count 1's in D[i] */
    if (D[i] & 0x01) b++;
    D[i] >>= 1;
  }
  B[i] = B[i-1] + b;      /* B[i] is cumulative
*/
}
```

	FU0	FU1	FU2	FU3
00:	-> 01: le n,#8 DONE	-> 01: iadd #1,#0,k DONE	-> 01: iadd #0,#0,b DONE	-> 01: store #0,#B0 DONE
01:	if cc0 30: I 02: nop DONE	if cc0 30: I 02: nop DONE	if cc0 30: I 02: nop DONE	if cc0 30: I 02: nop DONE
02:	-> 03: iadd #0, #0, b0 BUSY	-> 03: iadd #0, #0, b1 BUSY	-> 03: iadd #0, #0, b2 BUSY	-> 03: iadd #0, #0, b3 BUSY
03:	-> 04: load #D0,k,d0 BUSY	-> 04: load #D1,k,d1 BUSY	-> 04: load #D2,k,d2 BUSY	-> 04: load #D3,k,d3 BUSY
04:	-> 05: eq d0,#0 BUSY	-> 05: eq d1,#0 BUSY	-> 05: eq d2,#0 BUSY	-> 05: eq d3,#0 BUSY
05:	if cc0 10: I 06: and d0,#1,t0 BUSY	if cc1 10: I 06: and d1,#1,t1 BUSY	if cc2 10: I 06: and d2,#1,t2 BUSY	if cc3 10: I 06: and d3,#1,t3 BUSY
06:	-> 07: eq #0,t0 BUSY	-> 07: eq #0,t1 BUSY	-> 07: eq #0,t2 BUSY	-> 07: eq #0,t3 BUSY
07:	if cc0 04: I 08: shr d0,#1,d0 BUSY	if cc1 04: I 08: shr d1,#1,d1 BUSY	if cc2 04: I 08: shr d2,#1,d2 BUSY	if cc3 04: I 08: shr d3,#1,d3 BUSY
08:	-> 04: iadd b0,#1,b0 BUSY	-> 04: iadd b1,#1,b1 BUSY	-> 04: iadd b2,#1,b2 BUSY	-> 04: iadd b3,#1,b3 BUSY
10:	if ∏dn 11: I 10: nop DONE	if ∏dn 11: I 10: nop DONE	if ∏dn 11: I 10: nop DONE	if ∏dn 11: I 10: nop DONE
11:	-> 12: iadd b,b0,b DONE	-> 12: nop DONE	-> 12: iadd k,#B0,a DONE	-> 12: nop DONE
12:	-> 13: iadd b,b1,b DONE	-> 13: store b,a DONE	-> 13: iadd k,#B1,a DONE	-> 13: nop DONE
13:	-> 14: iadd b,b2,b DONE	-> 14: store b,a DONE	-> 14: iadd k,#B2,a DONE	-> 14: isub n,k,t DONE
14:	-> 15: iadd b,b3,b DONE	-> 15: store b,a DONE	-> 15: iadd k,#B3,a DONE	-> 15: lt t,4 DONE
15:	if cc3 30: I 02: iadd k,#4,k DONE	if cc3 30: I 02: store b,a DONE	if cc3 30: I 02: iadd #0,#0,b DONE	if cc3 30: I 02: nop DONE
30:	Clean Up Code for less than 8 iterations remaining.			

k, n, a, b, t, b0-b7, d0-d3, and t0-t3 are register names.
B0-B3, and D0-D3 are constants

B0 is the address of B[0], etc.
D0 is the address of D[0], etc.

∏dn is the condition $\prod_{1}(SS_i == \text{DONE})$.

Example 3: Explicit Barrier Synchronization.

are performed in parallel. In each loop iteration, the program compares one array element to the variables `min` and `max`. Based on these comparisons, it then independently replaces these variables with the array element. If either replacement does not take place, a `nop` is executed instead to maintain the path length. Once again, only four functional units are shown for clarity.

This program will execute as follows:

- All FU's begin executing together at address 00:
- The functional units continue to execute a single instruction stream for three cycles. They have fetched the first array element and compared it to minint and maxint.
- After Cycle 2, the program forks into three threads.
  - FU0 and FU1 continue to 03:
  - FU2 conditionally branches to 03: or 04: based on cc0, the comparison to maxint.
  - FU3 conditionally branches to 03: or 04: based on cc1, the comparison to minint.

  Since the relative states of these three program threads is data dependent, the current partition contains three SSETS, even if every FU happens to execute the instruction parcels at 03:
- After Cycle 3, every FU branches to 05:, therefore there is once again a single instruction stream. The three threads have now joined.

An address trace from a sample execution can further illuminate the behavior of this program. Figure 10 shows an address trace from the sample data set IZ() = (5,3,4,7). The condition code register contents are shown as they exist at the beginning of each cycle. The rightmost column shows the XIMD partition in each cycle.

Each iteration of this loop contains two critical conditional branches which can be performed in parallel. A VLIW processor can generally only perform one control operation at a time. XIMD can perform both control operations in parallel in a simple and straightforward manner, while allowing the remainder of the loop iteration to proceed.

3.3. Loops with Branches - Barrier Synchronization

Equal length paths are a simple solution to the scheduling of fork-join constructs. Since each thread has the same execution time, no communication is required between the threads in order to join. Unfortunately this method is inefficient if the execution time of the longest thread is not known. In these cases, it is preferable to explicitly synchronize threads when they complete. Barrier synchronizations are an effective technique for these programs on XIMD. In Example 3, the program forks into four threads. Each thread will execute code until it reaches the barrier instruction, then enter a busy-wait loop. When a thread reaches the barrier, it will signal every other thread, that it has done so using the synchronization signal SS_i. When every thread has reached the barrier, then they will all proceed in unison. This procedure can clearly be extended to eight instruction streams, however once again a four stream program running on a four FU processor is shown for clarity.

The example program BITCOUNT1 demonstrates the barrier synchronization concept. The program iterates through an array `D[]` of unsigned integers, counting the number of ones in each element. It then writes the cumulative number of ones into an array `B[]`. The inner loop, which counts the bits can require from 0 to 32 iterations. Since this inner loop is independent for each array element, the programmer has scheduled four copies of this loop in parallel, one on each functional unit. The final operation in each loop, the computation of `B[i]` depends on every previous iteration of the outer loop. Therefore, a barrier has been placed into the code to synchronize the four program threads into a single VLIW program thread. Software pipelining is then used to schedule the final operation of each of the four loop iterations. Additional code is required, but not shown, to handle the final few iterations when less than four elements remain. Note that an additional instruction field SS_i has been added to each instruction parcel in the code listing for this example. This contains the values BUSY or DONE to be used in synchronization.

Cycle	FU0	FU1	FU2	FU3	Condition Codes	Partition	Comment
Cycle 0	00:	00:	00:	00:	XXXX	{0,1,2,3}	Load initial values
Cycle 1	01:	01:	01:	01:	XXFX	{0,1,2,3}	compare to maxint,minint
Cycle 2	02:	02:	02:	02:	TTFX	{0,1,2,3}	Branch - form 3 threads
Cycle 3	03:	03:	04:	04:	TTFX	{0,1}{2}{3}	Update min & max
Cycle 4	05:	05:	05:	05:	TTFX	{0,1,2,3}	compare next element
Cycle 5	02:	02:	02:	02:	TFFX	{0,1,2,3}	Branch - form 3 threads
Cycle 6	03:	03:	04:	03:	TFFX	{0,1}{2}{3}	Update min
Cycle 7	05:	05:	05:	05:	TFFX	{0,1,2,3}	compare next element
Cycle 8	02:	02:	02:	02:	FFFX	{0,1,2,3}	Branch - form 3 threads
Cycle 9	03:	03:	03:	03:	FFTX	{0,1}{2}{3}	No update
Cycle 10	05:	05:	05:	05:	FFTX	{0,1,2,3}	compare last element
Cycle 11	08:	08:	08:	08:	FTTX	{0,1,2,3}	Branch - form 3 threads
Cycle 12	0a:	0a:	0a:	09:	FTTX	{0,1}{2}{3}	Update max
Cycle 13	0a:	0a:	0a:	0a:	FTTX	{0,1,2,3}	Finished

Figure 10: Address Trace for MINMAX

Figure 11 diagrams the state transitions of a portion of this program. The program begins executing startup code as a single SSET at state 00:. It continues to execute as a single SSET for three more cycles, starting four iterations of the outer loop in parallel. States 04: through 08: are the inner loop of the program. The program actually continues as a single SSET until the first time it reaches state 07:, the first branch which depends on different data for each FU. At this point the program forks into four threads as shown. Each SSET continues to execute the loop from 04: to 08: until that loop terminates. During this entire time, each FU has prevented the execution of the barrier by indicating that it has not yet reached the barrier. This is accomplished by holding SS_i at the value BUSY.

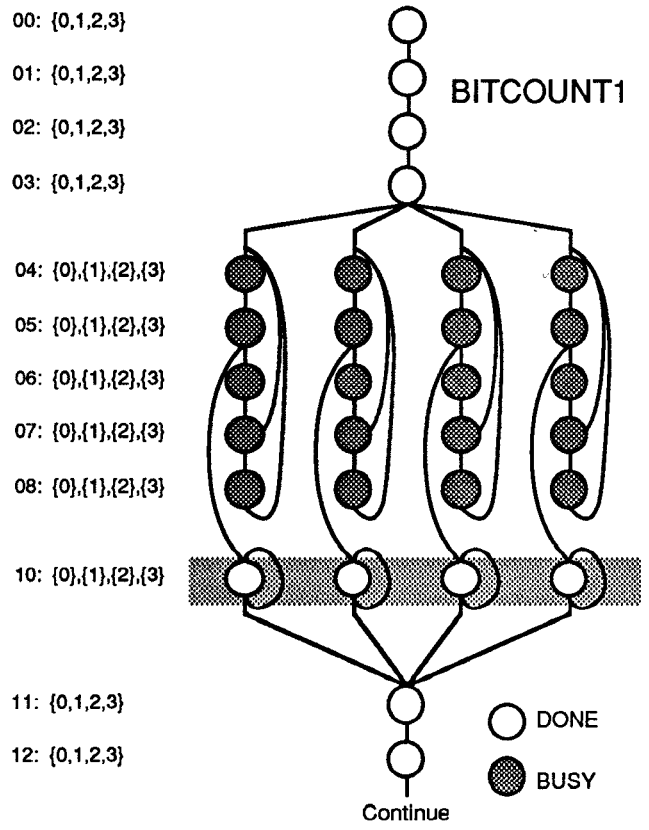

Figure 11: BITCOUNT1 Control Flow.

Once each program thread finishes the inner loop, it branches to state 10:, the 4-way barrier. Each FU which reaches this state will set SS_i to the value DONE. This will signal the other FU's that FU_i has reached the barrier. Each FU will also remain at this state, by conditionally branching, until every FU signals DONE. At that time, the branch condition "ALL-SS" is satisfied and every FU will branch to state 11: This provides a synchronizing barrier for the program threads. Once every thread reaches this barrier, they join into a single SSET again and continue together. States 11: through 15: then complete the four outer loop iterations.

The barrier synchronization mechanism can be generalized to include synchronizations between only some of the program threads, rather than all of them. Also, multiple barrier synchronizations can take place among different program threads.

3.4. Advanced Synchronizations

The XIMD model is capable of a vast range of complex synchronization mechanisms. The options available on a real XIMD architecture are practically limited by the range of values available for SS_i and CC_i, and the complexity of the next address evaluation function δi. Note that when more than one FU is included in an SSET, the range of values available for SS_i and CC_i increases. Figure 12 shows how multiple synchronization signals can be used to implement a complex set of synchronizations.

The flowchart in the figure describes two small concurrent processes. Each process reads some data from an I/O port until the port returns a non-zero, valid value. It also obtains some data from the other process and writes that data to a different I/O port. Each of these operations within a process must occur in order. Note that this creates numerous data dependencies between the processes. We will implement them using the XIMD synchronization bits rather than through register or memory based flags. This will result in increased performance.

It is not known in what order the I/O ports will return data. Ideally, we would like each process to be able to proceed until it is blocked by a data dependency. For example, if Process 1 receives the value for `a`, and Process 2 is still waiting for `x`, we want Process 1 to continue testing for `b`. When Process 2 then needs `a`, it should be immediately available even though Process 1 is busy testing for `b`.

We have implemented a number of non-blocking synchronizations by representing the availability of each variable using one synchronization bit. Process 1 uses the SSET {0,1,2,3}, and Process 2 uses the SSET {4,5,6,7}, executing on an eight functional unit processor. The variables are encoded as follow:

a	SS_0
b	SS_1
c	SS_2
x	SS_4
y	SS_5
z	SS_6

Each signal is set to `DONE` and held at that value whenever the corresponding variable is ready to be used. This allows each process to quickly test for the availability of values from the other process and to busy wait when they are not ready. The producing process can continue unhindered. A standard barrier synchronization is used after both processes are completed to allow later code to redefine the meaning of these signals. The barriers are shown shaded in Figure 12. The dependencies are indicated by arrows.

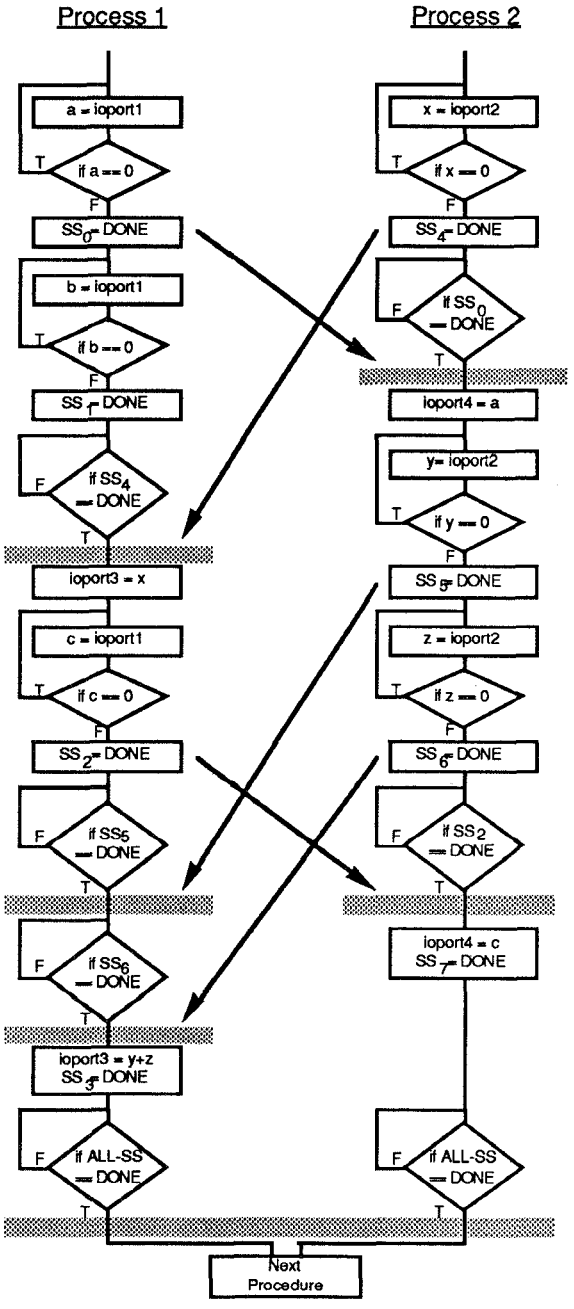

Figure 12: Multiple Non-Blocking Synchronizations.

4. Current Status and Future Work.

Initial analysis shows XIMD to be both an interesting and potentially useful architectural model. Several research efforts are currently in progress with respect to XIMD.

4.1. XIMD Architecture Simulator

A simulator for the XIMD-1 research model has been implemented. This simulator, called *xsim* is used for simulation of interesting programming methods for XIMD architectures and for measuring performance. A companion simulator, *vsim*, simulates a VLIW processor with similar characteristics. Simple hand-coded examples have been tested on this simulator to demonstrate the synchronization mechanisms available in the research model. A number of programs have been gathered to allow more sophisticated performance measurements. These programs will be simulated on both the VLIW and XIMD architectures using both compiled and hand-tuned code. The results will be used to measure the effectiveness of the XIMD architectural model and to assist in the development of compilation techniques. Preliminary results show a significant performance increase on many programs.

4.2. Compiler

Although the XIMD model can substantially leverage many of the compilation techniques developed for VLIW machines, it also demands the development of new techniques which can take advantage of the variety of control flow mechanisms available. An effective XIMD compiler is planned for future research. Currently one interesting compilation technique is under investigation. Figure 13 informally illustrates this method. The original program is separated into individual program threads, either manually, by a high level partitioning tool, or by the compiler. Six such threads are shown in the figure. This apportionment exposes medium and coarse-grain parallelism available in the program.

A retargetable VLIW compiler is used to compile each of these threads. This compiler, based on GNU C [Stallman88], incorporates many traditional and VLIW specific optimizing techniques including an expanded version of Percolation Scheduling, Software Pipelining, and run-time disambiguation. Each thread is compiled several times with varying resource constraints, for example, the compiler allows use of a different number of functional units. This results in a number of choices for the compiled code. Each can be modeled as a rectangle or tile whose width is the required number of functional units and whose length is the static code size. The best set of tiles for each thread is saved as shown in the figure.

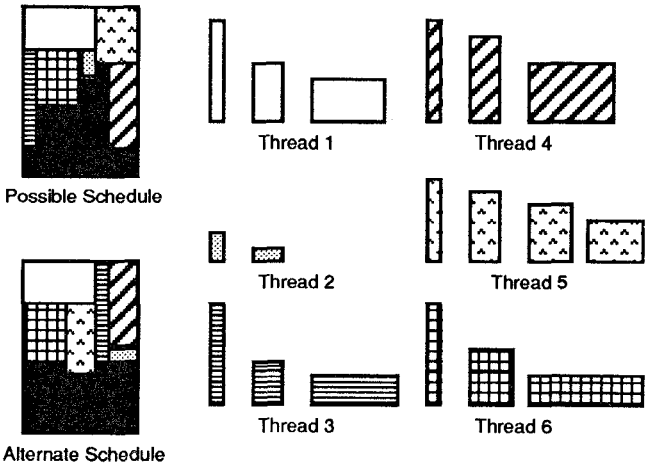

Figure 13: A Proposed Compilation Approach.

Once a set of tiles is produced for each code thread, a packing algorithm is used to schedule one implementation of each thread within a larger space representing the entire instruction memory. Two alternative solutions are shown. This problem is quite similar to the problem of standard cell placement in VLSI CAD, however it is still unknown which placement algorithm will work best given the constraint of data dependencies between tiles of code. This example clearly attempts to optimize for static code density. A similar method might be used to optimize for execution time. There are several other interesting compilation issues with respect to XIMD machines that warrant research.

4.3. Hardware Prototype

A prototype XIMD system has been designed and analyzed. Detailed logic design has been completed. The prototype retains many of the critical features of the research model previously presented.

- Eight Universal Functional Units
- 24--ported Global Register File (256 registers)
- Non-pipelined Control Path
- Support for all Modeled Synchronization Mechanisms.

A few differences arise when compared to the research model. These have been incorporated into the prototype design in order to decrease cycle time or reduce cost.

- 3-stage Data Path Pipeline
  (Operand Fetch - Execute - Write Back)
- Distributed Memory (1MB per FU)
- Traditional Sequencer
  (Incrementer + 1 explicit branch target)
- Limited Number of Constants per Operation

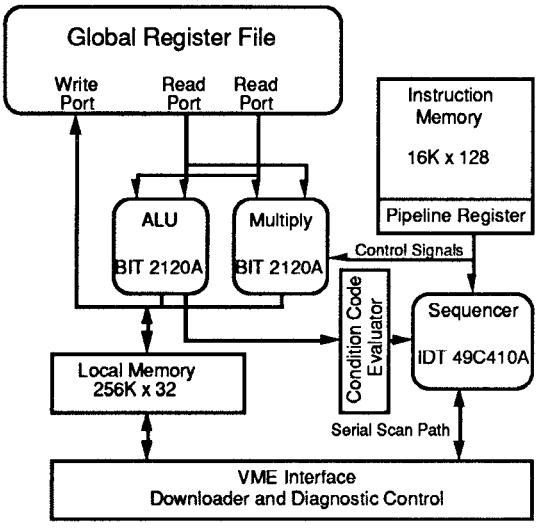

Figure 14: XIMD Prototype.

A simplified diagram of the register file, one functional unit and some support hardware is shown in Figure 14 The hardware prototype will be constructed as a modular system comprised of a central motherboard and eight attached functional units. Both custom and off the shelf VLSI components will be used in addition to standard MSI CMOS logic. The prototype will interface to a VME based host such as a SUN 3.

An initial performance analysis predicts a cycle time of 85ns. This will result in peak performance in excess of 90 MIPS/90 MFLOPS.

4.4. Custom Register File Chip

A high-performance, completely orthogonal, global register file is critical to the XIMD research model. It must also be large enough to support eight or more functional units. Such a device is not commercially available, therefore a custom register file chip has been designed and fabricated to support this architecture [Maly90]. Pinout limitations in the available process combined with architectural and performance factors strongly influenced the partitioning of the register file into multiple chips.

Each chip supports 8 simultaneous reads and 8 simultaneous writes. Two chips can be wired in parallel as shown to provide 16 reads and 8 writes. Each chip is two bits wide and contains 256 global registers. This results in a minimum requirement of 32 register file chips for the proposed prototype architecture. Addresses are multiplexed to reduce pin count and internal data and address latches have been included to simplify timing.

The register file chip is designed using the MOSIS 2 micron scalable CMOS process. The complete design contains approximately 70,000 transistors on a 7.9 x 9.2 mm die. The parts are packaged in a 132-pin pin grid array. Fabricated parts were recently delivered by MOSIS. The first parts are basically functional, however yield is very low. A second generation design is under consideration.

4.5. Research Schedule

The initial phase of research into XIMD architecture has produced a useful research model, a prototype design, and a critical component, the register file chip. Research is expected to continue on several fronts. The goals for 1991 are:

- A complete physical design for the hardware prototype.
- A fully working version of the VLIW compiler.
- Simulation results from several program examples.
- Fully functional, tested register file chips.

Acknowledgement

Others at Carnegie Mellon have contributed to this work. Mauricio Breternitz and Prof. Alex Nicolau (U.C. Irvine) developed the VLIW compiler used in this work. Prof. Wojtek Maly and his students designed the register file chip. Gregory Palmer worked on the XIMD prototype design. This work was supported by the Semiconductor Research Corporation and by the National Science Foundation.

References:

[Aho86] A. Aho, et al., *Compilers: Principles, Techniques and Tools.*,Addison-Wesley, Reading, Mass., 1986.

[Bakoglu89] H.B. Bakoglu, et al., "IBM Second-Generation RISC Machine Organization", *ICCD '89* ,138-142, IEEE, 1989.

[Breternitz90] M. Breternitz Jr., *VLIW Compilation and Architecture Synthesis*, Ph.D. Thesis in preparation, Carnegie Mellon, 1991.

[Colwell87] R. P. Colwell, et al., "A VLIW Architecture for a Trace Scheduling Compiler", ASPLOS-II, ACM., 1987.

[Colwell90] R. P. Colwell, et al., "Architecture and Implementation of a VLIW Supercomputer", *Supercomputing '90*, 1990.

[Ebcioglu87] K. Ebcioglu, "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps", *IEEE Micro-20* , Dec, 1987.

[Ebcioglu88] K. Ebcioglu, *Some Design Ideas for a VLIW Architecture for Sequential-Natured Software*, IBM Research Report, April 1988.

[Fisher81] J. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Transactions on Computers* , July, 1981, C-30:pp. 478-490.

[Fisher83] J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512", *Proc. of the 10th Symp.. Comp. Arch.*, IEEE Computer Society, 1983.

[Flynn66] M. J. Flynn, "Very High-Speed Computers", *Proceedings of the IEEE*, V 54, 1900-1909, Dec. 1966.

[Intel89] Intel, "Intel 80960CA User's Manual",Intel, 1989.

[Intel89a] Intel, "Intel i860 64-bit Microprocessor Programmer's Reference Manual",Intel, 1989.

[Labrousse90] J. Labrousse and G. Slavenburg, "A 50 MHz microprocessor with a VLIW architecture", ISSCC '90, IEEE, 1990.

[Maly90] W. Maly, et al., *Memory chip for 24-port global register file*, Technical Report, 1990.

[Nicolau85] A. Nicolau, "Percolation Scheduling: A Parallel Compilation Technique", Cornell CS. Dept Technical Report TR 85-678 , May, 1985.

[Palmer90] G. Palmer, *Design of an XIMD Computer*, Masters Thesis in preparation, Carnegie Mellon, 1991.

[Rau89] B.R.Rau, et al., "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs", *Computer*, Vol 22, No. 1 , January, 1989.

[Siegel81] H.J. Siegel, et al., "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition", *IEEE Transactions on Computers* , December, 1981, C-30:pp. 934-946.

[Stallman88] R. Stallman, *Internals of GNU CC*, Free Software Foundation, Cambridge, MA, 1988..

[Tomasulo67] R.M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal*, Vol 11, 25-33, January 1967.

[Wolfe89] A. Wolfe, *xsim Reference Manual*, Internal Report, Carnegie Mellon, 1989.

[Zaafrani90] A. Zaafrani, H. G. Dietz, and M. T. O'Keefe, "Static Scheduling for Barrier MIMD Architectures", TR-EE 90-10, Purdue University, January 1990.