# $SIM_dD$: A New Taxonomy for Data-Parallel Architectures


## Abstract

SIMD architectures improve execution throughput by exploiting data-level parallelism inherent in programs. Conventional SIMD implementations (referred to as packed-SIMD hereafter) require the data elements to be properly arranged in vector registers before SIMD operations can be performed on the vector elements. Packed-SIMD ($SIM_pD$) architectures are efficient for sequential data accesses but incur overhead in performing data rearrangement for more flexible data accesses where the vector elements could potentially come from arbitrary, disjoint sources. This report discusses disjoint-SIMD ($SIM_dD$) architectures, introduced by the IBM eLite DSP team[1], which address the limitation of $SIM_pD$ features in a vector-DSP. It is shown that there exists a group of telecommunication and multimedia benchmarks which exhibits data-level parallelism well mapped to $SIM_dD$ patterns. In this project, the $SIM_dD$ idea is adapted for a variant of general-purpose processors. Both $SIM_pD$ and $SIM_dD$ extensions are added to a basic 5-stage pipelined, MIPS-like architecture to demonstrate the concept of data-level parallelism. VHDL models are simulated to verify the designs and to assist performance evaluation of the designs in terms of clock cycles. Simulation results show that $SIM_dD$ has a performance advantage over $SIM_pD$ extensions for programs with flexible data access patterns.


## Organization of Report

## 1. Overview of Data Parallel Architectures

### 1.1 Overview of SISD, VLIW, $SIM_pD$ and $SIM_dD$ Architectures

Single-Instruction-Single-Data (SISD) is the basic processor architecture where a single instruction stream operates on a single data stream, as shown in *Figure 1(a)*. Assuming three-address instructions, register-to-register architectures, the instructions must specify an opcode identifying the operation to be performed; a destination register address for result storage; and two source register addresses providing the source operands.

The Very-Long-Instruction-Word (VLIW) architecture in *Figure 1(b)* is an example of Multiple-Instructions-Multiple-Data (MIMD) architectures where multiple instructions are executed in parallel on multiple data streams. Parallel execution is possible for a sequence of instructions with no data and control dependencies. VLIW architectures exploit both instruction-level and data-level parallelism in programs to improve execution throughput. The instruction format for VLIW architectures is necessarily a very long word, composed of multiple SISD instructions.

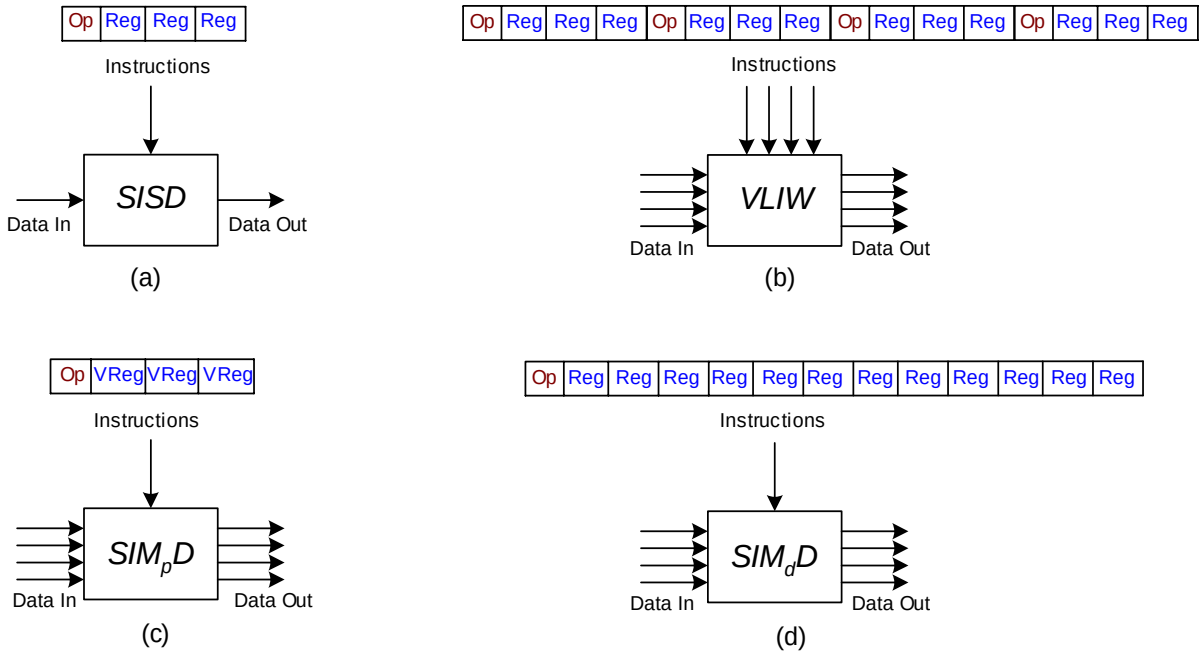


*Figure 1 (adapted from [2]): (a) Single-Instruction-Single-Data architecture. (b) Very-Long-Instruction-Word architecture. (c) Packed Single-Instruction-Multiple-Data architecture. (d) Disjoint Single-Instruction-Multiple-Data architecture.*

$SIM_pD$ extensions are normally seen in vector processors and general-purpose processors. In vector processing, a single instruction is performed on data elements pre-arranged in vector registers. $SIM_pD$ features are also typically implemented as multimedia extensions to general-purpose processors. Multimedia data types are typically 8-bit or 16-bit. For 32-bit general-purpose processors, four 8-bit (or two 16-bit) operations could be performed in parallel using 32-bit execution units. This is known as *sub-word parallelism*. Thus the 32-bit scalar registers in general-purpose processors may be seen as vector registers with four 8-bit elements (or two 16-bit elements). Thus the term "vector registers" may refer to the true vector registers in vector processors; as well as the scalar registers in general-purpose processors with SIMD extensions. The instructions for $SIM_pD$ architectures must specify an opcode identifying the single operation

to be performed; a destination vector register address and two source vector register addresses. Note that the multiple data elements must be "packed" into vector registers before SIMD operations can be performed, hence the name packed-SIMD ($SIM_pD$).

For $SIM_dD$ architectures, as shown in *Figure 1(d)*, the data need not be packed into vector registers prior to execution. The vector elements may be come from *disjoint* locations as long as they are available in one of the programmer-visible registers, and are composed dynamically during instruction execution. This implies that the instructions for $SIM_dD$ architectures must include an opcode; a destination register address and two source register addresses for *each* data stream. This result in long instruction words much like that of VLIW architectures. We will see later in the report that $SIM_dD$ architectures prove to be more efficient for programs with flexible data access patterns.

## 1.2 General vs. Indirect $SIM_dD$ Implementations

As seen in *Section 1.1*, general $SIM_dD$ architectures require a long instruction word to specify the single operation; source and destination register addresses for *each* data stream. This poses difficulty in extending the ISA of general-purpose processors that typically have fixed instruction formats. In this report, an alternative $SIM_dD$ implementation, named indirect-$SIM_dD$, is considered. The indirect-$SIM_dD$ implementation uses a level of indirection, i.e. pointers, to specify the locations of multiple data elements. Each pointer may contain multiple indices to indicate disjoint data sources, so fewer bits are required for the instructions. *Figure 2(b)* shows that the instructions for indirect-$SIM_dD$ architectures need only specify an opcode, a destination pointer address and two source pointer addresses. This instruction format closely resembles that of existing general-purpose processors and so indirect-$SIM_dD$ features can be easily added as multimedia extensions to general-purpose processors.

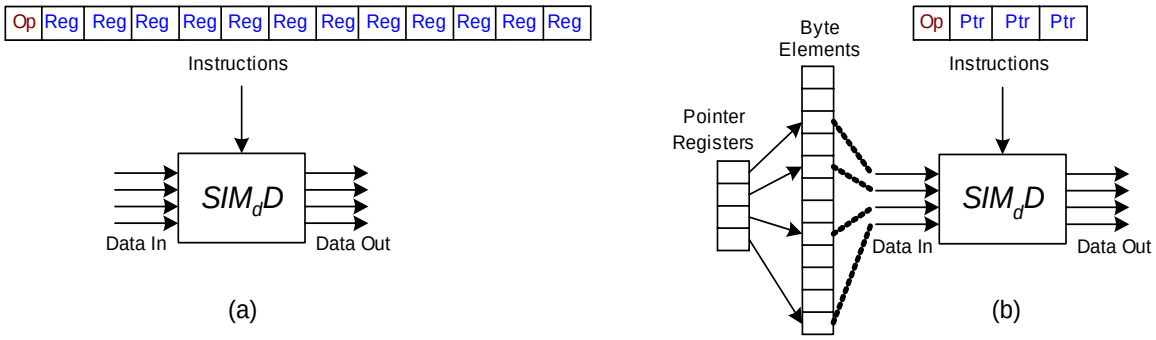


*Figure 2 (adapted from [2]): (a) General $SIM_dD$ architecture. (b) Indirect-$SIM_dD$ implementation.*

## 2. Motivation for $SIM_dD$ Architectures

The IBM eLite DSP team showed that while some telecommunication and multimedia algorithms are well mapped to $SIM_pD$ patterns, others are more efficiently scheduled and executed by $SIM_dD$ architectures. Data access patterns may be characterized by two metrics: *vector element distance* and *vector stride*. Recall that the term "vector elements" may refer to elements in vector registers for vector processors; as well as the sub-word elements in scalar registers for general-purpose processors with SIMD extensions. *Vector element distance*, $\Delta$, may be defined as the distance between the multiple elements pointed to by a single pointer register. This pointer register may then be updated such that a constant *vector stride*, $\Psi$, is added to the current position of each element. *Figure 3* and *Figure 4* illustrate the definition of these metrics.

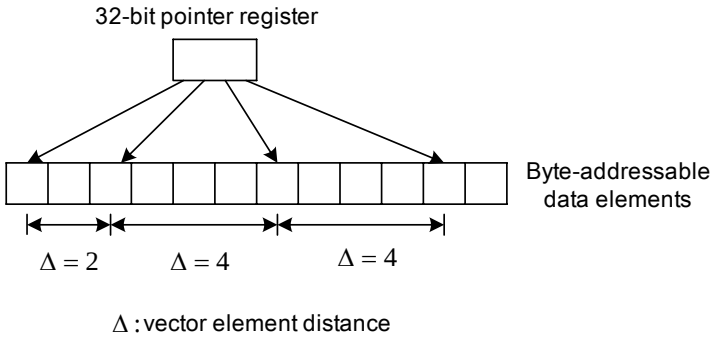


*Figure 3 (adapted from [2]): Illustration of vector element distance.*

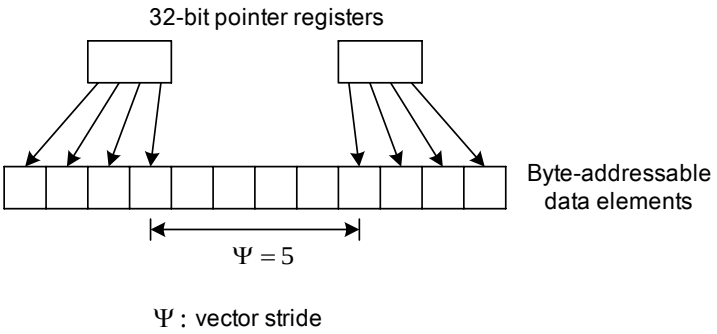


*Figure 4 (adapted from [2]): Illustration of vector stride.*

*Figure 5* shows the data access pattern characterization for a set of commonly used multimedia and telecommunication benchmarks based on the metrics $\Delta$ and $\Psi$. Benchmark kernels used include block FIR filter, autocorrelation matrix, decimation and interpolation, least-mean-squares filter, H.263 quantization, Viterbi decoding algorithm etc.

Data accesses with $\Delta$=0,1 and $\Psi$=0,1,4,8,12… are categorized as $SIM_pD$ patterns. ROT patterns ($\Delta$=1 and $\Psi$=1,2,3) are easily handled by *rotating registers* typically found in digital signal processors and vector processors. Rotating registers are used in hardware renaming mechanism to eliminate the need for software loop unrolling, and thus avoid unnecessary expansion of program code. All other (flexible) data access patterns are categorized as FLEX patterns.

From *Figure 5*, we may conclude that there exists a group of telecommunication and multimedia benchmarks containing significant amount of data access patterns more flexible than the conventional $SIM_pD$ and ROT patterns. Thus, there is a need to extend $SIM_pD$ architectures to allow for SIMD operations on disjoint vector elements. The extended architecture, $SIM_dD$, will be able to handle FLEX data accesses with minimal overhead while leaving unchanged the performance of $SIM_pD$ data accesses.

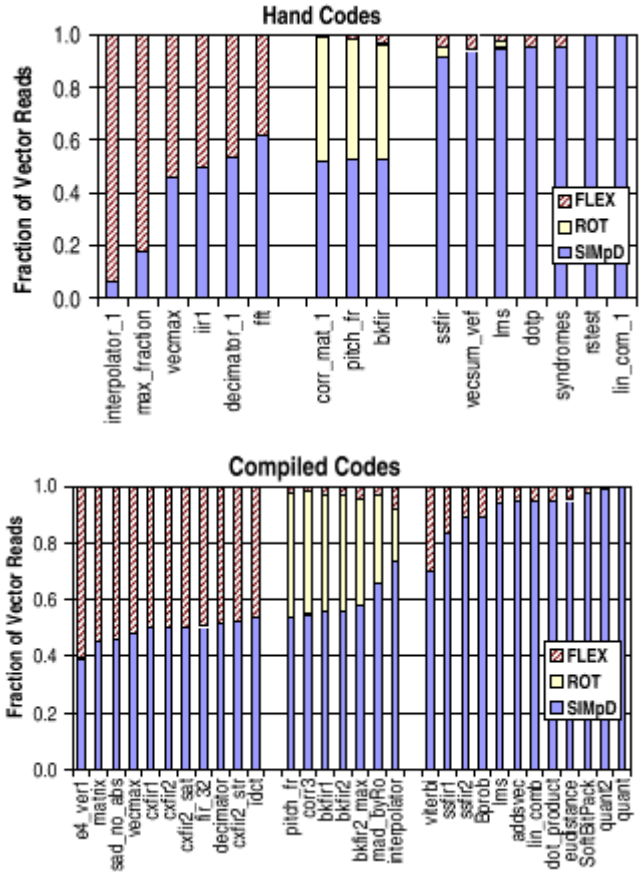


*Figure 5 (extracted from [2]): Data access pattern characterization of telecommunication and multimedia benchmarks.*

## 3. Architectural Impact of $SIM_pD$ and $SIM_dD$ Extensions On General-Purpose Processors

In this project, we will use a basic 5-stage pipelined, MIPS-like general-purpose processor as the reference architecture. This pipelined processor will be extended to incorporate conventional $SIM_pD$ features as well as the proposed indirect-$SIM_dD$ features. In this section, the architectural impact of $SIM_pD$ and indirect-$SIM_dD$ extensions will be discussed. In particular, additional hardware and changes to assembly programming requirements are identified for each type of extension.

### 3.1 Experimental Setup

#### 3.1.1 Basic 5-stage Pipelined Processor

*Figure 6* shows the block diagram of the basic pipelined, MIPS-like general-purpose processor with five pipeline stages as described in *Table 1*.

Necessary information (for example, control signals for later pipeline stages) is passed down the pipeline by the pipeline registers located between stages. Note that branch address calculations and the zero-condition are evaluated in the ID stage, and so branch decisions are available as early as the ID stage. In addition, a forwarding unit is added to the processor to feed the proper data among internal resources. Forwarding techniques are commonly used to eliminate data hazards in the pipeline, and are only possible when the destination stage is later in time than the source stage.

| ***Pipeline Stage*** | ***Pipeline Stage Name*** | ***Description of Pipeline Stage*** |
|---|---|---|
| 1 | Instruction Fetch (IF) | Instruction is read from the instruction memory using the current PC value. The PC address is incremented by 4 and the instruction is placed in the IF/ID pipeline register. |
| 2 | Instruction Decode & Register File Read (ID) | The instruction from the IF/ID pipeline register is decoded into an opcode, a destination register address, two source register addresses and a 16-bit immediate value. The opcode field of the instruction is fed into the control logic to produce the necessary control signals such as *RegWrite*, *ALUOp* etc. The register file is read using the decoded source register addresses. The immediate value is sign-extended to 32 bits and may be used for branch address calculation in this stage. The sign-extended value is also stored in the ID/EX pipeline register for use in later pipeline stages. |
| 3 | Execution or Address Calculation (EX) | The ALU performs arithmetic and logical operations (only additions in this version) on the source register contents passed from the ID/EX pipeline register. |
| 4 | Data Memory Access (MEM) | Data is either read from or written to the data memory using addresses passed from the EX/MEM pipeline register. |
| 5 | Write Back (WB) | Execution result or data from memory is chosen for writeback to the register file, depending on the control signal *MemToReg*. |

*Table 1: Brief description of the five pipeline stages in the basic pipelined, MIPS-like processor architecture (refer to [4] for more details).*

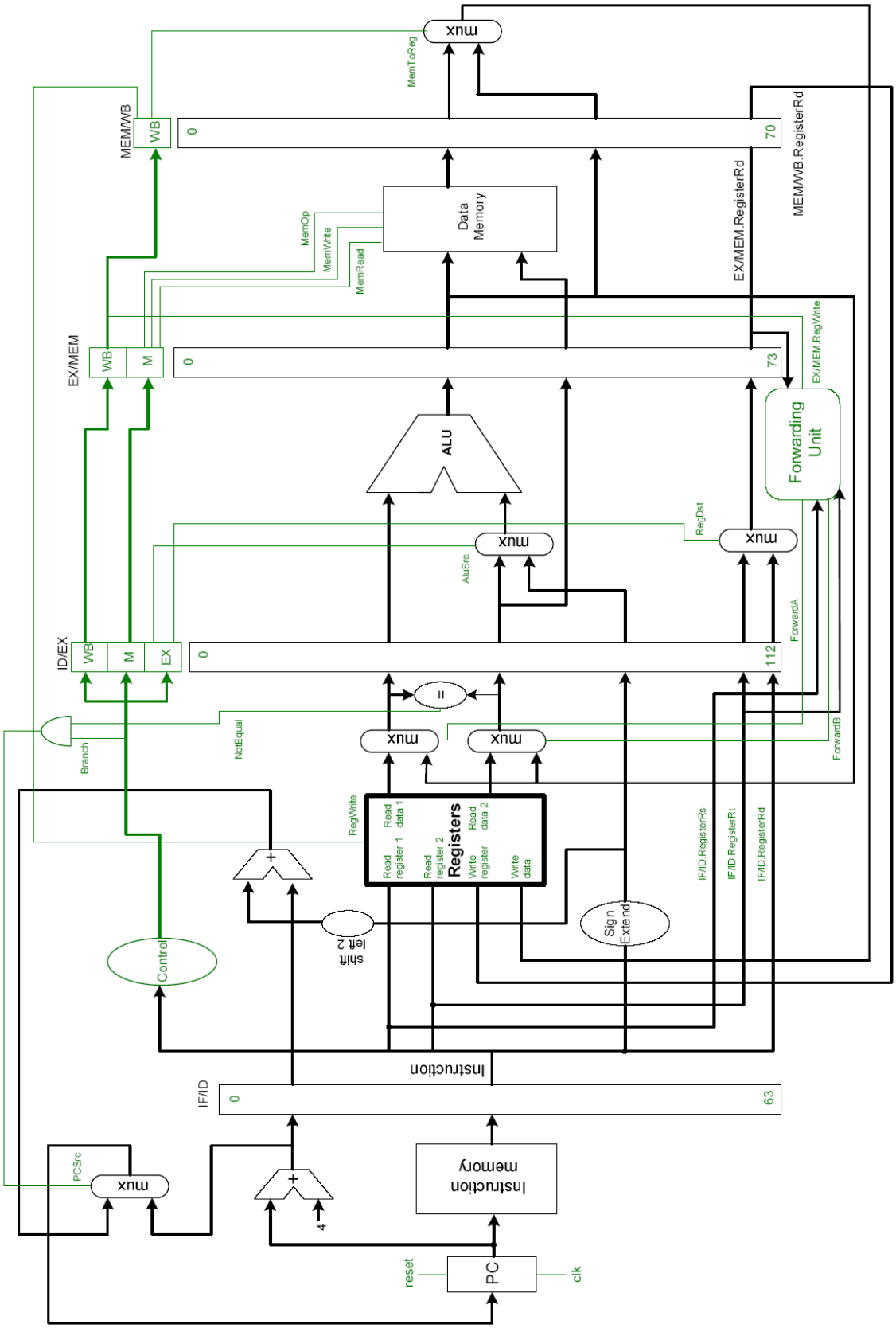


*Figure 6 (extracted from [4]): Basic five-stage pipelined, MIPS-like processor architecture with forwarding unit.*

### 3.1.2 Data Memory Access Patterns

Two simple loop programs with different data access patterns are hand-coded and manually scheduled for the following architectures:

(a) the basic five-stage pipelined processor (as described in *Section 3.1.1*)
(b) the pipelined processor from (a), extended with $SIM_pD$ features
(c) the pipelined processor from (a), extended with indirect-$SIM_dD$ features.

These two simple programs, as shown in pseudo-code in *Table 2*, are sufficient to demonstrate the improved efficiency of indirect-$SIM_dD$ extensions with respect to the processor architectures (a) and (b).

***Program 1***

```
# define N=8
char x[N], y[N], z[N];

for (int i = 0; i < N; i++)
    z[n] = x[n] + y[n];
end for;
```

***Program 2***

```
# define N=8
char x[N], y[N], z[N];

for (int i = 0; i < N/2; i++)
    z[n] = x[2n] + y[2n];
end for;
```

*Table 2: Two simple loop programs with different data access patterns are used for processor simulations in this project.*

The first program is a loop performing addition on sequential elements ($SIM_pD$ pattern with $\Delta=1$, $\Psi=4$) of the byte arrays *x* and *y*. On the other hand, the second program is a loop performing addition on every other elements (FLEX pattern with $\Delta=2$, $\Psi=8$) of the byte arrays *x* and *y*. We will see in *Section 4* that the VHDL simulations of the processor architectures (a), (b) and (c) using these two simple programs clearly show the advantage of indirect-$SIM_dD$ extensions for handling flexible data accesses.

We also assume that the data memory unit in the MEM pipeline stage contains the elements of the source arrays *x* and *y* in the manner shown in *Figure 7*. Memory locations 0x10 to 0x17 are reserved for the result array *z*. Note that the elements of the arrays *x*, *y* and *z* are 8-bit values ("sub-words"), much like the data types normally encountered in telecommunication and multimedia applications.

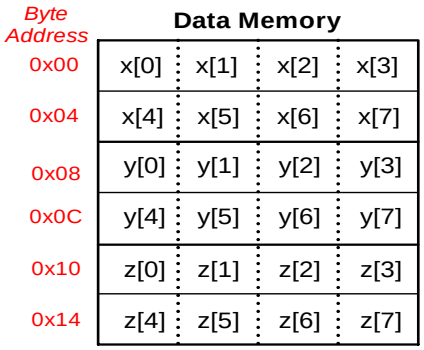


*Figure 7: Data arrangement in the data memory unit.*

## 3.2 Hardware Requirements for SIM$_p$D Extensions

### 3.2.1 SIMD Execution Unit

To extend the basic pipelined processor with SIM$_p$D features, all the execution units must be extended to handle SIMD operations. For our pipelined processor, there is only an ALU that performs additions. In typical ripple-carry adder implementation, the carry-out bit of each byte addition is channeled into the next more significant byte as the carry-in bit. Thus we must ensure that carry-out bit of each byte into the next more significant byte is disabled for SIMD operations. A control signal *ALUOp* is added as an input to the ALU to decide whether the currently executed instruction is a SIMD operation, and thus appropriately disable the carry-out bits of each byte addition.

More generally, general-purpose processors have ALUs that perform arithmetic (add, subtract) and logical (AND, NOT, XOR etc.) operations; as well as shifters to perform bit-shifting. The task of extending these functional units to handle SIMD operations may seem more complicated. For combined adder/subtractor implementation, the carry-out bit of each byte into the next more significant byte is simply disabled for SIMD instructions, just like for the ripple-carry adder discussed above. The same technique is applied to the shifters to disable the left-most or right-most shifted bits for each byte. As for the logical operations, nothing needs to be modified since the logical operations are bitwise.

### 3.2.2 Data Re-Arrangement of Vector Elements

SIM$_p$D paradigm requires the data elements to be properly arranged in vector registers. This does not pose any problem to sequential data accesses since the data elements are pre-arranged in sequential fashion. However, for non-sequential data accesses such as accesses with a constant vector element distance or vector stride, data re-arrangement is required. Typically, a combination of *mix* and *permute* operations will produce the required arrangement.

As an illustration, *program 2* performs addition on the pair of every other elements of arrays *x* and *y*. *Figure 8(b)* shows the elements of *x* that must reside in one of the source registers (say, register *R3*). Note that the elements of *y* must also be arranged in a similar fashion. Two memory loads are necessary to temporarily store the four elements of *x* required in a single loop iteration. Mix and permute operations are then performed on the elements in *R1* and *R2* to pack *x[0], x[2], x[4] and x[6]*, in that order, into register *R3*.

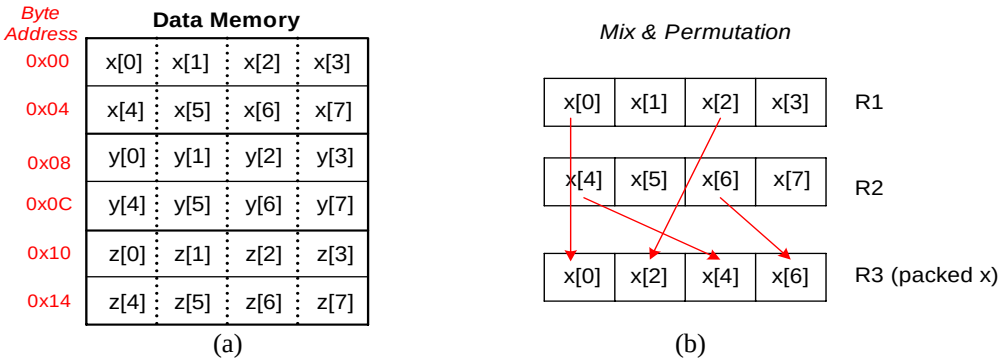


*Figure 8: (a) Locations of array elements in the data memory. (b) Data re-arrangement (mix and permute) is required to properly pack data for SIMD operations.*

We identify that an additional execution unit is needed in the EX pipeline stage to support the aforementioned data rearrangement. This "mix & permute unit" (MPU), as shown in red in *Figure 10*, operates in parallel with the ALU and supports two new instructions *mixr* and *perm*. The operations and instruction formats for *mixr* and *perm* are shown in *Figure 9*. The instruction formats of these two instructions are selected to conform to the existing ISA formats of the basic pipelined processor such that the hardware and software impact of these additions is minimal. However, the burden of mix and permute operations is placed on the programmer. These data re-arrangement instructions will contribute to overhead especially when the accessed data elements reside in random locations.

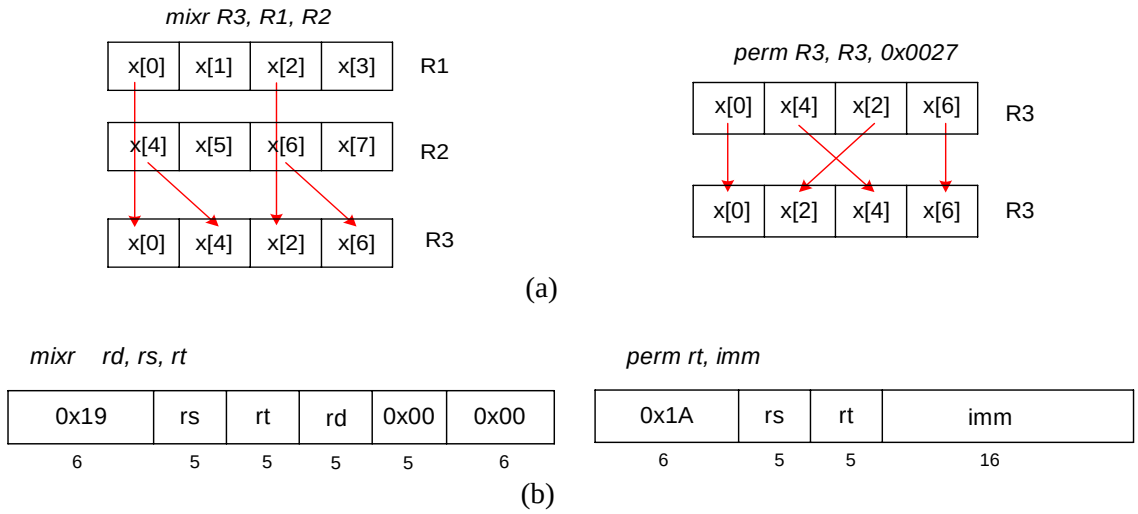


*Figure 9: (a) Operations of the new instructions* mixr *and* perm*. (b) Instruction formats of the new instructions* mixr *and* perm*.*

Putting it all together, the additional hardware required to extend the basic pipelined processor with $SIM_pD$ features is highlighted in red in *Figure 10*. Firstly, the ALU must be able to handle SIMD operations as decided by the control signal *ALUOp*. Secondly, a "mix & permute unit" (MPU) that handles "mix right" and "permute" operations is added to the EX pipeline stage. The output of the MPU must be passed down to the WB stage so that it may be selected as the data to be written back into the destination register. *Table 3* summarizes the additional control signals required for the $SIM_pD$ pipelined processor.

Note that the MPU is basically a functional unit rewiring the input signals to give the appropriate output arrangement. Thus the MPU is likely to have less computation delay than the ALU and so will not affect the pipeline clock period.

| *Control Signal* | *Stage* | *Values* | *Description* |
|---|---|---|---|
| *ALUOp* | EX | 0 | ALU performs normal addition operation. |
| | | 1 | SIMD addition is required. Carry out bits of each byte addition into the next more significant byte is disabled. |
| *MPOp* | EX | 0 | MPU is performing *mixr* on two source registers. |
| | | 1 | MPU is performing *perm* on a single source register. |
| *MemToReg* | WB | 00 | Writeback data comes from the ALU result. |
| | | 01 | Writeback data comes from the data memory. |
| | | 10 | Writeback data comes from the MPU result. |

*Table 3: Additional control signals required for $SIM_pD$ extensions.*

There are minor details that are not implemented in our version of $SIM_pD$ pipelined processor. For example, functional units in general-purpose processors normally output condition flags such as N (sign bit), Z (zero), V (overflow), C (carryout), for each arithmetic or logical instruction. The handling of SIMD operations depends on the implementation decision whether to output a set of condition flags for *each* SIMD element; or only output the condition flags based on the operation on the first (or last) SIMD element. The former approach is typically seen in vector processors such as UC Berkeley's VIRAM architecture[5]. On the other hand, the Intel processors only have a set of flag registers. Thus the Intel-MMX[6] instructions are implemented such that the condition codes for each byte operation are stored as the destination operand, rather than in the flag registers.

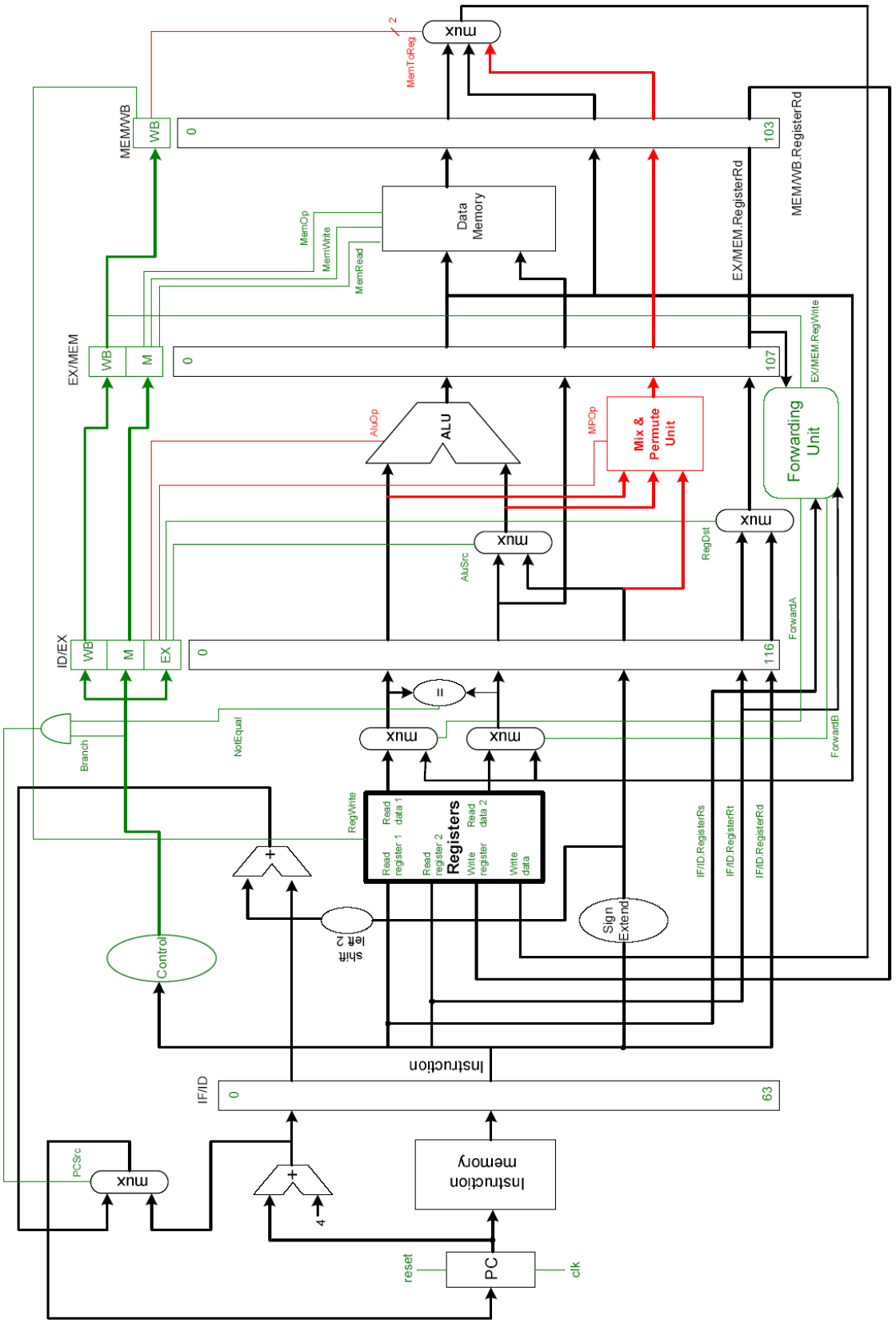


*Figure 10: Basic five-stage pipelined processor extended with SIM$_p$D features.*

## 3.3 Hardware Requirements for Indirect-SIM$_d$D Extensions

### 3.3.1 SIMD Execution Unit

Just like the hardware requirements for SIM$_p$D extension, the execution units for SIM$_d$D extension must be able to handle SIMD instructions. Once again, a control signal *ALUOp* is added as an input to the ALU to decide whether the currently executed instruction is a SIMD operation, and thus appropriately disable the carry out bits of each byte addition.

### 3.3.2 Indirect-SIM$_d$D Addressing Using Pointers

In *Section 1.2*, we introduced indirect-SIM$_d$D implementation to achieve simplicity in extending the ISA of general-purpose processors. Pointers are used to indirectly specify the possibly disjoint vector elements. The packing of vector elements is performed dynamically depending on the current pointer values. Thus, SIM$_d$D extensions free programmers from the burden of re-arranging the vector elements, and yet achieve flexibility in data accesses in comparison to SIM$_p$D extensions.

As one of the requirements of SIMD extensions, the vector elements must reside in the register file. Since indirect-SIM$_d$D extensions are able to specify multiple disjoint sub-words as the vector elements, the register file in the basic pipelined processor must be modified to contain byte-addressable registers. The block diagram of the modified register file is shown in *Figure 11*.

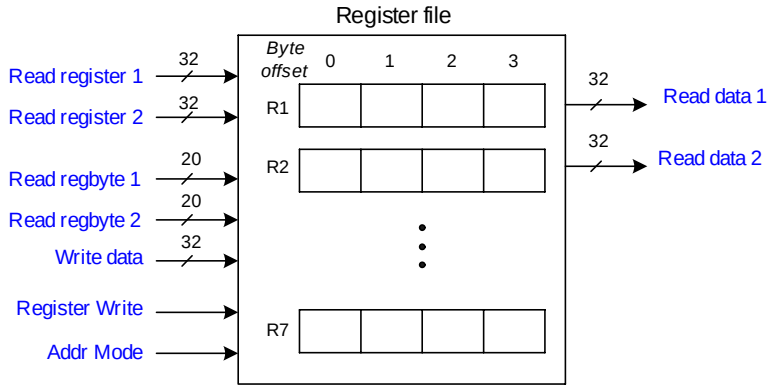


*Figure 11: Byte-addressable 32-bit register file*

The register file handles two addressing modes: normal register read, and register read using pointers. To support register read using pointers, a pointer unit is added to the ID pipeline stage, just before the byte-addressable register file. The pointer unit is essentially a small register file that performs pointer setup and manipulation. Note that the pointer values may be loaded from memory or explicitly set. For our implementation, a new instruction *lptr* ("load pointer") is added to explicitly set the value of a pointer. The instruction format of *lptr* is shown in *Figure 12*. The source addresses of the vector elements are specified by 5-bit immediate values: 3 bits for register address and 2 bits for the byte offset of the specified register. Note that the execution of the instruction *lptr* will complete by the second pipeline stage, the ID stage. We will see later in *Section 3.4* that this will further simplify the scheduling of assembly instructions and so contribute to reduced overhead for indirect-SIM$_d$D extensions.

*lptr pt, imm1, imm2, imm3, imm4*

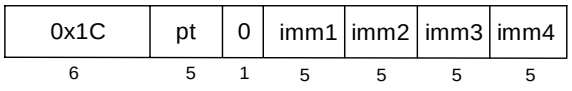


*Figure 12: The instruction format for* lptr.

*Figure 13* shows the pipelined processor extended with indirect-SIM$_d$D features. The additional hardware is highlighted in red. A pointer unit is added to the ID pipeline stage and the register file is byte-addressable to support indirect addressing with pointers. *Table 4* summarizes the additional control signal required to extend the basic pipelined processor with indirect-SIM$_d$D features.

| ***Control Signal*** | ***Stage*** | ***Values*** | ***Description*** |
|---|---|---|---|
| *PtrWrite* | ID | 0 | Pointer register read. |
| | | 1 | Pointer register write. |
| *AddrMode* | ID | 0 | Normal register file read (source register addresses provided in the instruction). |
| | | 1 | Register file read using pointers. |
| *ALUOp* | EX | 0 | ALU performs normal addition operation. |
| | | 1 | SIMD addition is required. Carry out bits of each byte addition into the next more significant byte is disabled. |

*Table 4: Additional control signals required for indirect-SIM$_d$D extensions.*

Adding a pointer unit just before the register file in the ID pipeline stage may increase the processor clock cycle. No quantitative measure is done on the impact of the pointer unit on the processor clock cycle. However, the pointer unit is typically a small register file and so may not incur too much propagation delay.

Furthermore, a more complex data forwarding unit is needed for the indirect-SIM$_d$D pipelined processor. In SIM$_p$D architectures, reuse of data in the pipeline is inconvenient because the forwarded vector elements may have to be re-arranged before data can be used by the subsequent instructions in the pipelined. This problem remains in the indirect-SIM$_d$D architectures and so the forwarding unit must be extended to forward individual bytes among the pipeline stages. The forwarding unit decides the appropriate feeding of data by performing comparison on the register source and destination addresses, where the SIMD elements are addressed using a 3-bit register address and a 2-bit byte offset.

We saw in *Section 3.2* and *3.3* that the additional hardware required for SIM$_p$D and indirect-SIM$_d$D extensions is minimal. However, the area occupied by the architecture implementations depends heavily on the placing and routing of hardware resources on the physical chips. While automated place-and-route tools for ASICs or FPGAs may not produce the optimal arrangement, manual placing and routing of hardware resources are extremely tedious and will not be attempted for the scope of this project.

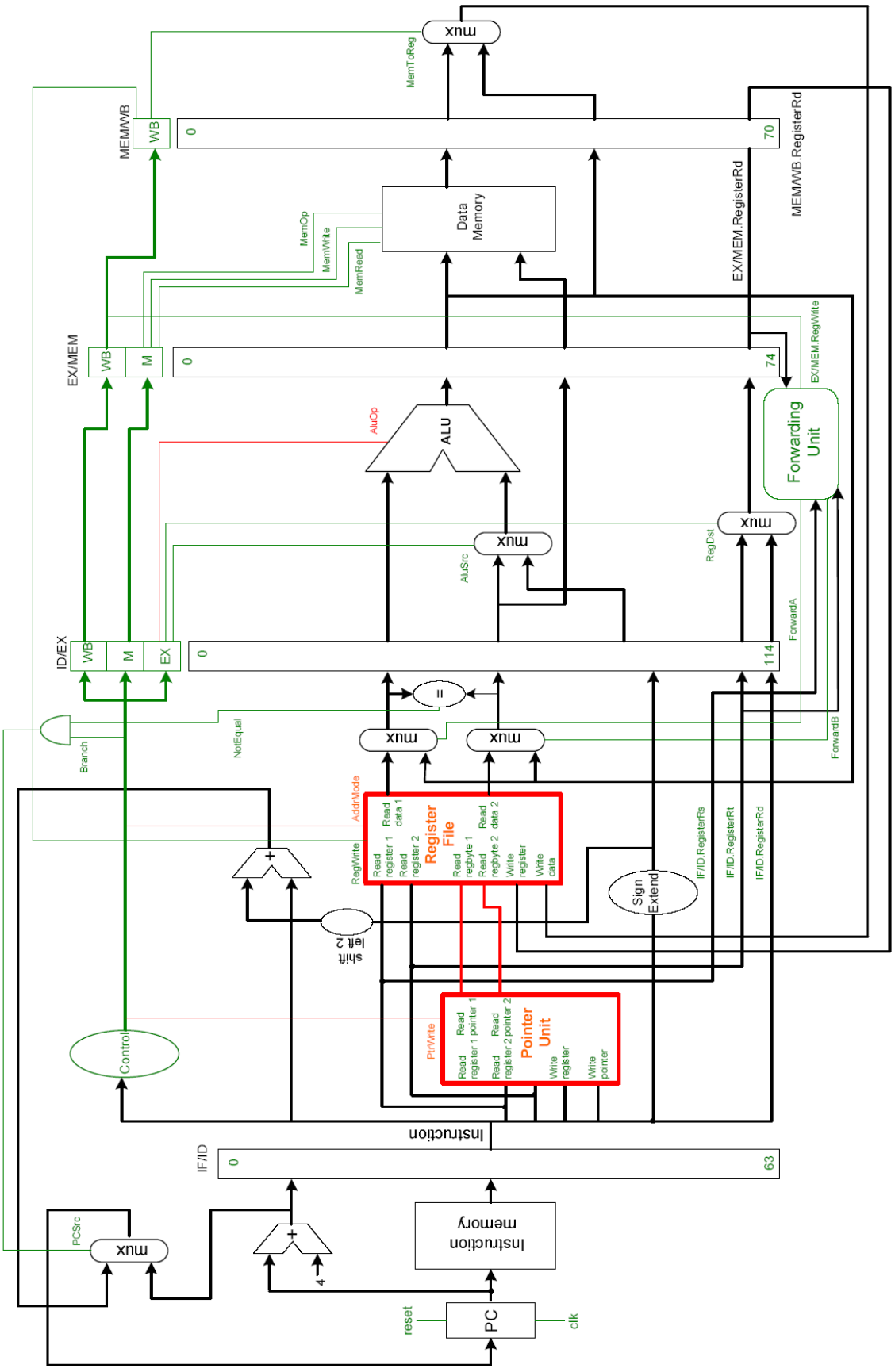


*Figure 13: Basic five-stage pipelined processor extended with indirect-SIM$_d$D features.*

### 3.4 Assembly Programming Requirements

In previous sections, we identified the additional hardware required to extend a pipelined, general-purpose processor with SIMD features. We now discuss the changes to assembly programming requirements for the extended pipelined processors, implemented as described in *Section 3.2* and *3.3*.

*Figure 15* shows the scheduled assembly instructions for *program 1* and *program 2* (refer *Section 3.1.2*) for the three architectures considered:

(a) the basic five-stage pipelined processor
(b) the pipelined processor from (a), extended with $SIM_pD$ features
(c) the pipelined processor from (b), extended with $SIM_dD$ features

Recall that *program 1* is a simple loop performing addition on sequential elements of the byte arrays *x* and *y*. *Figure 15* shows that the assembly instructions for *program 1* do not differ much for all three architectures. However, we expect the pipelined processor with $SIM_pD$ and $SIM_dD$ extensions to occupy less execution time for *program 1* since the number of loop iterations is reduced by a factor of 4, i.e. by a factor of the number of sub-word elements for SIMD operations. Quantitative performance measurements for all three architectures are presented in *Section 4*.

*Program 2* is a loop performing addition on every other element of the byte arrays *x* and *y*. The data access pattern in *program 2* can be characterized as FLEX pattern with $\Delta$=2 and $\Psi$=8. For pipelined processor with $SIM_pD$ extensions, assembly programmers have the responsibility to rearrange or pack the vector elements using *mix* and *permute* operations, contributing to instruction overheads. Furthermore, stalls are inevitable in the assembly *program 2* for $SIM_pD$ architectures due to the data dependencies of the re-arrangement instructions on the memory loads, and the data dependencies of SIMD instructions on the re-arrangement instructions. *Figure 14* illustrates the segments of *program 2* for $SIM_pD$ architectures.

```
Prog2 - packed SIMD
        addi    R0, R0, 0x0004
for1:   lw      R3, R2(0x0000)      }
        lw      R4, R2(0x0008)      }  lw: memory loads
        lw      R5, R2(0x0004)      }
        lw      R6, R2(0x000C)      }
        mixr    R7, R3, R4          }
        addi    R2, R2, 0x0004      }  mixr: data re-arrangement
        mixr    R8, R5, R6          }
        addi    R1, R1, 0x0001      }
            <<stall>>
        add_simd R9, R7, R8         }  add simd: SIMD addition
            <<stall>>
        perm    R9, R9, 0x0027      }  perm: data re-arrangement
            <<stall>>
        bne     R2, R0, for1
        sw      R9, R1(0x000F)
endfor1: nop
```

*Figure 14: Assembly instructions for program 2, scheduled for $SIM_pD$ architectures.*

On the other hand, $SIM_dD$ architectures eliminate the problems in $SIM_pD$ architectures by pointer load instructions. No instruction overhead is spent for data rearrangement, but pointers need to be setup for use. The *lptr* instructions will complete execution by the second pipeline stage and so ensure that any data dependencies on the *lptr* instructions are easily solved by data forwarding. As a result, the assembly instructions for $SIM_dD$ architectures are more easily scheduled without any stalls (refer *Figure 15(c)*).

For random data access patterns with arbitrary values for $\Delta$ and $\Psi$, we anticipate even less instruction overhead for $SIM_dD$ architectures. Note that the same number of memory loads (as many as the number of sub-word elements for SIMD operations) is needed to retrieve the vector elements that could potentially reside in arbitrary locations. However, two simple pointer setup instructions are sufficient to locate the disjoint vector elements for $SIM_dD$ architectures. On the other hand, many more *mix* and *permute* instructions are required to properly arrange the disjoint vector elements for $SIM_pD$ architectures, contributing to instruction overhead. In addition, the ISA of general-purpose processors need to be extended with many versions of *mix* and *permute* instructions to handle complex data re-arrangement. These factors reinstate the advantage of $SIM_dD$ architectures in handling flexible data accesses with minimal overhead.

## 4. Performance Evaluation

VHDL models are developed for all the three architectures in consideration. Functional simulations of the architectures are performed using the two programs discussed in *Section 3.1.2*. *Table 5* lists the performance of the three architectures in terms of clock cycles.

| ***Processor Architecture*** | ***Simulation Results for Program 1*** | | ***Simulation Results for Program 2*** | |
|---|---|---|---|---|
| | *Clock Cycles* | *Speedup* | *Clock Cycles* | *Speedup* |
| Basic 5-stage pipelined (reference) | 61 cc | 1.00 | 33 cc | 1.00 |
| Pipelined with $SIM_pD$ extensions | 19 cc | 3.21 | 21 cc | 1.57 |
| Pipelined with $SIM_dD$ extensions | 19 cc | 3.21 | 17 cc | 1.94 |

*Table 5: Performance evaluation of pipelined processor with and without $SIM_pD$ or $SIM_dD$ extensions.*

Simulation results show a performance speedup for both types of SIMD extensions with respect to the basic pipelined processor. This is because the number of loop iterations is reduced (roughly) by a factor up to the number of sub-word elements for SIMD operations. The $SIM_pD$ and indirect-$SIM_dD$ processors have the same performance (speedup of 3.2) for *program 1* where the vector elements reside in sequential locations. However, much data-rearrangement overhead is incurred in *program 2* for the $SIM_pD$ pipelined processor, when the vector elements do not come from sequential locations, but with a constant vector stride. The indirect-$SIM_dD$ pipelined processor eliminates much of this overhead by pointer addressing, achieving a total speedup of 1.94 for *program 2* with respect to the basic pipelined processor.

*Prog1 - pipelined*

```
        addi   R0, R0, 0x0008
for1:   lb     R2, R1(0x0000)
        lb     R3, R1(0x0008)
               <<stall>>
        addi   R1, R1, 0x0001
        add    R4, R2, R3
        bne    R2, R0, for1
        sb     R5, R1(0x000F)
endfor1: nop
```

*Prog2 - pipelined*

```
        addi   R0, R0, 0x0008
for1:   lb     R3, R2(0x0000)
        lb     R4, R2(0x0008)
        addi   R1, R1, 0x0001
        addi   R2, R2, 0x0002
        add    R5, R3, R4
        bne    R2, R0, for1
        sb     R5, R1(0x000F)
endfor1: nop
```

(a)

*Prog1 - packed SIMD*

```
        addi       R0, R0, 0x0008
for1:   lw         R2, R1(0x0000)
        lw         R3, R1(0x0000)
        addi       R1, R1, 0x0004
               <<stall>>
        add_simd   R4, R2, R3
        bne        R1, R0, for1
        sw         R4, R1(0x000C)
endfor1: nop
```

*Prog2 - packed SIMD*

```
        addi       R0, R0, 0x0004
for1:   lw         R3, R2(0x0000)
        lw         R4, R2(0x0008)
        lw         R5, R2(0x0004)
        lw         R6, R2(0x000C)
        mixr       R7, R3, R4
        addi       R2, R2, 0x0004
        mixr       R8, R5, R6
        addi       R1, R1, 0x0001
               <<stall>>
        add_simd   R9, R7, R8
               <<stall>>
        perm       R9, R9, 0x0027
               <<stall>>
        bne        R2, R0, for1
        sw         R9, R1(0x000F)
endfor1: nop
```

(b)

*Prog1 - disjoint SIMD*

```
        addi       R0, R0, 0x0008
for1:   lw         R2, R1(0x0000)
        lw         R3, R1(0x0000)
        addi       R1, R1, 0x0004
               <<stall>>
        add_simd   R4, R2, R3
        bne        R1, R0, for1
        sw         R4, R1(0x000C)
endfor1: nop
```

*Prog2 - disjoint SIMD*

```
        addi       R0, R0, 0x0004
for1:   lw         R2, R1(0x0000)
        addi       R1, R1, 0x0001
        lw         R4, R1(0x0008)
        lw         R3, R1(0x0000)
        lw         R5, R1(0x0008)
        lptr       P0, R2(0), R2(2), R3(0), R3(2)
        lptr       P1, R4(0), R4(2), R5(0), R5(2)
        addi       R7, R7, 0x0004
        add_simd_wp R9, R7, R8
        bne        R7, R0, for1
        sw         R6, R7(0x000C)
endfor1: nop
```

(c)

*Figure 15: Hand-coded and scheduled assembly instructions for Program 1 and 2 for (a) the basic pipelined processor, (b) the pipelined processor extended with SIM$_p$D features and (c) the pipelined processor extended with indirect-SIM$_d$D features.*

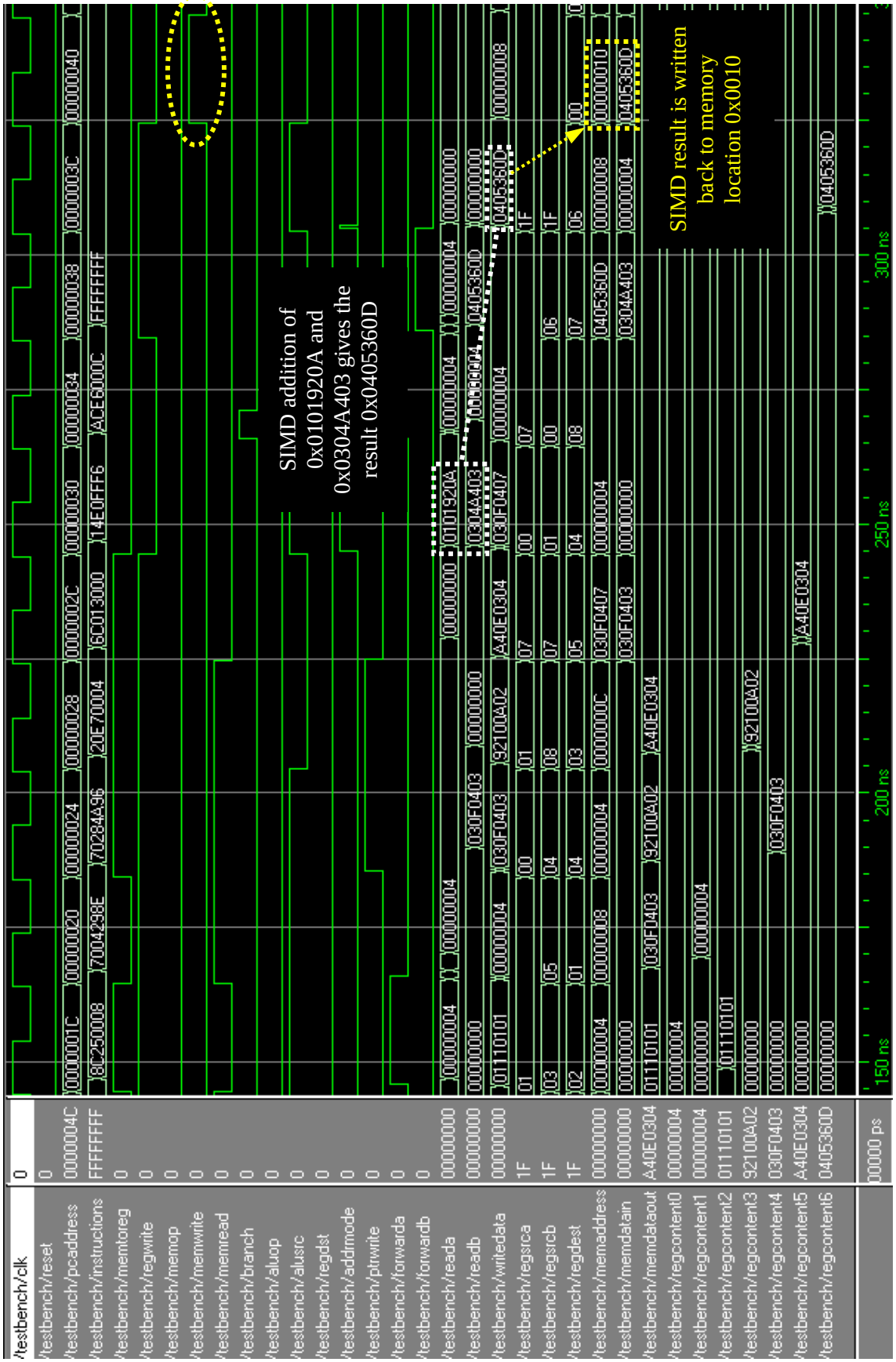


*Figure 16: VHDL Simulation of Program 2 for indirect-SIM$_d$D pipelined processor.*

## 5. Conclusion

We briefly discussed the VLIW and SIM$_p$D architectures that improve execution throughput by exploiting data-level parallelism inherent in programs. VLIW instructions specify multiple operations to be performed on multiple disjoint data streams. On the other hand, SIM$_p$D architectures specify a single operation to be performed on multiple packed data elements. Motivated by the flexible data access patterns found in commonly used telecommunication and multimedia kernels, SIM$_d$D architectures combine the features of both VLIW and SIM$_p$D architectures by specifying a single operation to be performed on multiple *disjoint* data streams. The indirect-SIM$_d$D implementation is proposed to achieve simplicity in extending the ISA of general-purpose processors. Indirect addressing using pointers allow the disjoint vector elements to be composed dynamically, and thus achieve much flexibility in data accesses without the data re-arrangement overhead of SIM$_p$D instructions.

We also considered the architectural impact of SIM$_p$D and indirect-SIM$_d$D extensions on a variant of general-purpose processors, the basic five-stage pipelined, MIPS-like processor. The architectural impact is considered from both the hardware and software perspectives. We saw that the additional hardware required to implement the SIMD extensions is minimal. On the other hand, the assembly instructions for SIM$_p$D architectures are harder to schedule without processor stalls in comparison to the assembly instructions for indirect-SIM$_d$D architectures.

Finally, VHDL models developed for the SIM$_p$D and indirect-SIM$_d$D pipelined processors are simulated using two simple loop programs that have different data access patterns, characterized by the metrics *vector element distance* and *vector stride*. Simulation results show a performance speedup for both types of SIMD extensions with respect to the basic pipelined processor. This is because the number of loop iterations is reduced (roughly) by a factor up to the number of sub-word elements for SIMD operations. However, much data re-arrangement overhead is incurred by the SIM$_p$D instructions when the vector elements do not come from sequential locations. The indirect-SIM$_d$D pipelined processor eliminates much of this overhead by specifying the disjoint locations of vector elements using pointers. We conclude that the indirect-SIM$_d$D architectures are more efficient in handling flexible data accesses; yet leaving unchanged the performance of programs with SIM$_p$D data accesses.

## References

1. IBM eLite DSP Project http://www.research.ibm.com/elite/elite_dsp_project.html

2. A New Look at Exploiting Data Parallelism in Embedded Systems
   *Hillery C. Hunter, Jaime H. Moreno, International Conference on Compilers, Architectures and Synthesis for Embedded Systems 2003.*

3. A High-Performance Embedded DSP Core with Novel SIMD features
   *Jeff H. Derby and Jaime H. Moreno, International Conference on Acoustics, Speech, and Signal Processing 2003.*

4. COMP3211 Computer Architecture Assignment 2 Report: Pipelined Processor Design
   *Weng Mun Au Yong, Lih Wen Koh, Seng Lin Shee, S2 2002.*

5. UC Berkeley Vector IRAM project http://iram.cs.berkeley.edu/

6. Intel MMX Technology http://www.intel.com