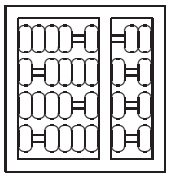


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

The 2D-VLIW Architecture

R. Santos *R. Azevedo* *G. Araujo*

Technical Report - IC-06-006 - Relatório Técnico

March - 2006 - Março

The 2D-VLIW Architecture

Ricardo Santos*† Rodolfo Azevedo† Guido Araujo†

Abstract

This technical report presents the main aspects of the 2D-VLIW architecture. This architecture is being developed in the Computer Systems Laboratory at the Institute of Computing of the State University of Campinas. The main purpose of the 2D-VLIW architecture is to provide resources to carry out a broad class of applications through a compromise between architectural arrangement and compiler assistance. Some preliminary experiments have shown that this combination can achieve an appealing performance over mainstream architectures like VLIW and EPIC.

1 Introduction

For many years, the constant increase in processor performance came from the advancement in VLSI process technology together with the improvement in architectural design. However, recent announcements on the technology limits due to the thermal barrier have motivated the research into innovative ways to sustain the increase in performance.

Previous work [11, 13, 18] as well as state-of-the-art architectures [2, 14–17] have pointed out towards architectures adopting new strategies for code optimization and execution models. Advanced compiler optimizations could look at large code regions to find out greater parallelism levels. The execution models could provide new resource arrangements to execute large code regions, thus increasing performance.

In this work, we present a general description of the 2D-VLIW architecture. Our architecture is named 2D-VLIW because large instructions, comprised of single operations, are fetched from the memory and executed into a (two-dimensional) matrix of functional units through a pipeline. This processing is aided by a set of local registers spread across the matrix. The matrix has a static and sparse interconnection network which connects two functional units at consecutive levels. This arrangement is independent of the running program but, on the other hand, is flexible enough to execute a broad class of applications. The 2D-VLIW architecture is being developed in the Computer Systems Laboratory (LSC) at the Institute of Computing-UNICAMP.

This technical report is organized as follows. Section 2 introduces the 2D-VLIW datapath. Section 3 details the 2D-VLIW functional unit. Sections 4 and 5 describe the temporary register files and the interconnection between FUs, respectively. The instruction

*Computer Engineering Department, Dom Bosco Catholic University, 79117-900 Campo Grande, MS.
†Institute of Computing, University of Campinas, 13081-970 Campinas, SP.

format is presented in Section 6. Section 7 exemplifies the 2D-VLIW execution paradigm. Finally, Section 8 summarizes the main features of this architecture and it presents new directions to be performed as future work.

2 The 2D-VLIW Architecture

The 2D-VLIW architecture exploits instruction level parallelism (ILP) through a pipelined multiple-issue datapath and a two dimensional VLIW execution model. This architecture fetches large instructions from the memory as well as VLIW and EPIC architectures do. Contrary to the instructions in these architectures, 2D-VLIW instructions are encoded in order to reduce the instruction fetch delay and to minimize the I-cache area.

The 2D-VLIW instructions are comprised of single operations executed by a set of functional units (FUs) organized as a $M \times N$ matrix, where M is the number of rows and N is the number of columns. The number of operations in one instruction is equivalent to the amount of FUs in the matrix. In the execution stage, these operations are dispatched to functional units on a per column basis. Figure 1 shows a simplified overview of a 4×4 2D-VLIW datapath.

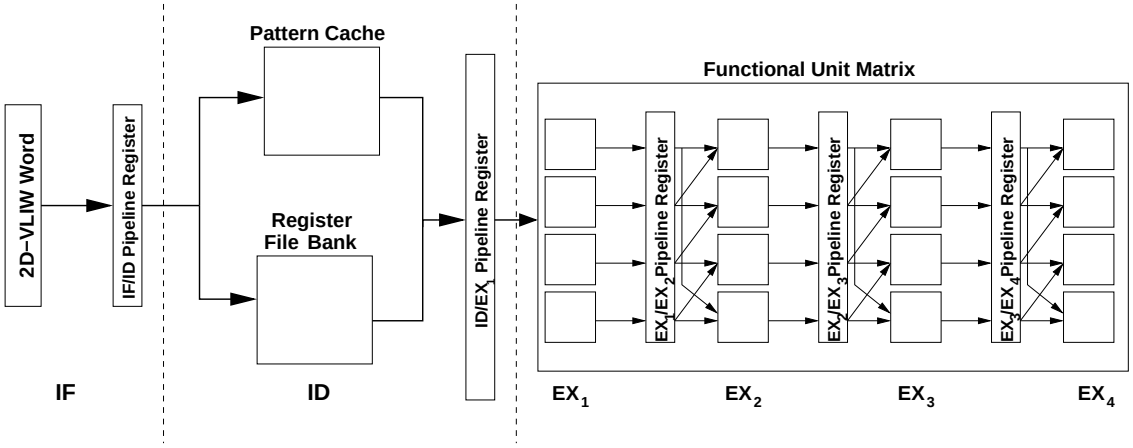


Figure 1: A simplified overview of the 2D-VLIW datapath.

At each clock cycle, one 2D-VLIW instruction is fetched from the memory and pushed into the IF/ID pipeline register. In the decode stage, an encoded instruction is decoded by the pattern-cache and, at the same time, the global register operands come from the encoded instruction are read from the global register file. After that, an whole 2D-VLIW instruction is ready to be executed into the functional unit matrix. The FU matrix presented in Figure 1 allows four operations to be executed at each execution stage EX_1, EX_2, EX_3, EX_4 of the pipeline. FUs are generic enough to enable any operation to be executed. Notice that, by using the interconnection network, each FU in column n, $1 \leq n < 4$, can provide operands to two FUs in the following column $n + 1$.

The 2D-VLIW architectural organization transfers several tasks to the compiler, such as: DAG creation, register allocation, instruction scheduling, instruction encoding, and hazards solution. In addition, the intrinsic features of this architecture makes the compiler task more difficult. In order to have a compiler platform able to perform code generation from real world programs to the 2D-VLIW architecture, the Trimaran compiler [3] has been adopted. Trimaran has been widely adopted as the basic compiler infrastructure for many projects related to instruction-level parallelism (ILP) exploration. Specifically, the most attractive features that we have found in Trimaran were the ability to handle hyperblocks and a toolset to simulate the program execution on multiple-issue architectures. Hyperblocks gather several basic blocks together in one unit. This matches architectures like 2D-VLIW very well, given that hyperblocks can provide enough operations to fill the functional units of the matrix. The simulation tool provides a set of resources to run a program over a multiple-issue architecture described by the HMDES language [6]. We have developed new features on the HMDES in order to allow it to describe the 2D-VLIW architecture.

3 The Functional Unit

The functional unit matrix is the main component of the 2D-VLIW architecture. This matrix is composed by generic FUs which enable any operation of the instruction set to be executed. Figure 2 depicts all logic blocks and signals inside a FU as well as its interconnection to temporary registers, global register file and the functional unit register.

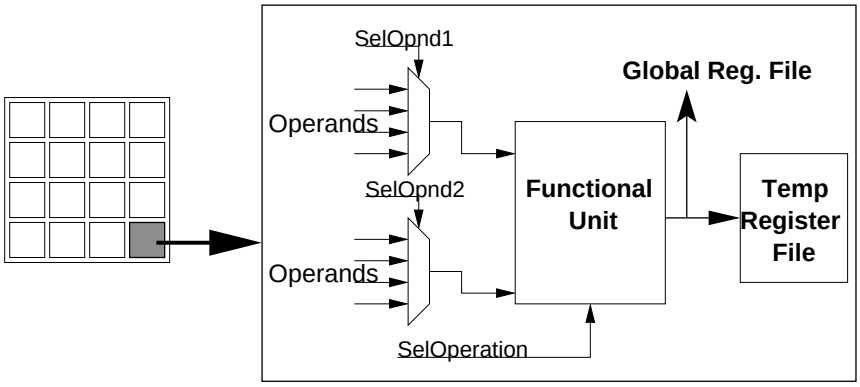


Figure 2: The 2D-VLIW functional unit.

Results from a FU may be written into a *Temp Register File* (TRF) or into the *Global Register File* (GRF). TRF is a small register bank with, typically, 2 local registers dedicated to each FU. Our experiments based on programs of the SPECint00 benchmark showed that two local registers are enough to enable most ($> 90\%$) of the temporary FU computation storage. TRF is used to minimize the pressure over the global register file, such that temporary results are available to FUs through the interconnection network. Moreover, TRF reduces the pressure on the number of read and write ports in the GRF. The result

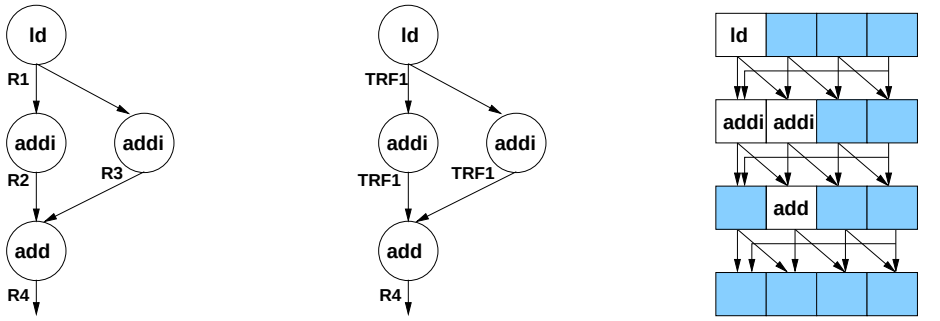
from an operation is always written into an internal register called *FU Register*. By using TRF and *FU Register*, a result from an operation in cycle i will be available for three other operations in the next cycle $(i + 1)$: two operations can use this result through the *Temp Register* whereas another operation uses it from the *FU Register*. *Operands* input data come from three possible sources: the global register file, the temporary register file through the interconnection network, or from the own FU by the *FU Register*. The *SelOpnd*1, *SelOpnd*2 and *SelOperation* input selection signals come from the pipeline registers.

4 The Temporary Register File

The adoption of TRFs in the 2D-VLIW architecture was motivated by the results in [9, 10]. The authors present many reasons why architectures with FUs, buses and Register Files (RFs), called TTAs (Transport Triggered Architectures), need fewer RF ports. The experimental results of this work reveal that, on these architectures, each operation on average, demands 0.5 register source and 0.35 register result. Futhermore, another experiment showed that reducing the GRF by 42%, the OPC is decremented only 4%, i.e, even having only 58% of the total number of global registers, the architecture achieves 96% of its optimal OPC.

The TRFs in each FU are a viable alternative to minimize the pressure over the global register file. This is achieved by storing temporary values (of the DAGs' inner nodes) into TRFs. The DAG leaves' read values from the global register file, while the operations inside the DAGs read and write values from/to the temporary register. The DAG root nodes write values into the global register file. Figure 3 shows how TRFs can be allocated to a DAG and the final mapping in the FU matrix. The load operation, *ld*, uses one value from the global register file, and needs to store its result into a register which will be used by the two *addi* operations. The results from these two operations are the inputs to *add*, which stores the result of the addition into the $R4$ global register. The DAG in 3(a) requires two read ports and two write ports to access global registers. In Figure 3(b), the results from the operations *ld* and *addi* are stored into the TRFs. Thus, the DAG demands just one read port and one write port from/to the GRF. Figure 3(c) shows the mapping of this DAG onto the matrix.

In general, the area upper bound complexity of a standard register file is given by $O(n^2)$ [1] whereas the latency upper bound is $O(log\ m)$, where n is the total number of read and write ports, and m the number of read ports. Notice that the number of read ports has an impact upon the area and latency of the register file. Multiple-issue architectures have the number of read ports bounded by $O(k)$ (usually $2 \times k$), where k is the amount of functional units in the architecture. By adopting TRFs and assuming that only leaves/roots from the DAGs will need to read/write from/to the global register file, we found out that the latency upper bound for our architecture GRF can be asymptotically reduced. This new upper bound is $O(\sqrt{k})$ (usually $2 \times \sqrt{k}$). For example, a 16-FUs EPIC architecture has 16×2 GRF read ports, while the equivalent 2D-VLIW (4×4 matrix) has only 8 read ports which leads to an area 4 times smaller, and a latency reduction of 40%, when taking into account that the same number of write ports is available for both architectures. It is



(a) One DAG using GRFs. (b) One DAG using TRFs. (c) The DAG (b) mapped onto the UF matrix.

Figure 3: Example of a DAG using TRFs.

important to notice, however, that the 2D-VLIW architecture has less write ports to the GRF since only DAGs' roots store their results into the global registers.

One could consider that this constraint on the read ports could affect the program size and the final performance. However, we have performed trace-driven experiments, based on SPECint00 and Trimaran programs, and figured out that this approach does not increase the program size, when compared to a model without restriction. By reducing the number of GRF read ports, we are considering that the maximum number of leaves scheduled on the matrix, at any specific time, is asymptotically bounded by the number of FUs in one column. Actually, if the DAG leaves' have only one input operand in the GRF, we could have more leaves, scheduled on the matrix, than the FUs in one column. In order to make this possible, we perform scheduling and allocation tasks over entire DAGs. The compiler identifies temporary values in the DAG and stores them into TRFs instead of GRFs. Moreover, the compiler platform prefers to create large DAGs because they increase the matrix utilization and, as a consequence, improve the parallelism rate. Small DAGs, on the other hand, decrease the matrix utilization due to the constraints on the GRF read ports. Besides reducing the GRF pressure, local registers eliminate the need of multiple-banked register files that, as pointed in [5] brings flexibility at the expense of IPC (instructions per cycle) decrease.

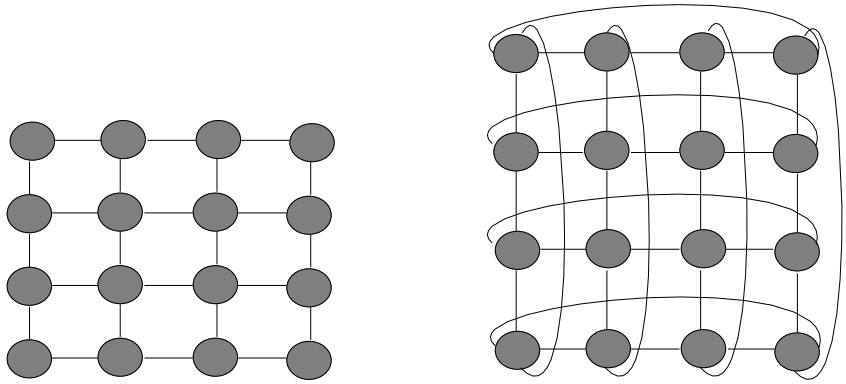
5 The Interconnection Network

It is well-known that the interconnection network heavily affects the performance and the area size in any architecture. For example, a completely connected network [7] would make possible a communication between any processor elements. However, that network model uses $N(N-1)/2$ (N=number of nodes/functional units/processor elements) communication links and, as a result, it greatly increases the architecture area. In this situation, a narrower model is a most suitable option.

Our interconnection network was inspired by classical static networks like mesh and torus networks [7]. A k-dimensional mesh with $N = n^k$ nodes has an interior node degree

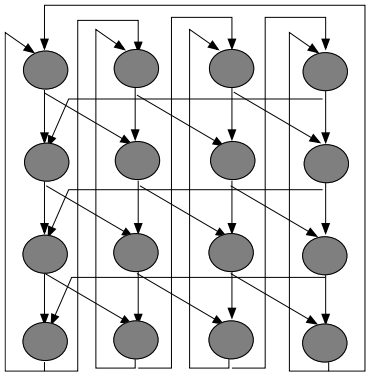
of $2k$, a network diameter of $k(n-1)$, and it is not symmetric since the node degrees at the boundary and corner nodes are 3 and 2, respectively. A torus interconnection has ring connections along each row and along each column of the array. In general, an $n \times n$ binary torus has a node degree of 4, a diameter of $2\lfloor n/2 \rfloor$, and it is a symmetric topology. All added wraparound connections help reduce the diameter by one-half from that of the mesh.

The 2D-VLIW FUs are connected through a static and sparse interconnection network where a FU in column n can read values from two FUs in column $n-1$ and it sends its results to two FUs in column $n+1$. The FUs degrees are symmetric since all the FUs read values from two others functional units. FUs at the same column do not communicate between themselves. This interconnection style is suitable to pipeline computation models because previous operations produce values to the next ones. Figure 4 shows two basic interconnection networks (Mesh and Torus) and the 2D-VLIW interconnection model.



(a) Mesh

(b) Torus.



(c) The 2D-VLIW interconnection network

Figure 4: Mesh, Torus and our interconnection model.

Due to its simplicity, this interconnection model makes the interconnection allocation easy. Moreover, the scheduler can take advantage of the interconnection model topology in

order to schedule entire DAGs on the matrix. Taking into account that DAG operations (nodes) have at most two inputs and DAG edges join operations at consecutive levels, many DAGs can be perfectly mapped onto the FU matrix. Our studies about the DAGs structure in SPECint00 and Trimaran programs revealed that $60\% - 80\%$ of the DAGs do not have any bypassing edge. Even if a DAG has bypassing edges that go beyond the reach of the available interconnections, our compiler strategy (based on a subgraph matching heuristic [4]) can assure the right execution by splitting up the source and destination operations between two 2D-VLIW instructions.

6 2D-VLIW Instruction Format and the Pattern Cache

A 2D-VLIW instruction is comprised of single operations. These operations looks like a single RISC instruction since each operation has a fixed format and it performs single actions. Specifically, the 2D-VLIW ISA (instruction-set architecture) is a subset of the MIPS ISA [8]. It is important to notice that apart from grouping parallel operations, a 2D-VLIW instruction has dependent operations. Parallel operations are executed on the same column FUs while dependent operations are executed on different columns of the functional unit matrix.

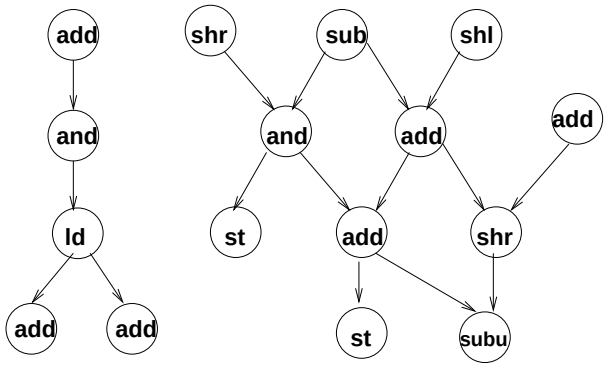
After scheduling and register allocation phases, the compiler creates 2D-VLIW instructions from the operations DAGs as depicted in Figure 5. Figure 5(a) shows a code fragment so that register allocation was already performed and local and global registers are identified as `trX` and `rX`, respectively, where `X` is the number of the register. The operations DAGs in 5(b) are scheduled to compose a 2D-VLIW instruction as presented in 5(c). For the sake of simplicity we have presented the 2D-VLIW instruction as a matrix of operations where each cell represents one operation that is executed by one functional unit of the FU matrix.

In order to minimize the impact of the 2D-VLIW instruction size on the i-cache, the 2D-VLIW architecture adopts an instruction encoding technique. This technique extracts operand patterns from the large instructions and it stores these patterns into the pattern-cache (p-cache) whereas the encoded instructions are stored into the main memory. The technique works as follows: (1) at compiler-time the LIF (Long Instruction Factorization) algorithm factors out redundant data from the instructions and creates patterns from these redundant data. Moreover, the algorithm takes the architectural constraints into account whenever encoding 2D-VLIW instructions; (2) at runtime the processor fetches a 2D-VLIW instruction, decodes it, and fetches the p-cache line associated to that word. It then provides the operands from the 2D-VLIW word, the DAG structure (interconnections) and FU types from the fetched p-cache line to the matrix.

The encoded 2D-VLIW instruction carries operands (registers and immediate values), and a tag pointing to a pattern-cache line. By using this encoding strategy, a 2D-VLIW instruction can be seen as a function call in which the arguments are encoded into its fields, and the operations are encoded into a pattern cache-line. Figure 6(a) shows the 2D-VLIW instruction from Figure 5(c) after applying our encoding technique. The respective pattern-cache line is also showed.

```
add tr1, r1, r2
and tr3, tr1, r2
ld tr1, tr3, 10
add r5, tr1, r4
add r3, tr1, r6
shr tr2, r1, r5
sub tr3, r4, r5
shl tr4, r1, r5
and tr1, tr2, tr3
add tr2, tr3, tr4
add tr3, r7, r9
st tr1, 9
add tr1, tr1, tr2
shr tr2, tr2, tr3
st tr1, 9
subu r9, tr1, tr2
```

(a) Code fragment.

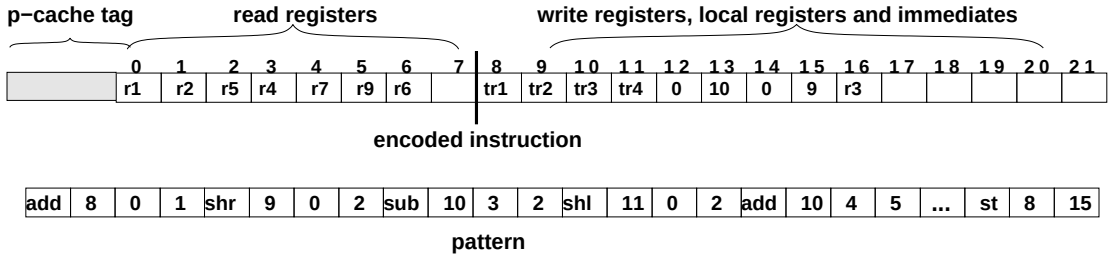


(b) DAGs representing the code fragment.

add tr1, r1, r2	shr tr2, r1, r5	sub tr3, r4, r5	shl tr4, r1, r5
add tr3, r7, r9	and tr3, tr1, r2	and tr1, tr2, tr3	add tr2, tr3, tr4
shr tr2, tr2, tr3	ld tr1, tr3, 10	st tr1, 9	add tr1, tr1, tr2
subu r9, tr1, tr2	add r5, tr1, r4	add r3, tr1, r6	st tr1, 9

(c) A 2D-VLIW instruction.

Figure 5: Code fragment and DAGs.



(a)

Figure 6: The encoded 2D-VLIW instruction and its pattern-cache line.

Taking into account that 2D-VLIW instructions have 16 32-bits operations, the instruction in 5(c) has 512 bits. On the other hand, an encoded 2D-VLIW instruction has 128 bits divided into 8 fields to read registers, 14 fields to write operands, local registers, immediate values, and one field to the p-cache tag. The encoded instruction is 4 times smaller (compression ratio of 25%) than the non-encoded instruction. In this architecture, a p-cache line has 496 bits.

Clearly, a p-cache line plus an encoded I-cache line has more bits ($496 + 128 = 624$ bits) than a non-encoded I-cache line (512 bits). However, some theoretical experiments over encoded instructions of SPECint programs showed that there is a surjective function between encoded I-cache elements and p-cache elements. This surjection leads to a p-cache smaller than an encoded I-cache. As a result, the area occupied by the encoded I-cache plus a p-cache can be smaller than a non-encoded I-cache.

7 The 2D-VLIW Execution Model

As presented in Section 2, program execution over the 2D-VLIW matrix is pipelined. At each clock cycle, one 2D-VLIW instruction is fetched from the memory and pushed into the pipeline stages. On the execution stages, the operations from this instruction are executed according to the number of FUs in each column. Figure 7 shows some DAGs extracted from the 181.*mcf* program and their respective 2D-VLIW instructions (without encoding). The 2D-VLIW architecture adopted has 16 functional units organized as a 4×4 matrix. The 181.*mcf* program is part of the SPECint00 benchmark and it was compiled by the Trimaran compiler with the hyperblock option on. The operations from these DAGs correspond to a subset of the HPL-PD [12] ISA. A particular feature from these operations is the multiple operand semantic depending on the context. For example, it is common to assume that all operands from the operation *and* are registers. In the HPL-PD architecture, on the other hand, the operation *and* (label & in Figure 7), has one input operand coming from the *ld* whereas the second operand is an immediate value. Due to this, it was possible to schedule all operations from the DAGs of 7(a) just using two 2D-VLIW instructions, respecting the read ports $O(\sqrt{k})$ bound as presented in Section 4.

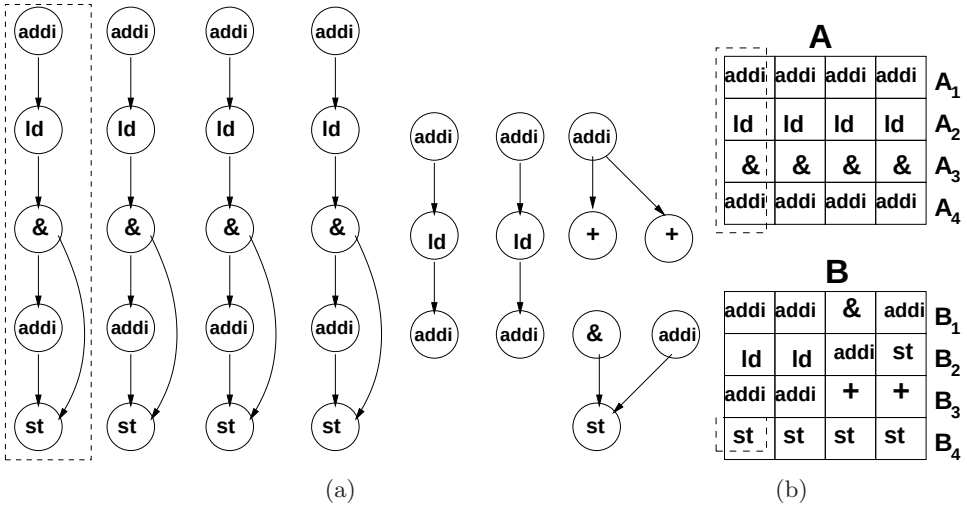


Figure 7: DAGs from the 181.mcf program and their equivalent 2D-VLIW instructions.

Figure 7(a) shows several DAGs from the 181.*mcf* program (from an unrolled inner loop) and 7(b) shows these DAGs scheduled into two 2D-VLIW instructions A and B. Notice that operations with read-after-write (RAW) dependencies are located in two different rows while independent operations are at the same row. Due to architectural constraints, some DAGs (dashed region) were split up between the instructions A and B. For the sake of the simplicity we left out operand information from these instructions.

Figure 8 depicts all execution steps of the 2D-VLIW instructions A and B from Figure 7(b) on the 2D-VLIW datapath. Since the decode and instruction fetch stages work like in a standard processor, we start our example at the EX_1 execution stage. After the ID/EX_1 pipeline register has been filled in, the execution starts over the matrix.

Figure 8(a) represents the first execution cycle on the FU matrix. The first column receives data from the ID/EX_1 pipeline register. Four functional units from the first column execute operations *addi*, *addi*, *addi*, and *addi* from row A_1 (instruction A). The dashed arrows indicate which FUs receive the results from these operations. Obviously, the consumers FUs are limited by the interconnection network. At the second execution cycle, 8(b), operations *ld*, *ld*, *ld*, and *ld* from A_2 are carried out on the second column. At the same time, operations from B_1 start their execution on the first column by running the operations *addi*, *addi*, &, *addi*. In the third execution stage, 8(c), the EX_2/EX_3 pipeline register carries information to execute operations &, &, &, & from A_3, the FUs in the second column are executing the operations *ld*, *ld*, *addi*, *st* from B_2 and the first row C_1, from a 2D-VLIW instruction C, started its execution on the matrix. In the fourth execution stage, 8(d), the operations *addi*, *addi*, *addi* and *addi* from A_4 are executed on the fourth column, the operations *addi*, *addi*, *add* and *add* from B_3 are executed in the third column, operations from C_2 are running in the second column and the instruction D (column D_1) started its execution. The last execution is shown in 8(e) where the four *st* operations from B_4 are being executed and the instruction A has already been finished. Following the pipeline execution, in the fourth execution cycle the matrix is totally filled with operations from four different 2D-VLIW instructions as indicated by the first, second, third and fourth columns (highlighted differently) in 8(d). After the fourth execution cycle, one 2D-VLIW instruction is finished at every cycle. Notice also that operations being executed on 8(a), 8(b), 8(c), 8(d), and 8(e) represent exactly all the operations from the 2D-VLIW instructions A and B in Figure 7(b). Looking at the operations and the interconnection used through the step-by-step execution in 8(a)-8(e), we can rebuild all DAGs from Figure 7(a).

8 Conclusions and Future Work

A new multiple-issue architecture called 2D-VLIW was presented in this work. This architecture fetches encoded instructions from the memory and it runs the operations onto the matrix of functional units. These functional units are connected by a static, sparse interconnection network. Furthermore, there are local registers spread across the matrix which are used to keep temporary values and to reduce the pressure over the global register file.

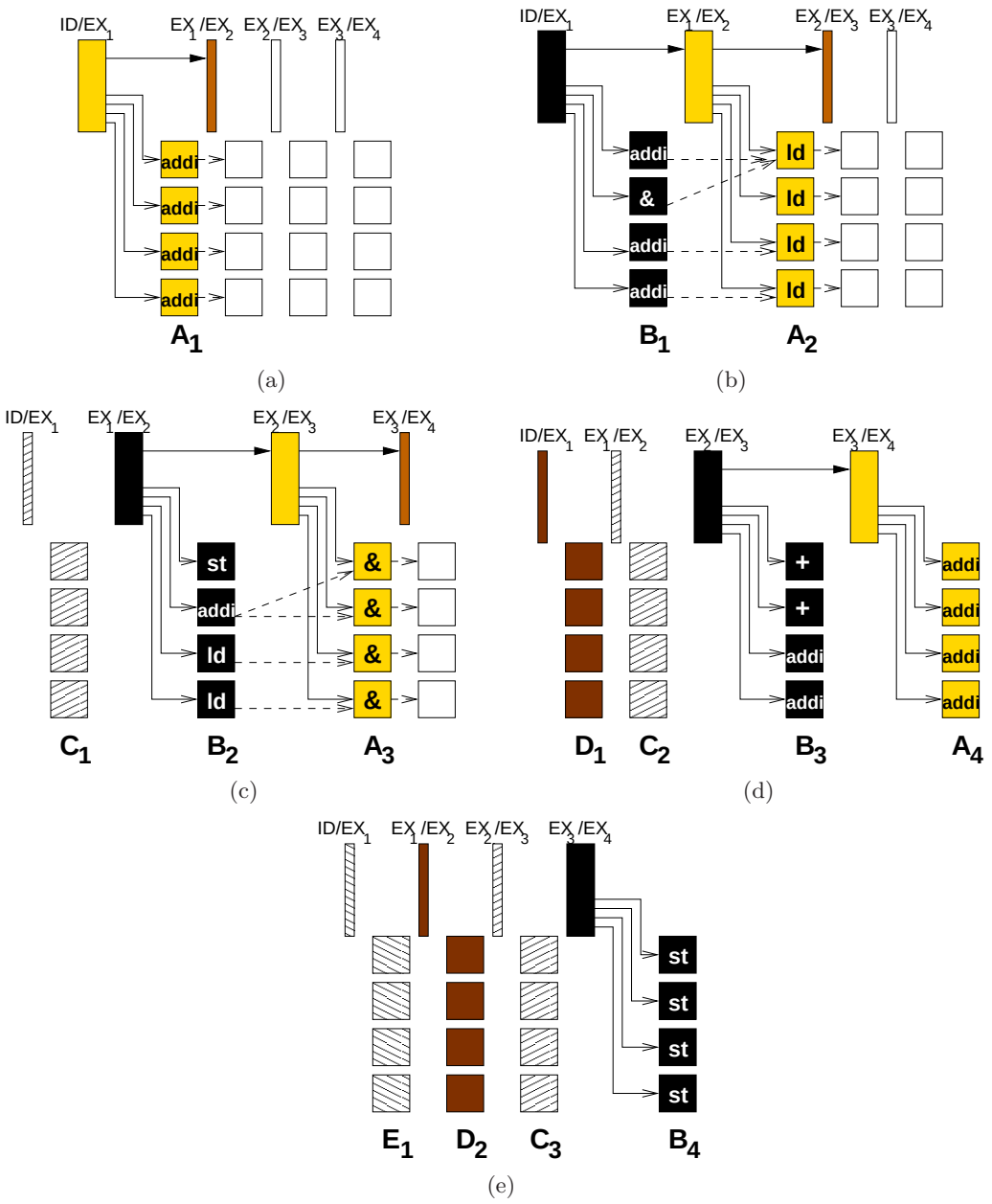


Figure 8: Execution stages on the 2D-VLIW architecture.

2D-VLIW presents an interesting combination of architectural arrangement, execution model and compiler assistance. 2D-VLIW minimizes the delay of fetching large instructions by factorizing patterns and put them on a pattern-cache inside the architecture datapath. Another key feature is the presence of local registers which store temporary values, minimizing thus the pressure over the global register and, as a result, reducing the occurrences of

spill code. The compiler plays an important role in the 2D-VLIW architecture. It performs several tasks such as: encoding large instructions, recognizing large pieces of computation, mapping their operations onto the matrix topology and using the local register to keep temporary values.

Currently, we started the implementation of two new scheduling and register allocation algorithms for 2D-VLIW. We already have a simulator and an assembler tool. The p-cache and the LIF algorithm were already implemented in our simulation infrastructure. We are finishing up a conversion tool from ELCOR (Trimaran's intermediate representation) to our assembler. For the next few months, we are going to develop a hardware prototype of this architecture as well as evaluate the 2D-VLIW performance over different application domains like multimedia applications.

References

[1] A. Abnous and N. Bagherzadeh. Architectural Design and Analysis of a VLIW Processor. *International Journal of Computers and Electrical Engineering*, 21(2):119–142, 1995.

[2] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burril, R. G. Mcdonald, and W. Yoder. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, 2004.

[3] L. N. Chakrapani, J. Gyllenhaal, W. Mei, W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran - An Infrastructure for Research in Instruction-Level Parallelism. *Lecture Notes in Computer Science*, 3602:32–41, 2004.

[4] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Subgraph Transformations for the Inexact Matching of Attributed Relational Graphs. *Computing*, 12:43–52, 1998.

[5] J. L. Cruz, A. Gonzales, M. Valero, and N. P. Topham. Multiple-Banked Register File Architectures. In *Proceedings of the* 27^{th} *Annual International Symposium on Computer Architecture (ISCA)*, pages 316–325, Vancouver, 2000. ACM Press.

[6] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3, Center for Reliable and High-Performance Computing - University of Illinois at Urbana-Champaign, Urbana-Champaign-Illinois, 1996.

[7] K. Hawng. *Advanced Computer Architecture*. McGRAW-HILL, 1993.

[8] J. Henessy and D. A. Patterson. *Computer Organization and Design - A Hardware/Software Interface*. Morgan Kaufmann, San Francisco, 3 edition, 2004.

[9] J. Hoogerbrugge and H. Corporaal. Register File Port Requirements of Transport Triggered Architectures. In *Proceedings of the* 27^{th} *Annual International Symposium on Microarchitecture (MICRO)*, pages 191–195. ACM Press, December 1994.

[10] J. Janssen and H. Corporaal. Partitioned Register File for TTAs. In *Proceedings of the* 28^{th} *Annual International Symposium on Microarchitecture (MICRO)*, pages 303–312. IEEE Computer Society Press, December 1995.

[11] N. P. Jouppi. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the* 3^{rd} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 272–282. ACM Press, May 1989.

[12] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL-PD Architecture Specification. Technical Report 93-80, Hewlett Packard Laboratories Palo Alto, February 2000.

[13] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the* 19^{th} *Annual International Symposium on Computer Architecture (ISCA)*, pages 46–57. ACM Press, May 1992.

[14] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the* 8^{th} *International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 46–57. ACM Press, 1998.

[15] B. Mei, A. Lambrechts, J-Y Mignolet, D. Verkest, and R. Lauwereins. Architecture Exploration for a Reconfigurable Architecture Template. *IEEE Design & Test of Computers*, 22(2):90–101, 2005.

[16] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *Proceedings of the* 30^{th} *IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 422–434. ACM Press, May 2003.

[17] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Argawal. Baring it All to Software: RAW Machines. *IEEE Computer*, 30(9):86–93, 1997.

[18] D. W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the* 4^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 176–188. ACM Press, April 1991.