The Case for VLIW-CMP as a Building Block for Exascale

Bruce Jacob, *Senior Member, IEEE*

Abstract—Current ultra-high-performance computers execute instructions at the rate of roughly 10 PFLOPS (10 quadrillion floating-point operations per second) and dissipate power in the range of 10 MW. The next generation will need to execute instructions at EFLOPS rates—100× as fast as today's—but without dissipating any more power. To achieve this challenging goal, the emphasis is on power-efficient execution, and for this we propose VLIW-CMP as a general architectural approach that improves significantly on the power efficiency of existing solutions. Compared to manycore architectures using simple, single-issue cores, VLIW-CMP reduces both power and die area, improves single-thread performance, and maintains aggregate FLOPS per die. To improve further on the power advantages of VLIW, we describe a mechanism that reduces power dissipation of both data forwarding and register-file activity.

1 Implications of the Obvious

Moving to exascale, i.e., building a 1 exa-FLOPS computing system (equivalent to 1,000,000 TFLOPS or 1,000 PFLOPS), is limited by how efficiently one can perform a staggeringly large number of operations. It is not really a question of "can we build a machine that executes this many operations per second?" but rather "can we build one and afford to power it?" The power levels for existing supercomputers are barely tolerated as high as they are today, and so a 1 EFLOPS machine tomorrow must not dissipate significantly more power than today's low-double-digit PFLOPS machines, which dissipate on the order of 10 MW. This leads to necessary conditions for exascale that are challenging—such as approaching 1 TFLOPS per Watt at the CPU or core level, and the ability to build a 1–10 PFLOPS rack that dissipates 10–100 kW. For perspective, typical CPUs today execute at roughly 0.01 TFLOPS per Watt, and typical cabinets (racks) today dissipate on the order of 100 kW to produce roughly 0.1 PFLOPS of execution.

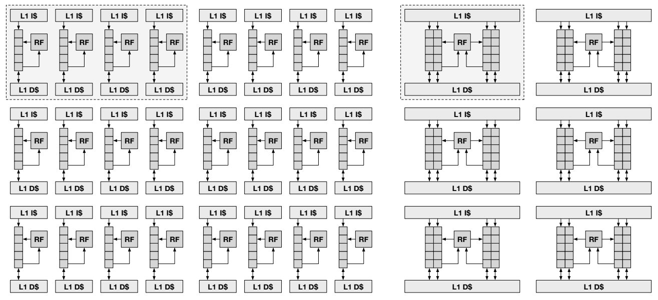
Achieving the desired level of performance efficiency will demand trade-offs between numerous conflicting requirements, such as the following:

- high-performance cores, to meet single-thread performance goals
- low-power cores (in direct conflict with above), to meet stringent energy/power limitations
- small numbers of nodes, to reduce the bandwidth (and thus power) needed of the system interconnect
- large numbers of nodes, to reach 1000 PFLOPS

This paper focuses on a small segment of the vast space that is exascale-level system design. To achieve 100-fold speedup without a commensurate increase in power dissipation will require re-tooling at all levels of the system, from application through operating system, middleware, computer architecture, circuit design, down to process technology. In this paper we focus on the general architecture approach used for the CPU.

Recent CPU trends have shied away from using numerous multi-issue, out-of-order cores on-chip, because those designs dissipate too much power; modern designs have instead favored large numbers of simple, single-issue, in-order cores [2] [10][11][12]. Some architectures use a heterogeneous mix including a small number of high-performance cores, but even then, when these designs tile large numbers (tens to hundreds) of cores in an array, those cores are single-issue, in-order. The well-known downside of this approach is that it sacrifices single-thread performance. Moreover, using numerous cores demands significant bandwidth from main memory to avoid overcrowding the memory channel. One way of looking at this is that, when the computer-design community hit a power wall using complex cores in the early 2000s, we made a 180° turn and went as far as possible in the opposite direction, avoiding the middle ground entirely. As it turns out, this middle ground is perhaps the most power-efficient part of the design space.

• *The author is with the Department of Electrical Computer Engineering, University of Maryland, College Park, MD 20742. E-mail: blj@umd.edu.*
Manuscript received 4 Mar. 2015; revised 28 Mar. 2015; accepted 16 Apr. 2015. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee. org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/LCA.2015.2424699.



(a) Typical manycore CPU organization when tiling homogeneous cores

(b) Proposed VLIW-based manycore CPU organization

Fig. 1. Two manycore arrays, equivalent sub-units highlighted: (a) typical tiling of single-issue, in-order cores, (b) tiling of n-way VLIW cores (n=4)

To illustrate, the typical manycore chip using simple cores looks like the stylized representation in Fig. 1(a): numerous pipelines are tiled across the chip, each with its own L1 data cache, its own L1 instruction cache, and its own register file. An alternate architecture is shown in Fig. 1(b), in which the fundamental building blocks are not single-issue cores but instead n-way VLIW cores, in this example 4-way VLIW cores. Note that, even though the diagram shows each VLIW pipeline having equal access to all resources, this need not be the case: not every pipe needs a port to the data cache; not every pipe needs a hardware multiplier in its ALU; etc. Commercial VLIW designs such as the TMS320C6000 from Texas Instruments have already explored the asymmetric-pipe design space, in which every pipeline accepts a different mix of instruction opcodes, and they have shown it to be viable [17].

With that in mind, there are a few important things to note about this architecture, as it compares favorably with the more orthodox arrangement in Fig. 1(a):

- The number of execution pipelines is the same. Thus, the aggregate performance (chip-wide instructions per clock) is also the same, assuming that the VLIW width n is small enough the compiler can extract near-linear

parallelism (VLIW speedup is effectively linear at small issue widths [3]).
- The number of register files in (b) is lower than the number of register files in (a) by a factor of n. This should improve die area, and it can reduce power dissipation as well, to the extent that the number of read/write ports and/or amount of read/write activity can be reduced. A solution to this is presented later.
- Data forwarding can be expensive in VLIW, and a solution is given later that scales linearly, instead of the quadratic nature of typical VLIW forwarding logic.
- The L1 cache capacity would remain roughly the same, as an n-fold increase in storage per core would likely be offset by the n-way pipeline arrangement. However, the total number of ports could be decreased, as an n-way VLIW would only need one instruction-cache port and fewer than n data-cache ports.
- The on-chip interconnect in (b) would have fewer endpoints by a factor of n than the interconnect in (a); thus its complexity should decrease, potentially improving power dissipation and/or die area.
- Shared caches should have fewer simultaneous threads vying for resources, potentially reducing complexity.

Nearly any VLIW instruction-set architecture is a potential candidate, including the commercial DSPs [6][17] and embedded processors [9][16] that already exist. With these, it is possible to aggregate multiple low-power VLIW cores into execution-dense arrangements that are not also power-dense. These embedded architectures today focus on low power, and as they execute multiple instructions from the same thread simultaneously, they tend to achieve good power/performance ratios relative to both single-issue cores and high-performance out-of-order cores. By comparison, CPU designs that require hundreds of watts of power, and, with that, large heat sinks and active fans to keep them cool, are going to have a much rougher time integrating into exascale-class systems at reasonable power levels. The power-density issues alone are problematic: when power delivery and heat extraction force socket spacing to be every cubic foot, then each socket has to deliver 10–100 teraflops to reach the desired performance level per rack. While it will certainly be possible to go that route, and most teams are already headed in that direction, it certainly seems like going low-power to begin with will make the way easier.

In particular, with VLIW-CMP, *power will decrease* relative to existing architectures, total aggregate performance (chip-wide IPC) will be maintained, and *single-thread performance will improve* by almost a factor of n. Thus, one should expect an immediate improvement of 2–3× at the node level, just by moving to VLIW-CMP, and this seems to be borne out in actual practice [4]. This is what is needed to reach exascale: re-thinking systems from a power and energy perspective.

This type of efficiency optimization will be required, at all levels, to build power-efficient exascale computers. For instance, if one can achieve a factor of 2–3× improvement in each of the system's building blocks (e.g., application, programming language, operating system, runtime system, microarchitecture, system-on-chip network, memory system, system interconnect, circuit design, process technology), then a system-wide factor of 100× is within reach today.

2 Problems and Solutions

Some VLIW-specific issues remain, including backwards compatibility and the complexities and increased power dissipation brought about by (a) data forwarding across multiple pipelines and (b) multiple read and write ports into the register file.

2.1 Backwards Compatibility

The first obvious issue is backwards compatibility. VLIW has always offered significant promise, but it has had little lasting success in general-purpose computing because the executable images are not typically compatible between generations.

The answer is two-fold: first, the problem has been solved by stop bits, as in TI's C6000 [17] and Intel's IA-64 [8]. These encode the widest available parallelism in the executable, and the chip extracts as much of that as it can. Second, and more importantly, backwards compatibility is a non-issue here: the proposed architecture is not intended for commodity general-purpose machines such as laptops and smartphones; the proposal specifically targets ultra-high-performance machines such as datacenters and supercomputers. These machines bear far more resemblance to *embedded systems* than to commodity general-purpose computers. Like embedded systems, and unlike general-purpose machines, these large-scale systems

- tend to execute the same small set of mission-specific software programs 24/7/365;
- often have teams of developers that spend significant time on code optimization;
- hold power as the limiting criterion (i.e., performance is desirable, as long as it is within the power envelope);
- and often use non-standard, non-commodity components like bespoke operating systems and hardware designs, for better power/performance than commodity systems.

Most supercomputers and datacenters already have software staffs and even non-standard and/or unique hardware. Backwards compatibility is not a must-have feature for them, especially if sacrificing it leads to a significant reduction in power dissipation. A well-known example is IBM's Cell Broadband Engine, in which the ISA was changed, and a significant amount of software was rewritten and optimized for it [7].

2.2 Forwarding and Register-File Ports

Several technical issues still need to be addressed:

- VLIW architectures typically require data-forwarding logic between their multiple pipelines that scales as $O(n^2)$ instead of $O(n)$, and thus potentially dissipates more power than that of n single-issue pipes.
- VLIW architectures typically require a complex register file to support their multiple pipelines; the increased number of read/write ports on the register file leads to more complex circuits and increased power dissipation.

Clustering [1][17] is typically used to address these issues in wide designs (e.g. 8-way or wider), at the cost of penalizing inter-cluster communication. Another solution, *software register-renaming*, addresses both issues, can be used even for 2-way VLIW, exacts no performance penalty, and actually reduces read and write activity, rather than simply reducing the number of ports on the local segment of the register file. The trade-offs are that it encodes details of the pipeline and therefore is not backwards compatible, and it cannot easily pass values across an exception boundary. This latter limitation means that precise interrupts must take forwarded values into account.

2.2.1 Renaming Hardware Registers in Software

As the literature shows [5][13][15], register lifetimes are typically short, and the number of active registers is typically low; i.e., most of a processor's output is temporary and not intended to last long in the register file. This should not be surprising, as it is why hardware register renaming works so well, and is

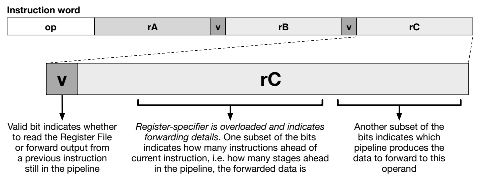
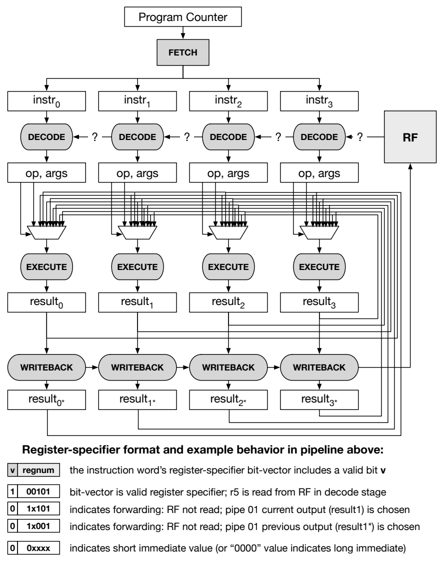

Instruction word
op
rA
v
rB
v
rC
v
rC
Valid bit indicates whether to read the Register File or forward output from a previous instruction still in the pipeline
Register-specifier is overloaded and indicates forwarding details. One subset of the bits indicates how many instructions ahead of current instruction, i.e. how many stages ahead in the pipeline, the forwarded data is
Another subset of the bits indicates which pipeline produces the data to forward to this operand

Fig. 2. Register-specifier bit values in software renaming



Register-specifier format and example behavior in pipeline above:

v	regnum	the instruction word's register-specifier bit-vector includes a valid bit **v**
1	00101	bit-vector is valid register specifier; r5 is read from RF in decode stage
0	1x101	indicates forwarding: RF not read; pipe 01 current output (result1) is chosen
0	1x001	indicates forwarding: RF not read; pipe 01 previous output (result1*) is chosen
0	0xxxx	indicates short immediate value (or "0000" value indicates long immediate)

Fig. 3. Pipeline organization showing forwarding MUXes

also one of the touted strengths of Tomasulo's Algorithm—for instance, that long strings of back-to-back writes to the same physical register will ultimately be ignored by the register file, and only the last write causes a physical update [18]. Software register-renaming acts in a similar manner.

In a VLIW pipeline, the number of forwarding paths is mn: the stages m between *execute* and *writeback* times the width n of the machine. Rather than have hardware perform dependency-checking across pipelines—i.e., compare every register specifier against every other, an expensive $O(mn^2)$ priority-encode operation—software register-renaming explicitly encodes the forwarding path under control of the compiler. This produces exactly the same performance; it is simply under the control of the assembler/compiler, not the hardware. A valid bit in the instruction word indicates that the associated register specifier indicates not a register in the register file but the output of another instruction still in the pipeline ahead. Thus, the per-operand hardware reduces to a single mn+1-wide multiplexer or a series of n+1-wide multiplexers, in which each *select* signal comes directly from the instruction word.

This is illustrated in Fig. 2. A valid bit associated with an operand identifies whether the operand should be read from the register file or not; in the case of a '1' valid bit, the register file is read. A '0' valid bit indicates that the register-file read can be gated off, thereby saving power, and either the operand field contains a short immediate value, or the register specifier field indicates which pipeline is producing the result and how many cycles ahead from the current instruction it is.

Fig. 3 shows this in use. A simple 4-way VLIW pipeline includes a set of $2n$-wide multiplexers, two for each pipe (only one shown, for simplicity), each controlled by software. In this example, three bits of the register specifier identify a forwarding path: one bit indicates whether the source of the data is one or two stages ahead in the pipeline, and two bits indicate the pipeline producing the result. Instructions depending on the result of the instruction immediately before them activate the forwarding path from *writeback* to *execute*. To accommodate instructions that depend results two stages ahead, an extra register is placed at the end of the pipeline. Instructions depending on the result of the instruction two stages before them activate the forwarding path from this extra register to the *execute* stage. Longer pipelines could simply use longer FIFO structures and wider forwarding MUXes. As the figure shows, the valid bit of the register specifier is known in the decode stage; therefore, if it indicates forwarding (or immediate value), the register-file read is deactivated for the corresponding operand during that cycle, avoiding unnecessary register-file read energy.

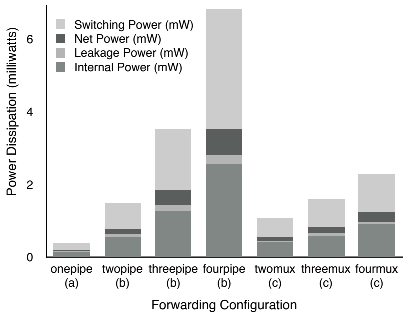


Fig. 4. Power of forwarding logic in (a) a single-issue pipe, (b) 2- 3- and 4-way VLIW pipes using traditional n×n forwarding, and (c) 2- 3- and 4-way VLIW pipes using software register-renaming

One obvious limitation is that forwarding data across branch boundaries can be tricky, as the distance is dependent on the branch penalty. The problem becomes more difficult when dealing with precise exceptions: if a producer-consumer pair is separated by an interrupt, the data is lost. A similar problem was solved by forcing the temporary data into the register file when an exception was raised [14], but that technique is only possible if each producer has a register-file target in mind; in our implementation, most producer instructions write to the $r0$ register, thereby throwing their results away. Our implementation mirrors typical software practices in the design of drivers, firmware, and other low-level software: we use markers to designate critical sections and hold off the handling of an interrupt until after the critical section is exited. If an exception occurs that would lead to killing the process (a software error as opposed to an external interrupt), then the critical section is ignored, and the exception is handled.

2.2.2 *Power Dissipation of Software Register-Renaming*

Fig. 4 compares the forwarding-logic power of a regular single-issue pipeline, three different instances of traditional VLIW forwarding schemes, and software register-renaming. Power is estimated from RTL by using Cadence's Encounter RTL Compiler and Synopsys's SAED 90nm Library. As the figure shows, traditional forwarding within VLIW pipelines scales quadratically: the power for forwarding logic in 2-way, 3-way, and 4-way VLIW are 4.1×, 9.3×, and 18.1× the power of a single in-order pipe. The 2-way, 3-way, and 4-way software register-renaming implementations scale linearly (from a higher base, due to wider MUXes) and exhibit power that is 2.9×, 4.2×, and 6.0× the power of the single in-order pipe.

Note that these results represent *only* the power dissipated in the forwarding logic; when the register-file read power is included, the savings are higher, as software register-renaming reduces the number of register-file reads, compared to a normal in-order pipeline. Thus, software register-renaming reduces the amount of register-file read energy the more it is used.

In addition to read energy, software register-renaming can also reduce the register-file write energy in two different ways. The first effect is similar to Tomasulo: when temporary values are produced and consumed within a short number of cycles, they never need to be written to the register file, and software register-renaming enables the pipeline to consume these temporary values immediately without ever writing them to the register file; they are simply produced and consumed within the pipeline and are never written out.

The second effect is that one can use fewer write ports on the VLIW register file. The more an application produces temporary results, the less important the register-file's write port becomes. For instance, a load-word instruction using r1+r2 as a load address could be implemented as follows:

```
add    r0:tmp0, r1, r2
lw     r29, tmp0
```

The first instruction adds the contents of r1 and r2 and "writes" the result to r0, which is defined by the architecture to be 0 and is therefore unwritable. Thus, this is the equivalent of performing a "cat > /dev/null" operation.

Through the ":tmp0" declaration, the instruction's output is tagged as *tmp0*, a label used by the assembler to forward values within the pipeline: for instance, the *tmp0* tag is referenced in the following instruction, indicating that the *lw* instruction's address operand will not be read from the register file but will instead be forwarded within the pipeline. At run time, the sum produced by the *add* instruction is forwarded by the control logic directly from the *add* instruction to the *lw* instruction, and the result never gets written to the register file because the *add's* write target is r0. This produces the exact same CPI as traditional data forwarding; it simply uses less power.

In addition to reducing register-file write energy, this can reduce the need for an entire register file *port*, and yet maintain the architecture's existing instruction-assignment symmetry (i.e., one can eliminate one or more write ports without having to change which opcodes can be assigned to which pipelines). One or more of the n pipelines can be dedicated to instructions that write no results, even ALU instructions such as the *add* instruction above. For instance, one could have an example pipeline arrangement such as the following:

- pipe0: all ALU types & memory types*
- pipe1: all ALU types & memory types*
- pipe2: all ALU types & memory types* (no RF writes)
- pipe3: all ALU types (no RF writes), branches & jumps

(* *maximum two total memory ops across the four pipes*)

This arrangement allows instructions in pipelines #0 and #1 to write the register file freely, while the other two pipelines (#2 and #3) can *execute* instructions but cannot *writeback* any results. Instructions in pipelines #2 and #3 may include memory-store instructions (pipe #2), or branches and jumps (pipe #3), or any ALU and memory-load instructions that produce temporary outputs—i.e., results consumed within the pipeline and never intended to be written to the register file.

Compared to a fully symmetric arrangement, in which all pipes have write ports to the register file, this organization reduces wiring, die area, and power dissipation; it also reduces the number of bits in the VLIW instruction word. Note these benefits are enabled by software register-renaming: traditional architectures have no simple way to indicate "do not write this result to the register file" without also causing problems in the forwarding paths (e.g., causing the value of zero to be forwarded instead of the output produced).

3 Conclusions

To achieve exascale-level computing rates, and simultaneously stay within single-digit megawatt power levels, we will need to reach CPU-level capabilities approaching 1 TFLOPS per Watt. We have argued that a VLIW-CMP chip architecture, irrespective of core instruction-set, is the right node-level architecture choice to achieve this target, as it reduces power dissipation without sacrificing single-thread performance. Even in hybrid proposals that combine latency-optimized cores with throughput-optimized cores, VLIW-CMP is competitive, as it approaches the performance of the former, while offering a better power footprint, and it matches the power footprint of the latter, while offering better per-thread performance.

Additionally, it should be clear that this kind of approach to power/performance optimization will be important all throughout the system hierarchy, from circuits to microarchitectures, to chip organizations, to networks on chip, to node architectures, to board interconnects, to rack architectures, to rack interconnects, and so on. Without such efficiency optimization, exascale will remain continually in the future.

References

[1] A. Aleta, J. M. Codina, A. Gonzalez, and D. Kaeli, "Heterogeneous clustered VLIW microarchitectures," in *Proc. CGO*, 2007, pp. 354–366.

[2] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, G. Venkatesh, and J. Xu, "Runnemede: An architecture for ubiquitous high-performance computing," in *Proc. HPCA*, Feb. 2013, pp. 198–20.

[3] R. P. Colwell, R. P. Nix, J. J. O'donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *Proc. ASPLOS*, 1987, pp. 180–192.

[4] E. Francesquini, M. Castro, P. H. Penna, F. Dupros, H. C. Freitas, P. O. Navaux, and J.-F. Mehaut, "On the energy efficiency and performance of irregular application executions on multicore, NUMA, and manycore platforms," *J. Par. & Dist. Comp.*, vol. 76, pp. 32–48, 2015.

[5] M. Franklin and G. S. Sohi, "Register traffic analysis for streamlining inter-operation communication," in *Proc. MICRO*, 1992.

[6] J. Fridman and Z. Greenfield, "The TigerSHARC DSP architecture," *IEEE Micro*, vol. 20, no. 1, pp. 66–76, 2000.

[7] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in Cell's multicore architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[8] L. Gwennap, "Intel, HP make EPIC disclosure," *Microprocessor Report*, vol. 11, no. 14, pp. 1–9, Oct. 1997.

[9] ——, "Xtensa 10 plays well with ARM," *Microprocessor Report*, Oct. 2013.

[10] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A single-chip multiprocessor," *IEEE Computer*, vol. 30, no. 9, pp. 79–85, 1997.

[11] P. Lotfi-kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out processors," in *Proc. ISCA*, Jun. 2012, pp. 500–511.

[12] H. McGhan, "Niagara 2 opens the floodgates," *Microprocessor Report*, Nov. 2006.

[13] P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in *Proc. DSN*, Jun. 2007.

[14] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalont, "Exploiting data forwarding to reduce the power budget of VLIW embedded processors," in *Proc. DATE*, 2001.

[15] R. Sangireddy and A. K. Somani, "Exploiting quiescent states in register lifetime," in *Proc. ICCD*, Oct. 2004, pp. 368–374.

[16] A. Suga and K. Matsunami, "Introducing the FR500 embedded microprocessor," *IEEE Micro*, vol. 20, no. 4, pp. 21–27, Jul. 2000.

[17] Texas Instruments, *TMS320C62x DSP CPU and Instruction Set Reference Guide*, May 2010.

[18] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM J. R & D*, vol. 11, no. 1, pp. 25–33, Jan. 1967.