Non-Uniform Access Asynchronous Register Files

David Fang, Rajit Manohar
Computer Systems Laboratory
Electrical and Computer Engineering
Cornell University
{fang,rajit}@csl.cornell.edu

Abstract

Register files of microprocessors have often been cited as performance bottlenecks and significant consumers of energy. The robust and modular nature of quasi-delay insensitive (QDI) design offers a toolchest of techniques for improving average-case performance and reducing energy consumption of register files, which cannot be leveraged as easily in synchronous designs. In this paper, we focus on the design of an asynchronous register core, the heart of a register file. We describe the vertical pipelining transformation and describe the locking mechanism that maintains pipelined mutual exclusion among reads and writes to the same register. The primary contributions of this paper are 1) detailed evaluation of the width-adaptive datapath (WAD) representation in register files, which leads to significant energy reduction by conditionally communicating higher significant bits of integers with little performance degradation, and 2) 'nesting' the register core to create non-uniform banks to facilitate faster and lower energy accesses to more frequently used registers and slower accesses to less frequently used registers without *increasing the interconnect requirement or control complexity. We present* `spice`*-simulated results for a wide variety of register files laid out in TSMC* $.18\mu m$ *technology.*

1. Introduction

In modern microprocessors, register files sit at the smallest and fastest end of the memory hierarchy. Register files are exposed as part of an instruction set architecture (ISA) to the compiler, whose task is to schedule uses of registers as efficiently as possible. Since the register file is used by almost every instruction, it is important that it be accessible on every cycle. However, several trends compound the challenge of meeting cycle times in superscalar register files:

- increasing number of ports to support wider instruction issue ⇒ increased area of each register word
- deeper pipelines ⇒ more in-flight instructions ⇒ more physical registers
- increasing architectural width
  (today 32 and 64 bits, tomorrow?)

The first two points arise from exploiting increasing instruction-level parallelism (ILP) in the instruction stream. The resulting increase of bit line loads slows register accesses while increasing access energy. The slowdown of register files (as registers grow in size, number of ports, and architectural width) has been well-modeled [23, 29]. To meet cycle time requirements, two countermeasures against these trends are:

- banked, distributed, or clustered register files
- multi-cycle access register files

Splitting register files often results in increased interconnect complexity and sometimes explicit communication through inter-cluster traffic [12, 23]. Multi-cycle register files lead to increased pipeline depth (and branch misprediction penalty), and *dramatically* increase the bypass and control complexity [1, 6].

In an asynchronous datapath, variations in access time need not complicate the pipeline organization, due to the data-driven nature of operation. This paper presents implementations of non-uniform access registers that are free from the re-timing complexities of synchronous datapaths.

On top of the challenge of sustaining performance in the face of growing complexity is the problem of increasing energy consumption in register files. Today's register files consume up to 40% of the datapath energy, or 15% of the core energy [27]. Modifying them for increasing ILP only increases the register file's proportion of energy consumption. Traditional techniques for reducing register file energy fall into one of several categories: 1) circuit techniques that reduce the energy per register access, 2) reducing the

number of accesses to register cores, 3) more efficient utilization of equal or fewer resources (number of registers, ports) [21, 27, 29]. In this paper, we exploit the compressibility of operand values and non-uniformity in register usage to achieve an average-case improvement.

With self-timed designs, timing constraints are lifted and replaced with a discipline of local handshakes on channels between concurrent processes. The question we address in this paper is: How can we leverage the benefits of asynchronous design — modularity, formal verification, and robustness of quasi-delay insensitivity (QDI) — to improve the performance and reduce the energy of growing register files?

The paper is organized as follows: In Section 2, we describe how the register core is vertically pipelined to improve throughput and how pipeline-locking is implemented to preserve correctness. In Section 3, we present the implementation of a width-adaptive datapath (WAD) register file, which saves considerable energy by communicating compressed integers. In Section 4, we present *nesting* as an implementation of a non-uniform access register core and discuss its potential for improving performance and reducing energy in the average case. In Section 5, we describe our simulation methodology and compare performance and energy results for the various designs of the core read and write ports.

2. Vertically Pipelining the Register Core

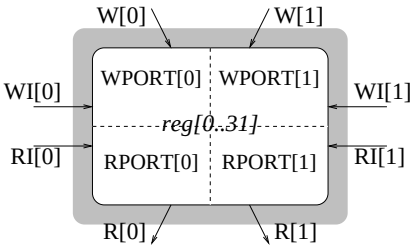

Figure 1. Register core decomposition

The entire register file, which includes the surrounding control and bypass, is decomposed in the same manner as in the MiniMIPS [9, 17]. Our model for the core is a 2-read, 2-write ported, 32-word x 32-bit register array, shown in Figure 1, which receives indices from the control on channels $RI\,[0..1]$ and $WI\,[0..1]$ that correspond to the register number, supplies data to the bypass (not shown) on the read bit line channels $R\,[0..1]$, and receives data from the bypass on write bit line channels $W\,[0..1]$. The received index in each port process is decoded into a 1of32 word line channel that selects which register is accessed. The values of the respective registers are represented by the reg variable

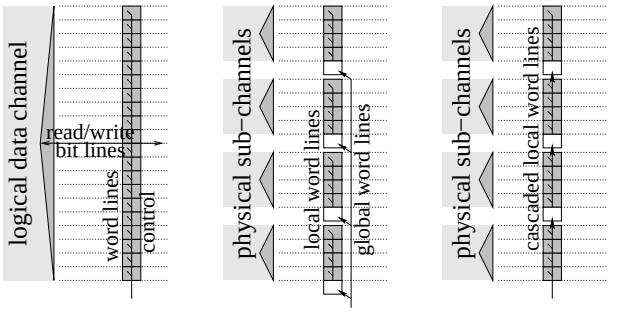

Figure 2. Different pipeline organizations of register cores. The triangles represent completion trees in a QDI design.

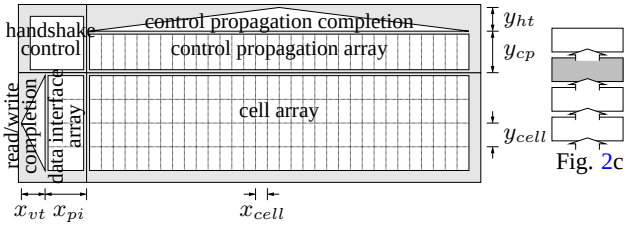

Figure 3. Template floorplan for a single block of a vertically pipelined register core from Figure 2c, shown here with a 4-bit x 32-word block size. Component dimensions are listed in Table 4.

array, which is shared among all port processes. The control for the core *guarantees* that on each iteration: 1) the indices it issues never concurrently read and write the same register, and 2) that each register is written by at most one write port. Since the port processes are decoupled from one another, reads and writes to *different* registers may safely complete in any order.

If one were to synthesize quasi delay-insensitive (QDI) production rules from the current specification of the core, one would find that the handshake cycle times would be severely limited by the completion trees (across the data channels), one of the byproducts of the QDI asynchronous design style. Figure 2a shows a schematic of an unpipelined register core, with completion trees that span the architectural width of the datapath. We discuss two techniques that reduce the size of completion trees.

2.1. Pipelined Completion

Pipelined completion, illustrated in Figure 2b, distributes the control for full-width operations across several blocks by wiring global control to each block, and in each block,

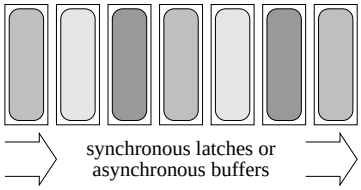

Figure 4. Aligned pipeline operation

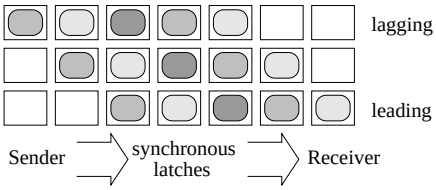

Figure 5. Synchronous parallel skewed vertical pipeline operation

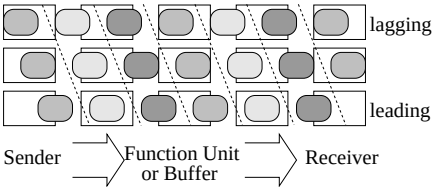

Figure 6. Asynchronous block-skewed vertical pipeline operation

copying control to local register cells. This moves partial completion off critical cycles and results in smaller, block-wide completion trees. In the MiniMIPS register file, each block was one byte wide [17]. The control acknowledge for each block is collected in a separate completion tree at the global distribution stage. A consequence of pipelined completion is that each register's port control requires both a global word-line *and* a local word-line. Doubling the control-interconnect bandwidth requires either another metal layer or roughly twice the word pitch, which is inefficient for word-arrayed structures such as the register core.

2.2. Vertical Pipelining

We chose to *vertically pipeline* the register core, as shown in Figure 2c.[1] In a QDI vertical pipeline, control is linearly pipelined from the least significant to most significant blocks via QDI handshakes on local (1of32) word-line channels, which *eliminates* the need for long-distance global word lines across the register core. Vertical pipelining can be thought of as an instance of pipelined completion, where control is distributed in a linear tree, as opposed to a balanced tree.

The basic floorplan for one block of the pipelined register core appears in Figure 3. The cell array stores bits in 2-read, 2-write ported register cells. The control propagation array contains precharge circuits (similar to Figure 7) that copy word lines for each port to the successor block and contains an OR-completion tree across each port's word line channel. The peripheral data interface circuits, shown in Figure 8, invert the read-data bit lines into channel R, drive the read-enable bit lines, control the resetting of the read-data bit lines and the write-validity bit lines. The $\_rv$ and $\_wv$ validity signals are completed in each block of the R and W channels. The handshake control collects control and data completion signals and communicates with the neighboring blocks using QDI validities and acknowledges.

[1] We call this *vertical* because most traditional pipeline diagrams show pipeline stages flowing horizontally. A design with horizontal and vertical pipelining is also called *two-dimensionally pipelined*.

Pipelined completion and vertical pipelining benefit from smaller completion trees and distributed control fanout. Another advantage of vertical pipelining over pipelined completion is that the control for each block only needs to collect the acknowledgment from its immediate successor, as opposed to one stage completing across all blocks. This means that the control for an unpipelined design works for an arbitrary number of vertical pipeline stages! Vertically pipelined architectures scale to any width with *constant cycle time* on data handshakes.

A natural property of vertical pipelining is that communicated data tokens on the datapath are *block-skewed*, i.e., as control ripples vertically across the blocks, less-significant blocks are communicated earlier than more-significant blocks, as illustrated in Figure 6. An example of extreme vertical pipelining with a 1-bit granularity is the *bit-skewed* pipeline of the asynchronous lattice filter [7]. The synchronous counterpart to vertically pipelining ("byte-parallel, skewed", Figure 5) has been proposed [4], however, the disadvantage in the synchronous domain is that each successive block is an entire clock-cycle behind its predecessor, which is much worse than the 2-gate vertical latency through each pipeline stage's control propagator, whose simplified circuit template is shown in Figure 7.

In a block-skewed vertical pipeline, since the most significant blocks' results always lag behind the least significant blocks', operations that depend on the more significant blocks (e.g. compare instructions, right-shifts) will execute slower than with block-aligned pipelined completion. The overall performance such architectures Is determined by a tradeoff between cycle time and the *total vertical latency*, which depends on the number of blocks.

2.3. Pipeline Locking

Even if the control issues read-write exclusive port indices on each iteration, vertically pipelining accesses to the shared variables reg without further measures can lead to a violation of read-write exclusion because the full-width completion of shared variable accesses is decoupled from the control handshake. Since we make no assumptions about when read and write operations complete, we need to

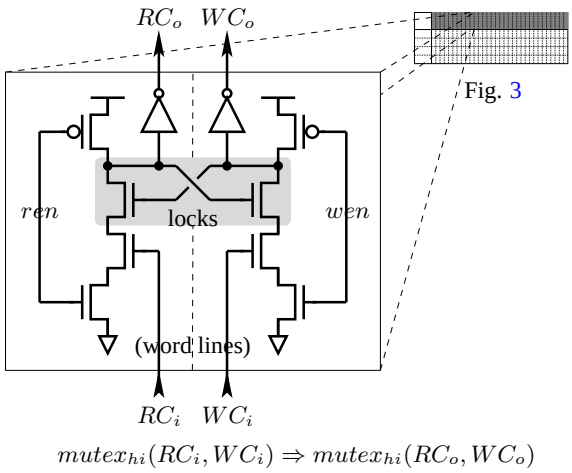

Figure 7. A 1-read, 1-write pipeline-locking template circuit for a control propagation element in each block of a read-write vertical control pipeline. ren and wen are local precharge signals for each port within a block. The shaded portion of the circuit shows the locks.

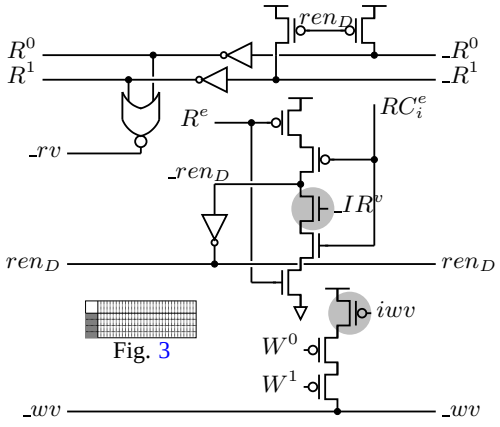

Figure 8. Bit line peripheral circuit for each read and write port. The shaded transistors are present for only the nested versions, described in Section 4.

prevent data hazards, subsequent reads from racing ahead of outstanding writes, and vice versa.

Previous asynchronous microprocessor designs have employed different techniques for dealing with read-write hazards in register files [10, 11, 20, 22, 26], however, we implement *pipeline-locking* in the same way as in the MiniMIPS [17]. Pipeline-locking propagates the guarantee of mutual exclusion and atomicity through each pipeline stage when the critical action is decoupled from control and thus, prevents violation of dependencies in the pipelined register core [16]. Figure 7 shows the precharge template for a 1-read, 1-write, pipeline-locked control propagator, which can be easily generalized for more ports. Preserving mutual exclusion requires locking for both pipelined completion and vertical pipelining because completion of the port operations is decoupled from the control. All vertically pipelined read and write ports in this paper were designed with a 4-bit block size.

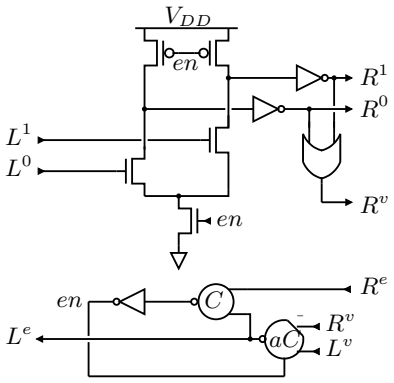

Figure 9. 1of2 precharge enable-valid full-buffer

2.4. Base Design Production Rules

QDI production rules for the register core read and write ports are synthesized based on a few template handshaking expansions [14]. To minimize the cost of wiring signals across large arrays, we restricted ourselves to reshufflings that use a single precharge signal and shared output validities [19]. The handshaking expansion (HSE) for our variation, called *precharge enable-valid full-buffer* (PCEVFB), is listed as HSE Program 1, and its 1of2 circuit template is shown in Figure 9. In this paper, we present results for only the PCEVFB versions of the register core; the half-buffer counterpart (PCEVHB) is slightly slower in most cases [9].

Program 1 HSE: precharge enable-valid full-buffer (PCEVFB)

$$*[[R^e]; en\uparrow; [L]; R\uparrow; L^e\downarrow; \\ [\neg R^e]; en\downarrow; R\downarrow, ([\neg L]; L^e\uparrow)]$$

In addition to the single-banked core of 32 registers, we also designed and simulated register cores with two banks of 16 registers. Not only does banking accelerate and reduce the energy of asynchronous memory-structure accesses, but it can also yield better average-case performance when sequential accesses touch different banks [8, 13]. To accommodate for two banks in the register file, the control processes perform one level of decoding before each bank's decode to distinguish between accesses to the two banks, the writeback bypass is extended to split a conditionally copied value to one of the banks, and the read bypasses are extended to merge values from one more bank source.

3. Width Adaptivity (WAD)

The observation that the vast majority of values communicated on the integer datapath require fewer bits than the full architectural width motivates the encoding of integers in a compressible representation to reduce switching activity and energy. Various surveys of 32-bit architectures have found that 30 to 80% (averaging over 65%) of integer datapath energy can be reduced by suppressing communication (or clocking) of leading 0s and 1s [4, 15]. 64-bit and wider architectures exhibit even greater compressibility.

Synchronous designs have exploited the compressibility of integers using opcode- and operand-based clock-gating to reducing switching and latching activity on the datapath [2, 3], however, these proposed architectures have only a very coarse granularity for detecting narrow operands. Synchronous byte-parallel, skewed implementations suffer from latching overhead (of synchronous vertical pipelining) and increased complexity in the control for bypass paths [4].

Table 1. The encoding of width-adaptive datapath (WAD) blocks

delim. bit	MSB	next block	control
0	0	normal	propagate
0	1	normal	propagate
1	0	$\overline{0}$	terminate
1	1	$\overline{1}$	terminate

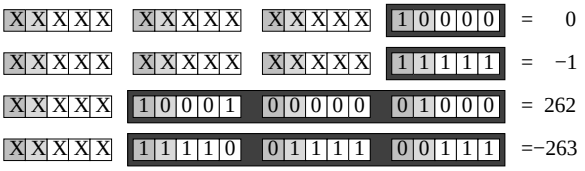

Figure 10. Examples of WAD numbers. MSBs are lightly shaded and delimiter bits are darkly shaded. Only darkly bordered blocks' bits are communicated.

Our work implements the block-skewed, *width-adaptive datapath* (WAD) architecture [15]. The WAD encoding is shown in Table 1, with examples given in Figure 10. Where the delimiter bit is true (in the terminal block), all remaining higher significant bits are understood (without their communication) to be leading 0s or 1s, depending on the MSB of the terminal block. Asynchronous WAD has several advantages over other similar proposed architectures: 1) Rather than constantly re-detecting leading 0s and 1s, delimiter bits are stored and communicated with significant blocks of data and re-evaluated for compaction with low overhead [15]. 2) The distributed nature of WAD is entirely transparent to the datapath control — *no* modification to existing control logic outside of the vertical pipeline is required. 3) The robust, self-timed nature guarantees that any timing variation that arises from WAD is correctly tolerated. One could conceivably vertically pipeline non-uniformly to optimize the placement of delimiter boundaries for energy minimization. Formally, WAD can be described with a template process transformation that can be applied to all pipelined functional units, including arithmetic blocks such as adders [9, 15, 18].

The problem with implementing WAD using pipelined completion is that control is *unconditionally* copied to all blocks with global word lines, which may lead to incorrect behavior. To adapt pipelined completion for data-dependent control requires feedback from every block to the global word-line copy stage, which lengthens the handshake cycle time. The linearly pipelined nature of vertical pipelining is much better-suited for WAD.

WAD introduces very little change to the block-skewed, vertically pipelined core read and write port circuits. Aside from adding one row of register cells and bit lines per block for the delimiter bit, WAD modifications are confined to the handshake control and control propagation cells in Figure 3. Read port control propagation depends on the state of the selected register's delimiter bit, which adds one series transistor to each precharge stack in Figure 7. Read port control termination is detected with a shared bit line, similar to read bit lines. In the read port handshake control, the reset phase only waits for the control output acknowledge in the propagation case, otherwise the wait is bypassed by the termination case. Write port control propagation depends on the value of the delimiter bit of the arriving data (also stored in the delimiter bit cell), which adds one series transistor to the original precharge stack.[2] The termination case is detected in the handshake logic. In the write port handshake logic, the reset phase only waits for the control output acknowledge in the propagation case, otherwise the wait is bypassed by the termination case.

In Section 5.2, we show that WAD leads to significant energy savings over the standard (non-WAD) pipeline block while suffering only slightly in cycle time.

4. Non-Uniform Nesting

Banking register files to uniformly reduce their access times and cycle times runs into limitations because of the increased interconnect requirement. How can one accelerate registers without further banking? Since register allo-

[2] We have studied two variations of reshufflings of the WAD write port, which are differentiated by whether the precharge enable signal is *conditionally* or *unconditionally* raised on each iteration [9]. In this paper, we present results for only the unconditional variation.

Table 2. MIPS register conventions. Bolded register classes are the most frequently used.

name	reg#	convention
$zero	0	**constant** 0
$at	1	reserved for compiler
$v0-$v1	2–3	**results**
$a0-$a3	4–7	**arguments**
$t0-$t7	8–15	(callee-saved) temps
$s0-$s7	16–23	**caller-saved**
$t8-$t9	24–25	(callee-saved) temps
$k0-$k1	26–27	reserved for OS
$gp	28	global pointer
$sp	29	**stack pointer**
$fp	30	**frame pointer**
$ra	31	**return address**

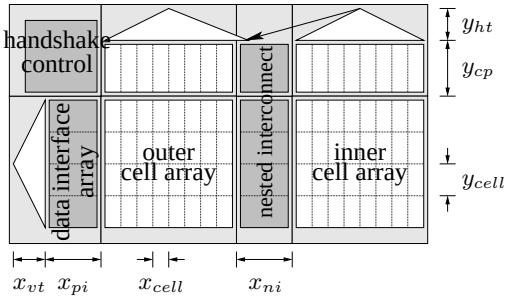

Figure 11. Floorplan of a nested 4-bit x 16-word pipeline block of the register core, with the outer partition on the left side and the inner partition on the right. New or modified components that arise from nesting are darkly shaded, while all other components corresponding to Figure 3 remain unchanged. The WAD, nested floorplan includes one more row of delimiter bit cells in the cell array. The dimensions for the various components are listed in Table 4.

cators in compilers obey register conventions, such as that of the MIPS ISA, listed in Table 2, the majority of register accesses is often covered by only a minority of registers: results, arguments, caller saves, stack and frame and return pointers. Table 3 shows the most frequently read and written MIPS registers, sorted by frequency.[3] Since asynchronous designs are not constrained to any timing requirements, there is potential to achieve average case speed-up and energy reduction for sufficiently skewed register usage distributions. We introduce asynchronous register *nesting* as a method for exploiting skewed register usage with non-uniform register accesses.

[3] Averaged across SPECInt95 benchmarks with training inputs: 099.go, 129.compress, 134.perl, 124.m88ksim, 130.li, 147.vortex, 126.gcc, 132.ijpeg, compiled with `gcc-2.95.3 -O3`, run on a MIPS simulator

Table 3. Cumulative dynamic usage frequencies of the 20 most read and written MIPS registers

N	reg	read%	cumul.	reg	write%	cumul.
1	0	32.95	32.95	0	27.63	27.63
2	3	14.90	47.85	2	18.96	46.59
3	2	12.59	60.44	3	18.35	64.94
4	30	9.78	70.22	4	9.57	74.51
5	5	8.03	78.25	5	6.83	81.34
6	4	6.36	84.61	6	4.87	86.21
7	29	4.73	89.34	31	2.95	89.16
8	16	2.36	91.70	29	2.52	91.68
9	31	1.73	93.43	16	2.04	93.72
10	17	1.43	94.86	30	1.67	95.39
11	6	1.19	96.05	14	1.49	96.88
12	28	0.91	96.96	1	1.03	97.91
13	14	0.85	97.81	17	0.52	98.43
14	1	0.60	98.41	18	0.41	98.84
15	18	0.48	98.89	7	0.30	99.14
16	7	0.29	**99.18**	19	0.22	**99.36**
17	19	0.23	99.41	8	0.13	99.49
18	20	0.15	99.56	20	0.13	99.62
19	21	0.13	99.69	9	0.08	99.70
20	8	0.09	99.78	21	0.08	99.78

Non-uniform register architectures have a long evolutionary history in clocked-designs. One of the early appearances of heterogenous register files was in the Cray-1 supercomputer [24], in which primary registers were used directly in computation, while intermediate registers were used for auxiliary storage. Movements between the two levels of registers were accomplished by explicit instructions. A key observation in superscalar architectures is that code with high ILP suffers very little from slower multi-cycle register accesses, while serially dependent code (with low ILP) is not accelerated by increasing the number of physical registers. Such contrasting characteristics of instruction streams demand small, fast register files backed by larger, slower register files or memory [5, 23, 25]. Proper scheduling of dependent instructions through fast registers leads to speed up of critical code, while larger register files are good for reducing spills that would otherwise pollute the cache.

Accommodating non-uniform accesses has proven to be a challenge in clocked designs. Support for variable multi-cycle accesses requires multiple levels of bypassing, which results in increased bypass latency, and pipeline depth (and hence, branch misprediction penalty) [1]. The interconnect requirement is the greatest limitation to fully-connected multi-level bypassing, and partial bypassing pushes complexity into the control logic [6]. Asynchronous register nesting suffers from none of these problems.

With nesting, we partition a single bank of registers into a fast outer bank and a slow inner bank using a QDI interconnect, *without* requiring any additional external interconnect. By isolating the load of the inner bank, the outer

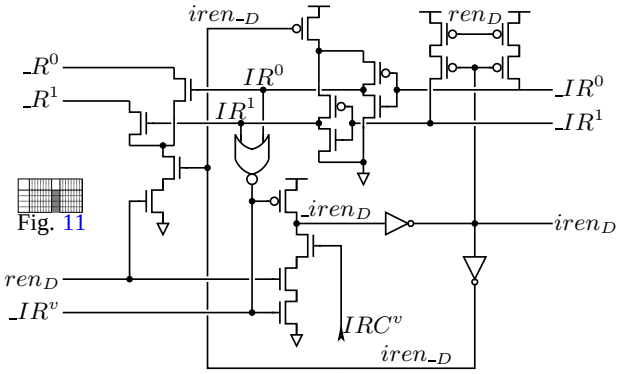

Figure 12. Read port nested interconnect

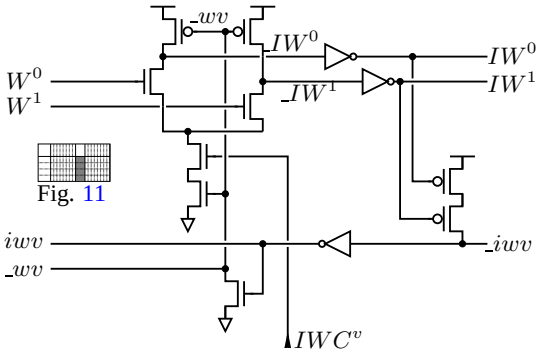

Figure 13. Write port nested interconnect

bank's bit line capacitance is roughly halved relative to the unpartitioned bank and therefore operates faster and with less energy. A similar technique uses only a pass gate to partition each bit line, which is most useful in clocked designs where the resulting slow-case bit line delay still meets the timing constraints [27]. Though a QDI implementation using pass gates may be possible, we have only explored one implementation using fully-restoring logic.

Our implementation of register nesting follows a few guidelines. 1) We maintain a strictly QDI design style, free of timing assumptions about gate delays. A consequence of this choice is that we have greater circuit overhead in the interconnect than other possible designs that use timing assumptions. 2) We want to introduce as few modifications to the existing non-nested design as possible to minimize the slowdown to the outer partition relative to a non-nested bank of the same size. Since accesses to the inner partition use the outer partition through the interconnect, this can be achieved by making inner accesses behave like an access to a normal (but slower) register from the outer partition's perspective. The interconnect functions like a bypass environment from the inner partition's perspective.

The floorplan for a nested block of the vertically pipelined register core is shown in Figure 11 with eight registers in the outer and inner partitions. The circuits in the register cell array and control propagation array remain unchanged from Figure 3. The arrow in the figure shows how the control completion trees have been unbalanced to favor faster completion in the outer partition than in the inner partition. To summarize the impact of nesting on the original register core design: In the peripheral circuits interfacing the R and W channels, one series transistor (shaded in Figure 8) has been added to one production rule per port per bit, to implement a guard signaling that the inner partition has reset before the read and write communication handshakes are allowed to proceed. Each of those transistor gates is connected to a wire run across the outer register array to the nested interconnect, which is the greatest cost of nesting. Checking the reset of each nested bit line independently eliminates the need for bit line completion trees in the inner partition. For the handshake control circuits, nesting introduces a single transistor modification in *only* the WAD read port because its control termination condition is detected on a shared bit line which is now split into two partitions [9]; all standard and write port handshake controls work *unmodified*!

New circuits are introduced in the nested interconnect component, shown in Figure 12 for each read port bit, and Figure 13 for each write port bit. The nested interconnect circuits for the control propagation arrays (not shown) are relatively simple for the standard and WAD versions. $\_IR$ and IW are the inner partition's counterparts of the read and write bit lines. IRC^v and IWC^v are the word lines that activate the nested interconnect. $\_IR^v$ and iwv, the validities for reading and writing the inner partition, are connected across the outer partition to the peripheral data interface circuits.

The conservativeness of QDI, along with the minimal (one-sided) guard addition to the outer partition's peripheral data interface, imposes a high penalty for reading from the inner partition [9]. Since only $\_IR^v\uparrow$ is checked by the outer partition's read data interface (Figure 8), we must guarantee that $\_IR^v$ be stably low *before* the outer partition's bit lines $\_R\downarrow$ may fire. This is accomplished by waiting for the inner partition to complete most of its reset phase before responding to the outer partition with data. To alter the interface between the partitions and allow greater partition concurrency (while maintaining QDI) would require more information about the state of the inner partition to be communicated to the outer partition, which would incur greater wiring overhead over the outer partition and transistor overhead in the peripheral data interface. The tradeoff for such a transformation reduces the inner partition access penalty while reducing the speedup of the outer partition.

Writing to the inner partition is not quite as restricted. The outer partition's write data interface only checks $iwv\downarrow$,

which means iwv must be stably high before the outer partition is allowed to see the write validity $\_wv\downarrow$. Since this is already guaranteed, the reset phase of the inner partition and the outer write-validity may proceed concurrently as soon as the write to the inner partition is complete.

5. Results

Table 4. Component sizes, corresponding to Figure 11.

dim.	λ	λ/x_{cell}	dim.	λ	λ/y_{cell}
x_{cell}	65	1.00	y_{cell}	210	1.00
x_{pi}	268	4.12	$y_{cp_{std}}$	380	1.81
x_{vt}	109	1.68	$y_{cp_{WAD}}$	401	1.91
x_{ni}	240	3.69	y_{ht}	140	0.67

Each version of the register core described in this paper was laid out in TSMC $.18\mu m$ technology using SCMOS design rules and simulated using a variant of `spice`.[4] The sizes and areas of all components are summarized in Table 4. Though our layout was not thoroughly optimized for any particular metric, transistor gates were reasonably sized. We measured performance and energy over a 25 ns period of full-throughput operation on each port. For reading in the width-adaptive case, we maintained all register delimiter bits as 0 to simulate only the propagation case, because the termination case is never throughput-limiting. For writing, we alternated bit values on every iteration, except for the WAD delimiter bits (0), to simulate worst-case write-validity delay and energy in writing activity.

Performance and energy results for all core read and write ports are listed in Table 5. We also report the *read latency* for each read port and the *write latency* for each write port. The read latency is measured as the delay from the read word line to the read bit line $R\uparrow$ (2 transitions), and the write latency is measured as the delay from the write bit line and word line input to write validity $\_wv\downarrow$ (3 transitions) and always includes the time to toggle the internal storage bits. Assuming that the delay of the inverters that drive $R\uparrow$ is invariant, the absolute differences in read latencies are accounted for by the fall time of the read bit line $\_R\downarrow$. Assuming that the delay of toggling the storage bits is invariant, the absolute differences in write latencies are accounted for by the fall time of the write validity line $\_wv\downarrow$.

The energy is measured per iteration per vertical pipeline block. To fairly compare energy between the standard and WAD designs, one would scale the WAD results by the expected block activity factor. To compare uniform to non-uniform access designs, one would take a weighted average of fast and slow accesses of the non-uniform design (which is only a first-order approximation of the actual performance) depending on the register usage statistics.

[4] The absolute energies reported by the simulator have not been validated and are suspected to be much higher than actual values, however, we deem the relative energies to be reasonable.

5.1. Impact of Bank Size

We only evaluate the performance and energy consumption of register banks as a function of the number of registers; we do not account for the potential speedup of interleaving accesses to different banks. Our energy measurements for N='16' exclude the contribution of the static power dissipation of idle banks.

For the read port, reducing the bank size from 32 to 16 improves throughput by 9.6% for the standard format, by 7.6% for WAD, and reduces the read latency by 100 ps (33%). Energy per block per iteration is reduced by about 40% for the standard and WAD formats. Clearly, the majority of energy consumed is in the heavily loaded bit lines.

For the write port, reducing the bank size from 32 to 16 improves throughput by 15.4% for the standard format, by 14.2% for the WAD format, and reduces the write latency by 110 ps (21%). Energy per block per iteration is reduced by about 60% for the standard and WAD formats. We attribute the super-linear energy reduction to reduced bit-line leakage current and voltage drop, which factor into static power dissipation.

5.2. Impact of Width Adaptivity

Width adaptivity results in a read port cycle time increase of 7 to 10%, and a write port cycle time increase of 6 to 8%. There is no impact on the read or write latencies because the bit line loads and driving strengths remain the same. As expected, the energy overhead of WAD is around 25% for reading and writing because of the additional delimiter bit. However, the expected energy savings from the reduced width of a WAD integer (over 60%) far outweighs the overhead energy per block at the 4-bit granularity.

5.3. Impact of Non-Uniformity

Using N='16' as the baseline for evaluating '16n', we see that register nesting has a great impact (positive and negative) on the latencies and energy. For the read ports, the fast partition operates with 4% faster cycles, 60 ps (37%) less read latency, 10% less energy, and the slow partition operates with 84% longer cycles, 5 times longer read latency, and 50% more energy. For the write ports, the fast partition operates with 2% faster cycles, 40 ps (10%) less write latency, 7% less energy, and the slow partition operates with 40% longer cycles, 2.3 times longer write latency, and 50% more energy.

Table 5. Register file reading and writing `spice` results

f	N	read freq. (MHz)	read latency (ns)	read en./cy. (pJ)	write freq. (MHz)	write latency (ns)	write en./cy. (pJ)
S	32	537.0	0.323	26.59	409.2	0.528	27.45
	16	588.8	0.222	15.78	472.1	0.417	11.30
	16n	613.5	0.163	14.09	481.0	0.375	10.49
		322.3	1.149	23.25	337.4	0.963	17.68
W	32	496.4	0.323	33.18	384.0	0.528	34.90
	16	534.3	0.222	19.61	438.5	0.417	13.46
	16n	554.8	0.163	17.66	446.9	0.375	12.54
		285.9	1.149	29.52	311.5	0.963	21.18

Legend: 'f' (format): 'S' is standard (non-WAD), 'W' is WAD. 'N' (organization of registers): '32' is a single bank of 32 registers (Figure 3), '16' is a single bank of 16 registers, '16n' is a nested bank with 8 registers in the outer and inner partitions (Figure 11). For '16n', the upper numbers in each fused row correspond to the faster partition, and the lower numbers correspond to the slower partition.

Since the penalty of accessing the slower partition is relatively high for QDI nesting, one should justify when nesting will be superior in the average case than with uniform accesses. With probability-weighted averages for cycle times, latencies, and energies, we compute *breakeven probabilities* $\hat{r}$ in Table 6, which represent the fraction of accesses hitting the fast partition ('hit-fast') at which the average-case metric with non-uniform '16n' accesses is expected to equal that of the baseline uniform '16' accesses. Over 95% of accesses must hit in the fast partition to average faster cycles, 93% of accesses must hit-fast to average lower latencies, and 89% of accesses must hit-fast to average a net reduction in energy. Since all breakeven probabilities are below the cumulative frequency of the 16 most frequently used registers from Table 3, roughly 99%, nesting wins on all metrics. For a total of 32 physical registers, the best organization of the register core we have surveyed uses two nested banks of 16 registers '16n', each bank with 8 fast and 8 slow registers.

Table 6. Breakeven probabilities for nested read and write ports, '16n' vs. '16'. From left to right, $\hat{r}$'s are the breakeven probabilities for cycle time, latency, and energy per block per iteration, respectively.

	f	$\hat{r}_\tau$	$\hat{r}_l$	$\hat{r}_E$
read	S	95.3%	94.0%	81.6%
	W	95.9%		83.5%
write	S	95.6%	92.8%	88.7%
	W	95.6%		89.4%

What is the cost of *extending* an existing register bank with deeper registers? We have compared accesses to a single bank of 16 registers to acccesses to the outer bank of 16 registers with 16 registers in the inner partition '32n'. Table 7 shows the impact of appending a nested interconnect to a bank of 16 registers. The load of the interconnect increases the cycle times of read and write ports in the outer partition by 7 to 10%, which no more than 1.5 times a single register's load. The *decrease* of read latency (by 6 ps) is purely an artifact of measuring delay as the time difference between the last of multiple arriving inputs to the output transition, which does not model the Charlie Effect of transistors [28]. The insignificant change in latencies is very promising to asynchronous designs whose performance can be limited by the total forward latency through the datapath as opposed to the cycle time of local handshakes. The increase in energy includes the static power dissipation of an idle inner partition (of 16 registers), so the increase in dynamic energy of the outer partition is actually much less.

Table 7. Impact of appending a nested interconnect to a bank of 16 registers

	f	$\frac{\tau_{32n}}{\tau_{16}} - 1$	$\frac{l_{32n}}{l_{16}} - 1$	$\frac{E_{32n}}{E_{16}} - 1$
read	S	10.7%	-2.6%	25.8%
	W	8.8%		26.8%
write	S	8.3%	3.6%	41.7%
	W	7.7%		44.6%

The number of registers in the inner partition has *no effect* on the performance or dynamic energy of the outer partition of 16 registers. Thus, a nested register bank can accommodate a huge number of registers, while allowing a subset of registers to be accessed quickly and efficiently.

6. Conclusion

In this paper, we have addressed the problem of designing large, yet high-performance and low-energy asynchronous register files. We have identified and exploited the characteristics of typical operand values and register usage that can be leveraged more easily with asynchronous design than with synchronous design.

We have described the vertical pipeline transformation of the core, which improves the performance of the read and write ports *without* global word-line distribution as pipelined completion requires. To preserve pipelined, atomic, and read-write exclusive accesses to shared variables in the core, we implemented pipeline-locking in the control propagation. We have shown how much banking the core (in two) improves the cycle time, latency, and energy of register accesses over the unbanked base design.

To exploit the compressibility of frequent datapath values, we have implemented width-adaptive (WAD) reading and writing as an extension of the vertically pipelined design. The WAD design saves a considerable amount of energy in the average case, overcoming the 25% energy overhead of adding a delimiter bit per 4-bit block. The WAD read and write ports maintain the same latencies while suffering only a small increase in cycle time. The distributed nature of WAD integer compression is transparent to the control logic, making WAD trivial to implement.

The other important contribution of this paper is the introduction of non-uniform asynchronous register nesting to accommodate large register files while allowing a subset of registers to operate as fast as a small bank. Two premises motivate the use of nesting: 1) further banking is difficult or impossible due to the limitations imposed by its interconnect requirement, and 2) the majority of register accesses uses only a relatively small subset of the available architectural registers. We implemented nesting conservatively, under the QDI timing model, which resulted in a high penalty for accessing the inner partition. Nesting has the greatest potential benefit for the latencies and energy of register file accesses because these metrics are dominated by shared bit lines, whose loads are nearly halved by isolating the load of the seldom used partition. Where cycle time is not the performance bottleneck, nesting can improve performance by reducing the total latency through the datapath. Nesting is also appealing from a complexity perspective because the control (external to the register core) is *no different* than that for a non-nested core; delay insensitivity of asynchronous designs accommodates any variation in operation delays while maintaining correctness — the same cannot be said for synchronous designs.

We have demonstrated that register core banking, width adaptivity (atop vertical pipelining), and core nesting are all transformations that can be independently applied to asynchronous register files. We have not even begun to apply more traditional circuit techniques for accelerating accesses and reducing energy.

Our work on asynchronous non-uniformity introduces many possible directions for future work. At the circuit level, one may be interested in what timing assumptions can be made in the nested register core design for optimizing and reducing the penalty of the accessing the inner partition without compromising robustness. Reducing the slow-hit penalty would lower breakeven probabilities and make nesting more appealing to architectures whose register usage distributions are not as skewed. The lessons we have learned in designing non-uniform access register files can also be applied to designing asynchronous memories, and may lead to re-organizations of the memory hierarchy. Nesting may help register files in register-demanding superscalar and parallel machines grow while sustaining a required level of performance for a subset of registers. Some interesting architecture questions arise in light of non-uniform registers:

- How can register renaming leverage non-uniformity in mapping logical to physical registers?
- Can wise instruction scheduling hide the latency of slower reads?
- If non-uniformity is exposed in the ISA, can compiler support help?
- Since we can add an arbitrary number of registers with nesting without slowing down a critical set of registers, what would be good uses for extra (slower) registers?

Finally, our work emphasizes that asynchronous systems should never be designed around the models, assumptions, and constraints of the synchronous domain.

Acknowledgments

The research described in this paper was supported in part by the Multidisciplinary University Research Initiative (MURI) under the Office of Naval Research Contract N00014-00-1-0564, and in part by a National Science Foundation CAREER award under contract CCR 9984299. David Fang was supported in part by a National Defense Science and Engineering Graduate Fellowship.

References

[1] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th International Symposium on Microarchitecture*, December 2001.

[2] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture*, January 1999.

[3] David Brooks and Margaret Martonosi. Value-based clock gating and operation packing: Dynamic strategies for improving processor power and performance. *ACM Transactions on Computer Systems*, 18(2):89–126, May 2000.

[4] Ramon Canal, Antonio González, and James E. Smith. Very low power pipelines using significance compression. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 181–190, Monterrey, CA, December 2000.

[5] Keith D. Cooper and Timothy J. Harvey. Compiler-controlled memory. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, San Jose, CA, October 1998.

[6] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 316–325, Vancouver, Canada, June 2000.

[7] Uri Cummings, Andrew Lines, and Alain Martin. An asynchronous pipelined lattice structure filter. In *Proceedings of the 1st Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126–133, November 1994.

[8] Virantha N. Ekanayake and Rajit Manohar. Asynchronous DRAM design and synthesis. In *Proceedings of the 9th Annual International Symposium on Asynchronous Circuits and Systems*, pages 174–183, Vancouver, Canada, May 2003.

[9] David Fang. Width-adaptive and non-uniform access asynchronous register files. Master's thesis, Cornell University, December 2003.

[10] S. B. Furber, J. D. Garside, and D. A. Gilbert. AMULET3: A high-performance self-timed ARM microprocessor. In *Proceedings of the 1998 International Conference on Computer Design*, pages 247–252, Austin, TX, October 1998.

[11] S. B. Furber, J. D. Garside, S. Temple, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *Proceedings of the 3rd Annual International Symposium on Asynchronous Circuits and Systems*, pages 290–299, April 1997.

[12] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[13] M. Lewis and L. Brackenbury. Exploiting typical DSP access patterns for a low power multiported register bank. In *Proceedings of the 7th Annual International Symposium on Asynchronous Circuits and Systems*, Salt Lake City, UT, March 2001.

[14] Andrew M. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1995.

[15] Rajit Manohar. Width-adaptive data word architectures. In *Proceedings of the 19th Conference on Advanced Research in VLSI*, Salt Lake City, Utah, March 2001.

[16] Rajit Manohar and Alain J. Martin. Pipelined mutual exclusion and the design of an asynchronous microprocessor. Technical Report CSL-TR-2001-1017, Cornell Computer Systems Lab, November 2001.

[17] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Penzes, Robert Southworth, Uri V. Cummings, and Tak Kwan Lee. The design of an asynchronous MIPS R3000. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

[18] Lars S. Nielsen and Jens Sparsø. A low-power asynchronous data-path for a FIR filter bank. In *Proceedings of the 2nd Annual International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 197–207, Aizu-Wakamatsu, Fukushima, Japan, March 1996.

[19] Recep O. Ozdag and Peter A. Beerel. High-speed QDI asynchronous pipelines. In *Proceedings of the 7th Annual International Symposium on Asynchronous Circuits and Systems*, pages 13–22, Manchester, UK, April 2002.

[20] N. Paver, P. Day, S. B. Furber, J. D. Garside, and J.V. Woods. Register locking in an asynchronous microprocessor. In *Proceedings of the 1992 International Conference on Computer Design*, pages 351–355, Boston, MA, October 1992.

[21] A. Podlensky, G. Kristovsky, and A. Malshin. Multiport register file memory cell configuration for read operation. U.S. Patent 5,657,291, Sun Microsystems, Inc., Mountain View, CA, August 1997.

[22] M. Renaudin, P. Vivet, and F. Robin. ASPRO-216: a standard-cell QDI 16-bit RISC asynchronous processor. In *Proceedings of the 4th Annual International Symposium on Asynchronous Circuits and Systems*, San Diego, CA, March/April 1998.

[23] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register organization for media processing. In *Proceedings of the 6th IEEE Symposium on High-Performance Computer Architecture*, pages 375–386, January 2000.

[24] Richard M. Russell. The Cray-1 computer system. In Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors, *Readings in Computer Architecture*, pages 40–49. Morgan Kaufmann, 2000.

[25] John A. Swenson and Yale N. Patt. Hierarchical register for scientific computing. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 346–353, Saint Malo, France, 1988.

[26] Akihiro Takamura, Masashi Kuwako, Masashi Imai, Taro Fujii, Motokazu Ozawa, Izumi Fukasaku, Yoichiro Ueno, and Takashi Nanya. TITAC-2: A 32-bit asynchronous microprocessor based on scalable-delay-insensitive model. In *Proceedings of the 1997 International Conference on Computer Design*, pages 288–294, October 1997.

[27] Jessica Hui-Chun Tseng. Energy-efficient register file design. Master's thesis, MIT, 1999.

[28] Anthony J. Winstanley, Aurelien Garivier, and Mark R. Greenstreet. An event spacing experiment. In *Proceedings of the 8th Annual International Symposium on Asynchronous Circuits and Systems*, pages 47–56, Manchester, UK, April 2002.

[29] V. Zyuban and P. Kogge. The energy complexity of register files. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED '98)*, pages 305–310, August 1998.