Proceedings of the 2001 Conference on Advanced Research in VLSI

Width-Adaptive Data Word Architectures

Rajit Manohar
Computer Systems Laboratory
Electrical and Computer Engineering
Cornell University
Ithaca NY 14853, U.S.A.

Abstract

We discuss number representations for width-adaptive data word architectures. The number representations are self-delimiting, permitting asynchronous implementations with dynamic width adaptivity and reduced energy-complexity. We describe how these architectures can be realized with asynchronous VLSI techniques, and show that they exhibit better energy and throughput characteristics than traditional asynchronous implementations. We study some of the tradeoffs in the design of this class of architectures.

1: Introduction

We present width-adaptive data word architectures (WAD) as a new datapath design technique for constructing precision-adaptive energy-efficient datapath circuits.

Earlier studies of arithmetic have been either algorithmic (cf. [5]) or have been from the perspective of synchronous circuit design (cf. [4, 17]). Clearly almost any synchronous algorithm can be trivially adapted for asynchronous design. However, asynchronous design gives us greater freedom in choosing our algorithms and therefore the ability to explore a larger design space. In this paper we present one such exploration into the design of number representations suitable for asynchronous VLSI implementations.

Existing work in the design of adders and incrementers has demonstrated the advantage of average-case performance exhibited by asynchronous circuits. It is well known that binary adders have an average carry chain length that is logarithmic in the number of bits in the representation (cf. [2]). The obvious synchronous implementation of an adder obtained by cascading full adders in series cannot take advantage of this fact, since it is *possible* that the carry chain may be proportional to the number of bits. However, the corresponding asynchronous implementation (cf. [14]) can and does take advantage of this fact. Carry lookahead adders have worst case delays that are logarithmic in the number of bits. It is known that this is a lower bound on the worst-case delay for binary addition [20]. Using a combination of ripple-carry and carry lookahead techniques, asynchronous adders can be designed with sublogarithmic average-case latency [11]. In a similar way, a ripple-carry clocked incrementer might have to wait for the carry chain to propagate through all the bits of the incrementer; an asynchronous implementation takes, on average, constant time to complete the increment operation.

In general, algorithms that are not amenable to a decomposition into "balanced" computation stages can have poor performance characteristics in synchronous implementations since they would

affect global performance; an asynchronous implementation only pays a local penalty in performance when the circuit is used, one that may be justified by the resulting savings in area or energy, or by other improvements resulting from the algorithm.

In this paper we discuss new number representations that allow the design of a new class of asynchronous datapath circuits that exploit data-dependent behavior beyond the current state-of-the-art. The result is a datapath architecture that adapts the amount of energy it uses for computation in a data-dependent manner. Section 2 presents the new adaptive number representation and discusses design choices as well as possible asynchronous representations. Section 3 presents basic datapath circuits for aligned datapath structures. Section 4 presents datapath circuits suitable for a finely pipelined implementation. Section 5 discusses additional implementation options. Section 6 presents our simulation results, both from architectural simulations as well as circuit simulations. Section 7 presents related work, and we summarize our results in Section 8.

2: Variable-Width Number Representation

In this section we introduce the concept of width-adaptive numbers (WAD numbers), and discuss various representations of such numbers in the context of an asynchronous VLSI implementation.

Consider the simple numerical operation "1+0." Given a binary representation with 64 bits, computing this operation utilizes the same hardware resources as executing "329102910 + 32188748," even through computing "1 + 0" seems intuitively easier. Both computations also cause all the bit-slices of the datapath to be used for producing the result, thereby resulting in wasted energy in the case of the simpler operation of "1 + 0." What is missing in the underlying number representation is information about the number of significant digits in the representation. We introduce *width-adaptive number representations* as the solution to this problem.

2.1: Number Representation

A *width-adaptive binary number representation* is one where numbers are represented by the vector $\overline{\mathbf{a}} = (a_{n-1} \ldots a_2 a_1 a_0)_{\mathcal{V}}$, where each digit is a member of the set $\{0, 1, \underline{0}, \underline{1}\}$, and at least one digit is in the set $\{\underline{0}, \underline{1}\}$. The numerical value specified by the vector $(a_{n-1} \ldots a_2 a_1 a_0)_{\mathcal{V}}$ is defined to be the same as the value specified by the binary radix complement representation $\overline{\mathbf{b}} = (b_{n-1} \ldots b_2 b_1 b_0)_2$ defined by

$$b_i = \begin{cases} a_i & a_i = 0 \text{ or } a_i = 1 \\ 0 & a_i = \underline{0} \\ 1 & a_i = \underline{1} \end{cases}$$

In addition, we impose an *alignment condition*, namely that $a_i \in \{\underline{0}, \underline{1}\} \Rightarrow a_j = a_k$, for all $j > k$. Notice that the WAD representation is redundant, in that a particular number can be represented in multiple ways. We say the *width* of the WAD number is $1 + \text{argmin}_i(a_i \in \{\underline{0}, \underline{1}\})$.

As an example, the 3-digit representation for $(1)_{10}$ could be $(001)_{\mathcal{V}}$, $(\underline{0}01)_{\mathcal{V}}$, or $(\underline{0}\underline{0}1)_{\mathcal{V}}$. Because we enforce the alignment condition, we can unambiguously rewrite the third case as $(\underline{0}1)_{\mathcal{V}}$—the *compressed* representation for the number—because the alignment condition specifies the rest of the digits. This makes this representation have variable width, since knowing a particular digit is $\underline{0}$ or $\underline{1}$ uniquely specifies the digits that have higher significance. For example, $(\underline{0}1)_{\mathcal{V}}$ would represent $(1)_{10}$ *independent of the number of digits* used by the WAD representation.

The representation shown above needs two bits to represent a single digit, since there are four possible alternatives for each digit position. This might be considered wasteful in the number of bits necessary to represent WAD numbers. This can be addressed in two ways:

- by constructing WAD numbers with a higher radix;
- by only permitting digits in fixed positions in the number representation to be of the form $\underline{0}$ or $\underline{1}$. These positions could be uniformly or non-uniformly distributed throughout the number.

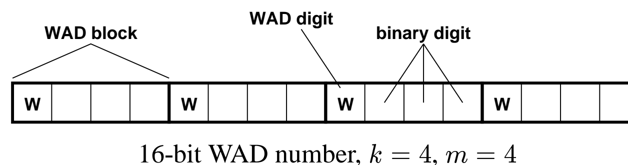
We adopt the latter approach when designing various circuits in this paper, since it simplifies the description of the resulting circuits. The circuits can be easily generalized to handle WAD numbers of higher radix [8].

The purpose of using WAD number representations is to improve performance and reduce the energy per operation when operations are being performed on small numbers. Asynchronous VLSI is a natural choice for the implementation of such numbers, because asynchronous design techniques can take advantage of average-case performance. A naïve clocked WAD datapath would have to wait for the worst-case delay in a WAD computation, and this would likely slow down the clock rate unnecessarily. A WAD number has data-dependent width, and asynchronous implementations can exploit this data-dependent representation to improve performance while reducing the average energy per operation.

2.2: Hybrid and Hierarchical Representations

There are several other hybrid WAD number representations possible. For instance, one might only use a WAD representation for part of the number, or for the part of a number where one expects a run of zeros or ones to occur. Another possibility is that a number could be represented using multiple WAD numbers. For example, a 32-bit number could be represented by 8 4-bit WAD numbers, or by 2 16-bit WAD numbers, or by 4 8-bit WAD numbers, etc. In addition, these individual WAD numbers could have differing radices. For example, a 4-bit radix 16 WAD number would use $\underline{0}$ and $\underline{1}$ to indicate the presence of all zeros or all ones. If we use 8 of these numbers to represent a 32-bit number, then the representation has the property that it "compresses" runs of zeros and ones of length 4 if the runs are aligned with the WAD number boundaries. The rest of this paper focuses on the basic implementation techniques for binary WAD representations. Some of the alternatives we have mentioned in this section are discussed in greater detail in [8].

Another possible number representation is one that is hierarchical. A particular number is broken down into blocks, and each block may be all zeros, all ones, or some other bit pattern. (This is similar to the case when we use numbers of higher radix.) The information about the block itself can be encoded using either a normal or an adaptive number representation. When using a normal number representation, the extra information associated with each block specifies whether the block is all zeros, all ones, or a mixture of zeros and ones. When using an adaptive representation, we would also encode whether the other blocks that contain digits that are of higher significance contain



16-bit WAD number, $k = 4$, $m = 4$

Figure 1. Width-adaptive number

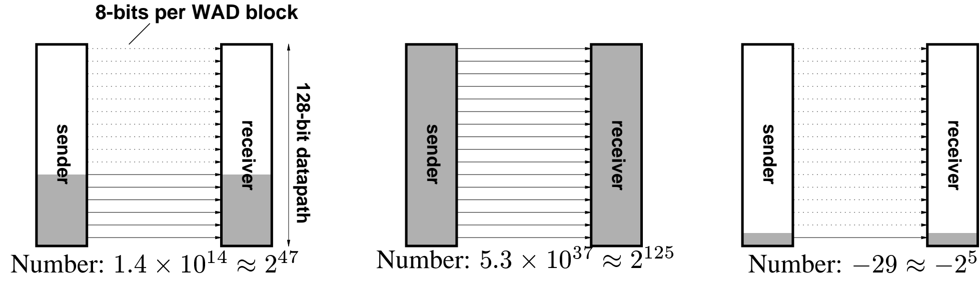


Figure 2. Transmitting numbers in a WAD datapath. Unshaded regions/dotted regions of the datapath do not have any switching activity.

all zeros or all ones as well. This can be repeated to achieve a hierarchical WAD representation. The discussion of these alternative representations and their design tradeoffs can be found in [8].

2.3: Asynchronous Encoding for Width-Adaptive Numbers

Codes that permit the receiver to detect when valid data is being transmitted independent of the delays in wires are said to be *delay-insensitive* (DI) codes (cf. [18]). Asynchronous circuits that are correct under the speed-independent, delay-insensitive, or quasi-delay-insensitive (QDI) model use DI codes for data communication between computation blocks. Specific datapath design techniques for QDI circuit design that use DI codes for communication and computation are discussed by Martin [14].

The most commonly used DI code is the one-hot code that uses k wires to represent k different numbers. Initially, all k wires are 0. Data value i is transmitted by setting the ith wire to 1. The code returns to the initial state before the next data value is transmitted. One-hot codes for $k = 2$ are known as *dual-rail codes*, and those for $k = 4$ are known as *quad-rail codes*. These two codes are commonly used for data encoding.

In traditional number representations, binary digits are typically encoded using dual-rail codes. Following this approach, a single radix-2 WAD digit can be represented using a quad-rail code, since the digit can take on 4 different values. To reduce the number of wires needed to encode WAD numbers, we use one WAD digit for every $(k-1)$ binary digits; we refer to this combination of a WAD digit and $(k-1)$ binary digits as a WAD *block* of size k. Each such block uses $(2k+2)$ rails to represent a WAD number, and a single WAD block is transmitted using a standard DI code consisting of $(2k+2)$ wires as outlined above. In what follows, we assume that the N bit datapath is divided into m WAD blocks of size k (and therefore $N = mk$). Figure 1 pictorally depicts a 16-bit WAD number with $k = 4$.

Transmitting data in an N-bit WAD datapath is significantly different from transmitting data in a traditional datapath. If a particular WAD block has a $\underline{0}$ or $\underline{1}$ as its most significant digit, then we *do not transmit* any WAD blocks that would specify bit positions more significant that the block containing the $\underline{0}$ or $\underline{1}$. For this reason, we refer to $\underline{0}$ and $\underline{1}$ as *delimiters* of the WAD representation, and WAD block containing them is called a *delimiting block*. Figure 2 shows an example of transmitting two different WAD numbers using the same datapath, showing how part of the datapath not required for transmitting a digit exhibits no switching activity.

If a WAD number has n significant digits, then we only use $k\lceil n/k \rceil$ of the WAD datapath to communicate the number. The remaining $(1 - \lceil n/k \rceil / m)$ fraction of the WAD datapath does not exhibit *any* switching activity. This means that small numbers can be communicated with much lower energy than larger numbers. What is unique about this datapath structure is that the

representation adapts to the number of significant digits required without any software intervention, and the full datapath is available for N-bit operations if necessary, as illustrated in Figure 2.

3: Aligned WAD Datapath Structure

An aligned datapath structure is one where operations on adjacent bits in the datapath are performed at about the same time. Examples of aligned datapaths include the datapath for the first asynchronous microprocessor [15] as well as the MiniMIPS processor [16]. In this section we describe how aligned WAD datapaths are constructed. Almost all modern processors use aligned datapaths. We use CHP (communicating hardware processes) and handshaking expansions (restricted CHP) to describe the circuits in this paper. A brief description of the notation is provided in the appendix.

3.1: Standard Communication Cells

A standard technique applied when designing asynchronous circuits is to separate control from data. For instance, a one-place FIFO

$$*[\ L?x; R!x\ ]$$

would be replaced with one control process and two datapath processes, as shown below:

$$*[\ L'; R'\ ] \parallel *[\ L' \bullet L?x\ ] \parallel *[\ R' \bullet R!x\ ]$$

This transformation is known as control-data decomposition, and was extensively used in the design of the first asynchronous microprocessor [15]. We show how standard datapath processes of the form $*[L' \bullet L?x]$ (receivers) and $*[R' \bullet R!x]$ (senders) can be designed for WAD datapaths.

WAD Aligned Sender Process. Processes of the type $*[C \bullet D!x]$ are designed by decomposing them into a number of concurrent cells, each capable of handling one bit of data along with a completion tree to combine individual acknowledge signals into a single acknowledge for the C channel [14]. We begin by providing the compilation of a cell that is capable of handling one WAD block following Martin's synthesis procedure [13].

The cell has k bits of state, stored in dual-rail state variables (xt_i, xf_i) for $0 \leq i < k$. The dual-rail state bit (xtt, xff) determines if the cell is a delimiter in the datapath (the cell is a delimiter if xtt is **true**).

The control input to a single cell is received on channel C encoded using two wires $(ci; co)$. The WAD block of size k is communicated over a channel D using a single acknowledge di and data rails that are encoded as follows:

- The first $k-1$ bits named are encoded using a dual-rail code (a quad-rail encoding is also possible for these bits; we omit this minor change in the interests of clarity). The code for data bit i is named (dt_i, df_i).
- If the data block being transmitted over channel D is not a delimiter, then (dt_{k-1}, df_{k-1}) encodes the data bit (xt_{k-1}, xf_{k-1}); otherwise, (dtt, dff) encodes the data bit (xt_{k-1}, xf_{k-1}). In other words, $(dt_{k-1}, df_{k-1}, dtt, dff)$ form a quad-rail code that encodes the most significant bit of the WAD block.

To summarize, channel D uses $(2k+2)$ wires for data and a single acknowledge di. The handshaking expansion for a sender, using a passive port C and an active port D is given by:

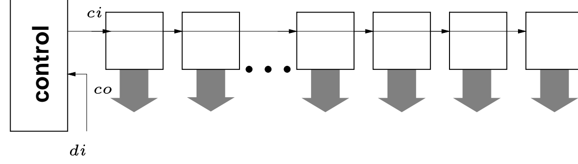


Figure 3. Aligned WAD sender showing circuit structure. The boxes correspond to the individual WAD blocks, and the arrows correspond to the $(2k+2)$ wires per cell for data communication.

$$
\begin{array}{l}
*[\,[ci];(\|\ i:k-1:\ \ [xt_i \longrightarrow dt_i\uparrow [] xf_i \longrightarrow df_i\uparrow]\,),\\
\quad [xtt \longrightarrow [xt_{k-1} \longrightarrow dtt\uparrow [] xf_{k-1} \longrightarrow dff\uparrow]\\
\quad [] xff \longrightarrow [xt_{k-1} \longrightarrow dt_{k-1}\uparrow [] xf_{k-1} \longrightarrow df_{k-1}\uparrow]\\
\quad];\\
\quad [di];co\uparrow;[\neg ci];(\|\ i:k:\ \ dt_i\downarrow, df_i\downarrow), dtt\downarrow, dff\downarrow;[\neg di];co\downarrow\\
\ ]
\end{array}
$$

The reader is referred to [8] for the circuit implementation of this handshaking expansion.

A WAD sender contains m processes of the form described above. Since the single signal di for the data communication acknowledges the transmitted data for every block, the full N-bit datapath compilation is performed by simply sharing signals ci, di, and co across each sender process. Figure 3 shows the compilation of an N-bit sender. Each blue square denotes one of the sender processes.

WAD Aligned Receiver Process. The receiver process $*[C \bullet D?x]$ is slightly more complex. Intuitively, the problem arises because overwriting a WAD variable changes the number of datapath processes that hold a valid data value. Once the operation is complete, all WAD blocks beyond the delimiter must have their local state variables all set to **false** for the sender process to function correctly. We split the write operation to a WAD word into two distinct phases: the reset phase where all the state bits are set to **false**, and the write phase where the WAD word is read in from channel D.

For the reset phase, we would like to ensure that we do not exercise the datapath cells that lie beyond the delimiter cell. Therefore, a handshaking expansion of the form:

$$*[\,[ri];(\|\ i:k: xt_i\downarrow, xf_i\downarrow), xtt\downarrow, xff\downarrow; ro\uparrow;[\neg ri];ro\downarrow]$$

for the reset phase would not be acceptable, because the ro signal would switch in every datapath cell. To eliminate this problem, we introduce a dual-rail acknowledge that indicates when we have hit the delimiter cell. If the cell holds valid data but is not a delimiter, the cfo signal is used as the acknowledge; if the cell holds valid data and is a delimiter, the cto signal is used as the acknowledge; if the cell does not hold valid data, none of the acknowledge wires are raised. The reset phase is described by:

$$*[\,[ri];(\|\ i:k: xt_i\downarrow, xf_i\downarrow);[xtt \longrightarrow xtt\downarrow; rto\uparrow [] xff \longrightarrow xff\downarrow; rfo\uparrow\ ];[\neg ri];rto\downarrow, rfo\downarrow]$$

The reader is referred to [8] for the circuit implementation of this handshaking expansion.

Once the block has been reset, we can proceed with reading a value from input channel D and write the value received on D into the local state variables. The write phase is described by

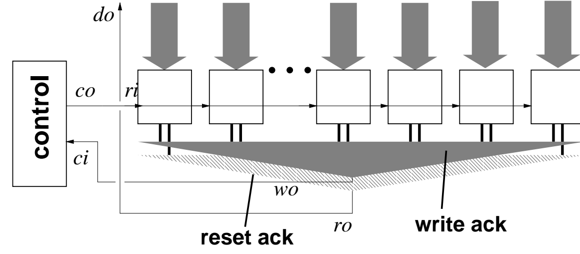


Figure 4. Aligned WAD receiver showing one possible combination of reset and write phases. The boxes correspond to the individual WAD blocks, and the arrows correspond to the $(2k+2)$ wires per cell for data communication.

$$
\begin{aligned}
*[\,&[wi];\, do{\uparrow};\, (\| \ i : k-1 : [dt_i \longrightarrow xt_i{\uparrow} \,\square\, df_i \longrightarrow xf_i{\uparrow}]),\\
&[dtt \vee dff \longrightarrow xtt{\uparrow} \,\square\, dt_{k-1} \vee df_{k-1} \longrightarrow xff{\uparrow}],\\
&[dtt \vee dt_{k-1} \longrightarrow xt_{k-1}{\uparrow} \,\square\, dff \vee df_{k-1} \longrightarrow xf_{k-1}{\uparrow}];\\
&wo{\uparrow};\, [\neg wi];\, do{\downarrow};\, [(\wedge k : \neg dt_i \wedge \neg dt_i) \wedge \neg dtt \wedge \neg dff];\, wo{\downarrow}\\
]&
\end{aligned}
$$

(assuming an active protocol on port D). The reader is referred to [8] for the circuit implementation of this handshaking expansion. An interesting side-effect of separating the reset phase from the write phase is that the write acknowledge circuitry is simplified. However, the overall circuit is slower because we have to reset the circuit before performing the write.

We make the following observations to design the acknowledge circuit for the complete datapath. Consider two adjacent WAD blocks, j and $(j+1)$ ($j+1$ holds bits that are more significant) with acknowledge signals (cto, cfo) and (dto, dfo) respectively. If the acknowledge cto is **true** for block j then block j is a delimiter, and block $(j+1)$ does not participate in this particular communication action. If cfo is **true** for block j, block $(j+1)$ participates in this communication action because j is not a delimiting block. Therefore, acknowledges (cto, cfo) and (dto, dfo) can be combined to produce a single acknowledge (eto, efo) as follows:

$$
\begin{aligned}
*[\,&[cto \longrightarrow eto{\uparrow} \,\square\, cfo \longrightarrow [dto \longrightarrow eto{\uparrow} \,\square\, dfo \longrightarrow efo{\uparrow}]];\\
&[\neg cto \wedge \neg cfo \wedge \neg dto \wedge \neg dfo];\, eto{\downarrow}, efo{\downarrow}\ \ ]
\end{aligned}
$$

The reader is referred to [8] for the circuit implementation of this handshaking expansion. This transformation can be applied recursively to obtain an acknowledge pair (ato, afo) for the entire datapath. For this pair, we know that afo can never be **true** because a WAD datapath contains exactly one delimiter block. Therefore, ato is the acknowledge for the entire reset or write phase.

There are several ways we can combine the reset and write phases into a single communication action. The simplest way is to simply use the following handshaking expansion, that translates the write request on input channel C into a reset request on channel R followed by a write request on channel W:

$$
*[[ci];\, ro{\uparrow};\, [ri];\, wo{\uparrow};\, [wi];\, co{\uparrow};\, [\neg ci];\, ro{\downarrow};\, [\neg ri];\, wo{\downarrow};\, [\neg wi];\, co{\downarrow}]
$$

This combination is illustrated in Figure 4. We can overlap the second half of the handshake on both R and W by using the following handshaking expansion:

$$
*[[ci];\, ro{\uparrow};\, [ri];\, wo{\uparrow}, ro{\downarrow};\, [wi];\, co{\uparrow};\, [\neg ci];\, wo{\downarrow};\, [\neg wi \wedge \neg ri];\, co{\downarrow}]
$$

The reader is referred to [8] for the circuit implementation of this handshaking expansion.

3.2: Function Computation

Function block compilation provides a canonical way to translate the computation of a function into an asynchronous QDI implementation [14]. In this section we show how single argument and multiple argument functions can be computed for aligned WAD datapaths.

Consider the computation of a function by the CHP process $*[L?x; R!f(x)]$. The first step in designing the production rules for this process is control data decomposition. Applying this transformation, we obtain:

$$*[\ L?x; R!f(x)\ ] \ \triangleright\ *[\ L'; C\ ] \ \|\ *[\ L' \bullet L?x\ ] \ \|\ *[\ C \bullet R!f(x)\ ]$$

We focus on the compilation of the third process above. Following Martin [14], we use the notation $X := e$ to indicate that the data rails of channel X are set to a valid value e, and the notation $X \Downarrow$ to indicate that the data rails of channel X are set to the neutral value (all rails **false**). Communication channels are encoded using the WAD representation discussed in the previous section.

The handshaking expansion for $*[C \bullet R!f(x)]$ is given by:

$$*[[ci]; R := f(x); [ri]; co\uparrow; [\neg ci]; R \Downarrow; [\neg ri]; co\downarrow\ ]$$

This handshaking expansion can be broken down into two concurrent parts, as shown below:

$$\begin{array}{ll} & *[[ci]; X := x; [ri]; co\uparrow; [\neg ci]; X \Downarrow; [\neg ri]; co\downarrow\ ] \\ \| & *[[v(X)]; R := f(X); [n(X)]; R \Downarrow\ ] \end{array}$$

where $[v(X)]$ denotes waiting for X to have a valid value, and $[n(X)]$ denotes waiting for X to have a neutral value [14]. The first part is simply the process $*[C \bullet X!x]$, that we have already compiled. We focus on processes of the form $*[[v(X)]; R := f(X); [n(X)]; R \Downarrow]$. Without repeating all the arguments provided for function block compilation, we state that we can perform the following decomposition:

- Let $Y_0, \ldots, Y_{N-1}$ be set of subcodes of the encoding of X such that they cover the entire code X;
- Let the output bit R_i only depend on the value of X specified by the subcode Y_i

Then,

$$*[[v(X)]; R := f(X); [n(X)]; R \Downarrow\ ] \equiv (\| \ k :: *[[v(Y_k)]; R_k := f_k(Y_k); [n(Y_k)]; R_k \Downarrow])$$

where the function f_k is simply the part of the function that computes output R_k. This is simply the function block transformation outlined in [14].

This general strategy applies to aligned WAD datapaths as well. The main problem with computing functions using an aligned datapath is illustrated by considering a simple example where we compute a function of two arguments.

Consider computing the logical AND of two inputs on channels A and B using the function block strategy, where the datapath is to be constructed using an aligned, 16-bit WAD code with k=4 and m=4. If we look at bit 14 of the output, not only does its value depend on the corresponding inputs on A and B, but it also depends on the WAD digit in WAD block 0, 1, and 2! Another way to think about this problem is that the two inputs received on A and B might have different widths. (Note that whether or not this particular problem occurs depends on the nature of the function being computed.)

In general, we can solve this problem by introducing an *alignment* stage that ensures that the two inputs have the same width by exploiting the redundancy in the WAD number representation. Aligning the widths of two numbers is equivalent to padding the narrower number so that it matches

the width of the wider number. The circuit for alignment is constructed by using the following observation: the two inputs have different widths if and only if there is a digit where one of the two inputs is in $\{0, 1\}$ and the other input is in $\{\underline{0}, \underline{1}\}$. In this circumstance, we must widen the narrower number. Details of this construction can be found in [8]. Once we have aligned the two inputs, function block style computation of functions like logical AND is straightforward. The only observation that is necessary to complete the compilation is that logical AND can be extended to $\underline{0}$ and $\underline{1}$ as follows: $\underline{0} \wedge \underline{0} = \underline{0}$; $\underline{0} \wedge \underline{1} = \underline{0}$; $\underline{1} \wedge \underline{1} = \underline{1}$.

4: Pipelined WAD Datapath Structure

A disadvantage of the compilation for WAD blocks shown above is that the control signals are still broadcast to the entire datapath. Therefore, while we have reduced the amount of energy dissipated by the datapath cells, we have not reduced the contribution of the control cells to the total energy. Simply put, we still have an $\mathcal{O}(N)$ energy term from the capacitive load on the control distribution wires.

The average cycle time of the WAD system designed in the previous section scales as $\mathcal{O}(\log N)$, This is asymptotically the same as the cycle time for the style of datapath design used for the first asynchronous microprocessor [15]. This is because the width-adaptive completion tree still has logarithmic depth. In practice, the completion tree delay may have smaller constant factors because it has a data-dependent delay [8]. The energy used by the system is $\mathcal{O}(kE[b])$, where $E[b]$ is the expected number of WAD blocks used by the computation (ignoring the $\mathcal{O}(N)$ term from the control distribution wires for the moment).

More recent developments in the design of pipelined asynchronous circuits have resulted in the elimination of the completion tree bottleneck for wide datapaths, and $\mathcal{O}(1)$ throughput for finely pipelined asynchronous computations [16]. In this section we examine how WAD datapaths are designed such that they exhibit $\mathcal{O}(1)$ throughput. We begin with a few observations as to why this extension is non-trivial.

The key property that makes finely pipelined QDI circuits faster is that the control distribution is decoupled from the datapath, and the different datapath cells are not tightly synchronized. As a result, datapath processes for a particular function that are physically adjacent to one another might be processing different data items. For example, a finely pipelined implementation of the buffer $*[L?x; R!x]$ uses projection to decompose the process into:

$$(\| \ i :: *[\ L_i?x_i; R_i!x_i\ ])$$

where each process operates on a fixed number of bits of x, and the different processes are not synchronized [9]. In the general case, appropriate control distribution ensures that the datapath operates correctly. Note that separate cells in the datapath are not synchronized with one another, and it is this transformation that results in improved throughput.

Decoupled datapath cells can lead to the situation shown in Figure 5. In the example shown, the WAD word 2 (shown in red) follows WAD word 1 on the communication channels. However, because the two word lengths are different, datapath cells for a single operation may contain parts of different words. Figure 5 shows a case where the bottom blocks of word 2 are prevented from advancing because of word 1, but the top blocks can advance into the empty datapath cells above word 1. When this situation is combined with pipelined completion detection [16], the control arriving at a cell may no longer be matched with the data arriving at the cell. The problem arises because the control distribution network needs to know which datapath cell contains the WAD data word delimiter so that the control distribution can stop at that cell.

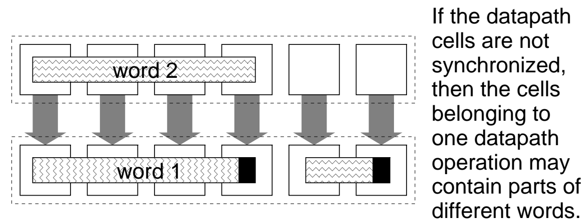


Figure 5. The two different WAD words are shown with with a solid delimiter. The dashed lines group together datapath cells used to implement one datapath operation, and the arrows denote communication channels.

We avoid these problems by adopting a *block-skewed* WAD datapath design, similar to traditional bit-skewed datapaths. The major difference between these datapaths and traditional bit-skewed ones is that the computation might terminate without using all m WAD blocks (once we hit a delimiter block). Note that using a bit-skewed approach would result in a poor clocked implementation whenever we have to examine all the bits (for instance, doing a zero detect for a branch comparison).

We illustrate this design style by showing the block-skewed implementation of logical operations as well as binary addition. As usual, control distribution is handled by rippling the control vertically through the datapath. To permit the entire computation to be finely pipelined, each WAD block uses a separate acknowledge signal, instead of a shared acknowledge signal for all N datapath bits. Therefore, the number of wires needed for data communication in the block-skewed WAD datapath is $m(2k+3)$, as opposed to $m(2k+2)+1$ for the aligned datapath—an increase of $(m-1)$ wires. Note that this wiring overhead is similar to the overhead required for implementing high-throughput asynchronous QDI circuits using pipelined completion detection [16].

In what follows, we assume the system that incorporates the block-skewed datapath structure is locally slack elastic [10]. This assumption is necessary to permit the design of a finely pipelined asynchronous computation using projection-based synthesis techniques [9]. This assumption is not overly restrictive, since the assumption of local slack elasticity holds for the entire MiniMIPS design [16]. Also, we provide the CHP construction for datapath operations when $k = 1$, i.e., when each digit in the datapath is a WAD digit. The construction for $k > 1$ is straightforward [8].

4.1: Block-Skewed Logical Operations

Let $l(x, y)$ be a function that takes two bits and produces one bit. The function l represents the logical operation to be performed for each output bit that is computed. To describe a block-skewed implementation of this operation, we introduce function $done(x)$ to indicate that $x \in \{\underline{0}, \underline{1}\}$.

Consider a particular bit-position i in the datapath. There are four possible cases for this bit-position:

- An input is pending on channels A and B;

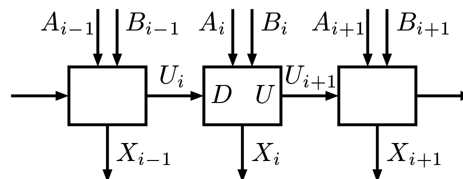


Figure 6. Process connections for block-skewed logical operations.

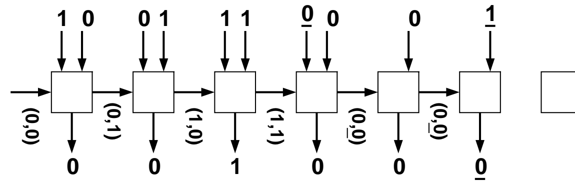


Figure 7. Block-skewed AND, showing the channels that are used.

- An input is pending on channel A;
- An input is pending on channel B;
- No inputs are pending on A or B.

The problem discussed earlier also implies that when inputs are pending on both A and B, these two inputs might correspond to *different* operand pairs for the logical operation. The only datapath digit that is guaranteed to receive an input on A and B, and also knows that those two inputs are part of the same operand pair is the least significant digit. If we propagate the information about when a WAD number ends "up" the datapath, then we can use this information to prevent the datapath problem discussed previously. The information we propagate must be sufficient to determine if the input on A and B corresponds to the current operand pair for the computation block. When we see the delimiter digit for both A and B, we know that we need not use the rest of the datapath for the computation.

The computation is performed by a linear array of processes, each having three inputs and two outputs: A and B are inputs corresponding to the WAD block; D is an input that is used to communicate control information among adjacent processes; X is an output channel corresponding to the WAD block; U is an output channel used to communicate control information vertically among adjacent processes. Figure 6 pictorially depicts the channel connections among the processes.

The logical operation is performed by the following CHP process:

$$*[D?(a,b);\ [done(a) \longrightarrow B?b\ \, [\!]\, done(b) \longrightarrow A?a\ \, [\!]\ \textbf{else} \longrightarrow A?a, B?b];$$
$$X!l(a,b),\ \ [done(a) \wedge done(b) \longrightarrow \textbf{skip}\ \, [\!]\ \textbf{else} \longrightarrow U!(a,b)]\ \ ]$$

The input on channel D at the least significant bit position can be the pair $(0,0)$; in general it can be any pair (x,y) such that $\neg done(x) \wedge \neg done(y)$ holds. Figure 7 shows an example of how the datapath computes the logical AND of two WAD numbers, with arrows indicating the channels that are used for the computation. Note that while some channels have not been shown, those channels might have pending inputs; they are simply ignored by the operation being performed.

From the CHP, one can immediately conclude that any pair (a,b) sent on channel U satisfies $\neg done(a) \vee \neg done(b)$. Therefore, this property holds for any input on channel D (since the D and U channels are connected to one another—see Figure 6). From the CHP program above, we can conclude that if $\neg done(a)$ holds, we need not know whether $a = 0$ or $a = 1$; it is sufficient to know that $a \in \{0,1\}$. This observation can be used to reduce the number of wires per data item from 4 to 3 in the representation used to encode a and b for channels U and D, or to even combine the two into a single 1-of-5 code.

4.2: Block-Skewed Binary Addition

Binary addition can be performed using a similar structure to the one for logical operations. Along with propagating information about whether or not a delimiter digit has been reached for an input operand, we also propagate the carry in the usual way. When the data reaches the most

a	b	cin	$cout$	sum
$\underline{0}$	$\underline{0}$	0	X	$\underline{0}$
$\underline{0}$	$\underline{0}$	1	$\underline{0}$	1
$\underline{0}$	$\underline{1}$	0	X	$\underline{1}$
$\underline{0}$	$\underline{1}$	1	X	$\underline{0}$
$\underline{1}$	$\underline{1}$	0	$\underline{1}$	0
$\underline{1}$	$\underline{1}$	1	X	$\underline{1}$

Delimiter Digits, non-MSB

a	b	cin	$cout$	sum
$\underline{0}$	$\underline{0}$	0	X	$\underline{0}$
$\underline{0}$	$\underline{0}$	1	X	$\underline{1}$
$\underline{0}$	$\underline{1}$	0	X	$\underline{1}$
$\underline{0}$	$\underline{1}$	1	X	$\underline{0}$
$\underline{1}$	$\underline{1}$	0	X	$\underline{0}$
$\underline{1}$	$\underline{1}$	1	X	$\underline{1}$

Delimiter Digits, MSB

Table 1. Addition tables for delimiter digits ("X" indicates no output).

significant digit that contains an input for either operand, we use Table 1 to determine the sum and carry. The carry-in is, therefore, a WAD digit. The CHP for this is given below:

$$\begin{array}{l}
*[D?(a,b,ci);\\
\quad [\mathit{done}(ci) \longrightarrow \textbf{skip}\\
\quad [\!] \neg \mathit{done}(ci) \longrightarrow [\mathit{done}(a) \longrightarrow B?b \,[\!]\, \mathit{done}(b) \longrightarrow A?a \,[\!]\, \textbf{else} \longrightarrow A?a, B?b]\\
\quad]; X!\mathit{sum}(a,b,ci),\\
\quad [\mathit{done}(a) \wedge \mathit{done}(b) \longrightarrow [\neg \mathit{done}(\mathit{sum}(a,b,ci)) \longrightarrow U!(\underline{0},\underline{0},a) \,[\!]\, \textbf{else} \longrightarrow \textbf{skip}]\\
\quad [\!]\, \textbf{else} \longrightarrow U!(a,b,\mathit{cout}(a,b,\mathit{cin}))\\
\quad]]
\end{array}$$

When $\neg done(a) \vee \neg done(b)$ holds, the sum and carry functions correspond to the usual sum and carry-out functions used for binary addition; the underscores on the digits can be ignored for purposes of computation. When $done(a) \wedge done(b)$ holds, then Table 1 determines the sum and carry-out.

MSB Circuit. The circuit at the most significant digit in the entire datapath has to be modified. Since the two WAD inputs must have reached their delimiter digit when this process is used, we can assert that $done(a) \wedge done(b)$ holds immediately before the $X!sum(a,b,ci)$ action. Table 1 shows how the addition table must be modified to preserve the property that the WAD number ends in a delimiter digit. The CHP can be simplified accordingly.

The adder can be modified to perform a subtraction by taking the two's complement of one of the inputs. This can be done by using a carry-in of 1 and by complementing all the bits in the usual way. Note that the complement of $\underline{0}$ is $\underline{1}$, and the complement of $\underline{1}$ is $\underline{0}$; therefore, the WAD representation does not introduce additional complications when performing the complement operation.

4.3: Block-Skewed Width Alignment

Both logical operations and addition have a similar structure. This similarity is a result of the part of the computation that ensures the two inputs have the same width. If we need to perform this alignment by itself (normally it should be folded into the function computation as done in the previous two sections), it can be done by the following CHP process:

$$\begin{array}{l}
*[D?(a,b); [\mathit{done}(a) \longrightarrow B?b \,[\!]\, \mathit{done}(b) \longrightarrow A?a \,[\!]\, \textbf{else} \longrightarrow A?a, B?b];\\
\quad Xa!a, Xb!b, [\mathit{done}(a) \wedge \mathit{done}(b) \longrightarrow \textbf{skip} \,[\!]\, \textbf{else} \longrightarrow U!(a,b)]\ ]
\end{array}$$

The outputs on the channels Xa and Xb in the datapath contain copies of the inputs A and B respectively. However, the copies of A and B are guaranteed to have the same width. This circuit

could be used on the output of a register file to ensure that the two operands being fetched for a particular function unit have the same width.

4.4: Block-Skewed Comparator

Given two WAD numbers, the comparison $a < b$ will take time that is proportional to the number with the largest width. Comparing $a = b$ can be made faster on average, because we can determine $a \neq b$ as soon as we reach a digit where the numbers a and b differ. If we know that the representation for a particular number a uses the most compact representation possible (for instance, the number is not represented as $\underline{0}01$ when it could be represented as $\underline{0}1$), then a compare to zero is a constant time operation since all we need to know is whether the least significant digit is $\underline{0}$. We have used this technique to design a number of other datapath elements, and the interested reader is referred to [8] for details.

5: Compact Representations and Compaction

The WAD representation is redundant, permitting a number to be represented in multiple ways. This simplifies the implementation of various datapath operations, but the result of the operation might not be represented in the most compact manner possible. For example, consider the operation $\underline{1}101 - \underline{1}100$. The result of this operation would be $\underline{0}001$. (The reader is invited to check this by examining Table 1.) While this is correct, we would prefer to have the output be $\underline{0}1$ since that would reduce the number of datapath digits necessary to represent the number. When the representation has the property that it uses the minimum number of datapath digits possible, we say the representation is *fully compacted*.

Full compaction circuitry introduces throughput overhead, because we may have to check all the transmitted WAD blocks before we can determine the correct fully compacted representation. If the number representation is aligned, it is possible to design a full-compaction circuit with average-case latency $\mathcal{O}(E[\log p])$, where $E[\log p]$ is the expected value of the logarithm of the number of digits in the WAD input [8]. When using a pipelined, block-skewed datapath the input digits arrive skewed in time and this imposes an additional overhead in full compaction.

Instead, we can gradually compact the representation as the number is transmitted through various routing circuits (splits, merges, crossbars, etc) that are usually present in any architecture. We can design a *one-step compaction* circuit that attempts to compress the representation by one block if possible. For example, the one-step compaction circuit would take $\underline{0}001$ and compact it to $\underline{0}01$. The advantage of using one-step compaction is that it only introduces a constant latency overhead, and it does not asymptotically reduce the throughput of the system.

The design of the various compaction circuits along with an analysis of their behavior can be found in [8]. Full-compaction would be useful if we keep re-using a particular piece of data over and over again, because the number of digits in its WAD representation may grow unchecked. If we expect that re-use is low, then the faster one-step compaction step might suffice.

6: Simulation Results

In this section we present some of our results from both circuit and architectural simulation of WAD datapaths, as well as some analysis of the throughput and energy characteristics of WAD datapaths with more traditional datapath design approaches. The architectural simulations are used

run	m88ksim	gcc	compress	li	ijpeg	perl
	Datapath: 1×32-bit WAD number					
1/n	11.3	12.2	16.9	14.2	13.2	13.3
1/o	9.3	11.1	14.1	13.6	11.5	12.1
1/f	9.1	10.6	11.5	13.3	11.2	11.5
2/n	13.4	13.6	18.1	15.2	14.9	14.6
2/o	9.9	11.6	14.6	14.1	12.1	12.6
2/f	9.6	11.2	12.1	13.8	11.8	12.0
4/n	16.6	16.2	20.4	17.3	17.8	17.0
4/o	11.2	12.8	15.9	15.1	13.5	13.7
4/f	11.0	12.3	13.3	14.8	13.3	13.1
8/n	21.4	21.1	24.9	21.5	22.4	21.2
8/o	13.7	15.1	18.2	17.3	16.3	15.9
8/f	13.6	14.7	15.6	16.9	16.1	15.4
	Datapath: 4×8-bit WAD number					
1/f	9.5	10.9	11.9	13.8	11.7	11.8
2/f	13.1	14.5	14.8	15.7	14.5	14.8
	Datapath: 4×8-bit radix-256 WAD number					
8/f	11.2	13.0	13.6	14.7	13.4	13.3

Table 2. Average operand width for a 32-bit MIPS processor.

to determine typical widths of numbers used by applications. The results validate the qualitative discussions we provided earlier.

6.1: Architectural Simulation

We used `chpsim`, a compiled asynchronous CHP simulator that generates an estimate of both energy and cycle time for an asynchronous computation. We used the CHP for the MiniMIPS asynchronous processor [16], and determined the WAD block width used by each 32-bit integer function unit for a number of different benchmarks from the SPEC95 benchmark suite. The graphs in Figure 8 show the cumulative distribution function of the widths of the data transferred to and from the integer function units for an architecture similar to the MiniMIPS processor. Points in Figure 8 that are closer to the top left corner of the graph correspond to better cases for WAD designs, because that means that more operands had smaller operand widths. Each graph shows three distribution functions: "no compaction" corresponds to the case when we do not use any compaction circuitry on the output of function units; "full compaction" corresponds to the case when we use the relatively slow full-compaction circuitry; "one-step compaction" corresponds to the case when we use the fast, one-step compaction strategy. We also gathered this information for block sizes 1, 2, 4, and 8. The results show that we need not implement full compaction circuitry in practice, as register re-use does not happen frequently enough to cause operand widths to grow significantly.

Table 2 summarizes the operand width averages for 6 different benchmarks from the SPEC95 integer benchmark suite. The distribution plots for these benchmarks can be found in [8]. The rows are labelled by block width followed by "f" for full compaction, "n" for no compaction, and "o" for

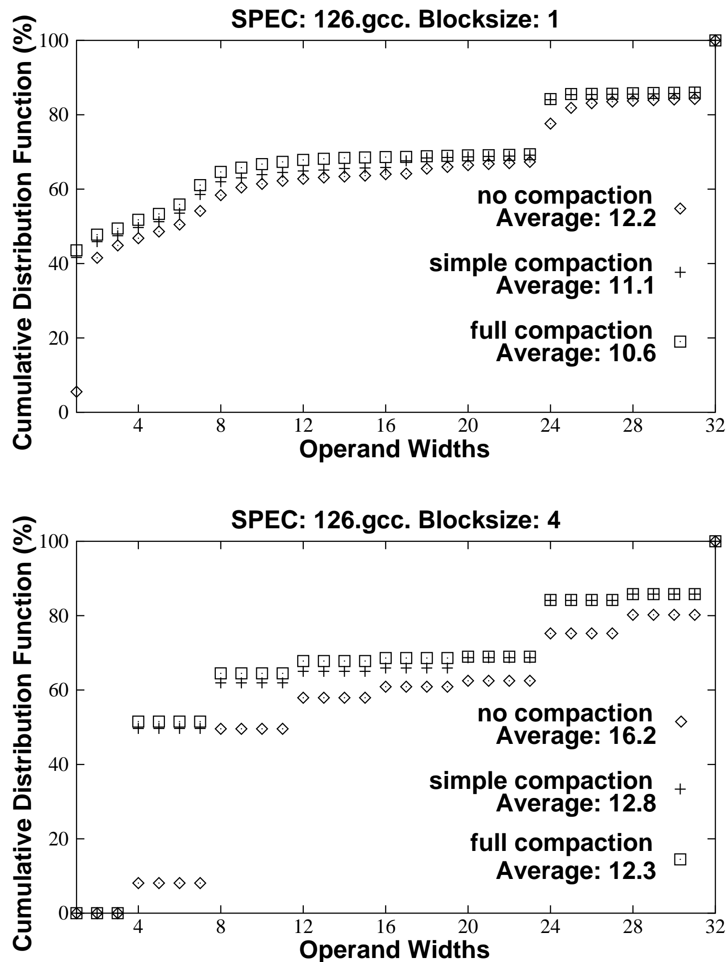


Figure 8. Distribution of operand widths for the SPEC95 benchmark `gcc`.

one-step compaction. Many other hybrid datapath design choices were also simulated for average operand width. Three of these design points are also shown in Table 2. The n/f $4 \times 8 - bit$ datapath corresponds to a datapath constructed using 4 8-bit WAD numbers, where the WAD numbers use full compaction and a block size of n ($n = 1, 2$). The last entry in the table corresponds to a datapath containing 4 radix-256 WAD numbers. In this representation, when an individual byte is all zeros or all ones, the byte is fully compacted (width 1).

The table shows that one-step compaction is a reasonable design point, especially when the block size of the WAD number increases. Also, observe that the difference between the average operand widths for different block sizes is close to half the block size, as might be expected. Using multiple WAD numbers does not always improve the operand width, because the savings resulting from compressing runs of 0's and 1's in the middle of an operand are offset by the overhead of having to always transmit information for each WAD number. The table also shows that using numbers of higher radix might be advantageous even though such representations typically require more complex VLSI implementations.

The table shows that, on average, a WAD representation using one-step compaction would activate between 29% and 57% of the datapath, depending on the application and type of WAD representation used. This is a significant improvement over a traditional datapath where each datapath

cell is activated for every operation.

6.2: Circuit Simulation

The physical design of the circuits we have described in this paper was done using the `magic` VLSI layout editor. The extracted layout was simulated with `aspice`, a mixed analog-digital simulator. All reported results are for quasi delay-insensitive circuits. We compare WAD circuits to normal bit-skewed designs as well as the finely pipelined aligned datapath designs used by the MiniMIPS asynchronous processor [16].

WAD datapaths have different energy and throughput characteristics compared to traditional datapath designs. We compare the circuits on $E\tau^2$, where E is the energy per operation and τ is the cycle time of the circuit. We use this metric because it is independent of voltage to first order, and therefore can be used to combine both energy and throughput into a single metric [16]. We designed datapath circuits using three design styles: WAD, traditional bit-skewed, and finely pipelined circuits like those used by the MiniMIPS. In each case we designed the circuits to minimize $E\tau^2$ in order to make a fair comparison. The spice results show that WAD datapath circuits have $E\tau^2$ that is between 40% and 70% the $E\tau^2$ of competing designs. This means that, e.g., for a fixed τ, the energy of a WAD implementation can be as low as 40% of the energy of other circuit styles. Our results did not model RC delays on wires; block-skewed WAD datapath circuits are superior to their traditional (both pipelined and unpipelined) counterparts in this regard, because they do not have vertical control distribution wires that traverse the height of the datapath.

To summarize, our preliminary circuit and architectural simulation results show that operand-based adaptivity using a WAD representation more than compensates for any circuit overhead introduced by adopting a WAD design style.

7: Related Work

It is easy to envison a datapath that can be partitioned into chunks having narrower width. Existing architectures that have such support include the Intel MMX extensions, Sun's VIS extensions, MicroUnity's MediaProcessor, Hewlett-Packard's MAX-2, and the PowerPC Altivec extensions. What differentiates a WAD datapath is the use of dynamic partitioning while the computation is being performed. There is no software support necessary for exploiting a WAD datapath, and the amount of datapath resources used by a computation is dynamically determined by the number of significant digits necessary to perform the computation. Brooks and Martonosi present a technique for 64-bit clocked processors that uses clock gating based on the value of operands to "turn off" the top 48 bits of the datapath [1]. Their approach, however, uses a traditional number representation and has several drawbacks. They use a zero detect circuit every time the integer unit is used. This circuit consumes $\mathcal{O}(N)$ energy and takes $\mathcal{O}(\log N)$ time (where N is the number of bits checked by the zero detect). They have to widen the result MUX for the integer function unit. The impact of introducing operand-based clock gating on the cycle time of the clocked processor is not evaluated—their proposal requires a 48-bit zero-detect to complete before the top 48 bits of the integer unit can be clocked. The impact of this structure on clock skew and jitter is also not evaluated. Spice simulations we have done show that such a comparator would take about 25% of the cycle time of an aggressively designed clocked processor. This zero-detect delay would increase for wider datapath structures, exacerbating the problem. As opposed to this, the number representations we have introduced naturally carry width information, and using an asynchronous implementation

automatically gives us fine-grained operand-based power management. Our representation is symmetric and provides lower power consumption for small positive as well as negative numbers, and scales to arbitrary width datapaths without modification.

Sign-magnitude numbers have been used for low power implementations for both clocked as well as bundled-data asynchronous circuits. This representation is attractive because the high order bits stay zero when small positive or negative numbers are used. However, when a dual-rail representation is used for performance reasons, the sign-magnitude representation does not save power. As opposed to this, WAD datapaths do not activate high order datapath cells when small numbers are used.

8: Summary

In this paper we introduced a new class of number representations for datapath design. The representations use a variable number of digits to represent integers based on the number of significant digits used by the computation. These representations are suitable for asynchronous implementation methods, and we provided several candidate implementations for both the finely pipelined and unpipelined datapath design points. We reported preliminary architectural and circuit simulations that demonstrate that WAD designs can provide a significant reduction in energy consumption for typical operand widths in a number of applications.

Acknowledgments

This work 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.

A: Notation

The CHP notation we use is based on Hoare's CSP [3]. A full description CHP and its semantics can be found in [13]. What follows is a short and informal description.

- Assignment: $a := b$. This statement means "assign the value of b to a." We also write $a\uparrow$ for $a := true$, and $a\downarrow$ for $a := false$.
- Selection: $[G1 \rightarrow S1 \; [\!] \; ... \; [\!] \; Gn \rightarrow Sn]$, where Gi's are boolean expressions (guards) and Si's are program parts. The execution of this command corresponds to waiting until one of the guards is $true$, and then executing one of the statements with a $true$ guard. The notation $[G]$ is short-hand for $[G \rightarrow skip]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
- Repetition: $*[G1 \rightarrow S1 \; [\!] \; ... \; [\!] \; Gn \rightarrow Sn]$. The execution of this command corresponds to choosing one of the $true$ guards and executing the corresponding statement, repeating this until all guards evaluate to $false$. The notation $*[S]$ is short-hand for $*[true \rightarrow S]$.
- Send: $X!e$ means send the value of e over channel X.
- Receive: $Y?v$ means receive a value over channel Y and store it in variable v.
- Probe: The boolean expression $\overline{X}$ is $true$ iff a communication over channel X can complete without suspending.

- Sequential Composition: $S; T$
- Parallel Composition: $S \parallel T$ or S, T.
- Simultaneous Composition: $S \bullet T$ both S and T are communication actions and they complete simultaneously.

References

[1] David Brooks and Margaret Martonosi. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. HPCA-5, January, 1999.

[2] A.W. Burks, H.H. Goldstein, and John von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Institute for Advanced Study, Princeton, N.J. June 1946. Also available in A.H. Taub, Ed., *Collected works of John von Neumann*, **5**:34–79, Macmillan, New York 1963.

[3] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, **21**(8):666–677, 1978

[4] Kai Hwang. *Computer Arithmetic.* John Wiley & Sons, 1979.

[5] Donald Ervin Knuth. *Seminumerical Algorithms.* The Art of Computer Programming, Vol. 2. Addison-Wesley Publishing Company, Reading, Massachusetts, 1981.

[6] Andrew Matthew Lines. *Pipelined Asynchronous Circuits*. M.S. thesis, Caltech Computer Science, 1995.

[7] Rajit Manohar. *The Impact of Asynchrony on Computer Architecture.* Ph.D. thesis, Caltech Computer Science Technical Report CS-TR-98-12, July 1998.

[8] Rajit Manohar. *The Design and Implementation of Width-Adaptive Datapaths.* Cornell Computer Systems Laboratory Technical Report CSL-TR-2000-1005, September 2000.

[9] Rajit Manohar, Tak-Kwan Lee, and Alain J. Martin. Projection: A Synthesis Technique for Concurrent Systems. *Proceedings of the Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1999.

[10] Rajit Manohar and Alain J. Martin. Slack Elasticity in Concurrent Computing. *Proceedings of the Fourth International Conference on the Mathematics of Program Construction*, Lecture Notes in Computer Science 1422, pp. 272–285, Springer-Verlag, June 1998.

[11] Rajit Manohar and José Tierno. Asynchronous Parallel Prefix Computation. *IEEE Transactions on Computers* **47**(11):1244–1252, November 1998.

[12] Alain J. Martin. Formal Program Transformations for VLSI Circuit Synthesis. In E.W. Dijkstra, editor, *Formal Development of Programs and Proofs*, UT Year of Programming Series, pp. 59–80, Addison Wesley, 1989.

[13] Alain J. Martin. Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4), 1986.

[14] Alain J. Martin. Asynchronous Datapaths and the Design of an Asynchronous Adder. *Formal Methods in System Design*, **1**:117–137, 1992.

[15] Alain J. Martin, Steven M. Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pp. 351–373, MIT Press, 1991.

[16] A.J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. Cummings, and T.K. Lee. The Design of an Asynchronous MIPS R3000 Processor. *Proceedings of the 17th Conference on Advanced Research in VLSI.* Los Alamitos, Calif.: IEEE Computer Society Press, 1997.

[17] O.L. MacSorley. High-Speed Arithmetic on Binary Computers. *Proceedings of the IRE*, **49**(1):67–91. 1961.

[18] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, **3**:1–8, 1988

[19] Ted Eugene Williams. *Self-timed rings and their application to division.* PhD Thesis. Computer Systems Laboratory, Stanford University, May 1991.

[20] S. Winograd. On the Time Required to Perform Addition. *Journal of the Association for Computing Machinery* **12**(2):277-285, April 1965.