THEORETICAL UNDERPINNINGS

André DeHon
Department of Electrical and Systems Engineering
University of Pennsylvania

Throughout this book there are examples for which reconfigurable designs offer superior performance to processor-based solutions. The reconfigurable implementation is typically orders of magnitude faster than the processor-based system. Even when we normalize the performance advantage to the number of components used in the solution, or to the number of square millimeters of silicon in the same process technology, we often see the reconfigurable solution providing one to two orders of magnitude higher computational capacity per square millimeter. These observations raise questions about reconfigurable computing systems.

- Why do we see this greater computational capacity per unit area?
- How can we predict when reconfigurable systems can deliver significantly higher performance than processor-based implementations?
- What does this tell us about how we should engineer reconfigurable designs?

This computational density advantage is not an accident. It occurs for real, structural reasons resulting from where silicon is allocated in reconfigurable architectures. Field-programmable gate arrays (FPGAs) and reconfigurable architectures organize their instructions differently from processors, making different trade-offs between instruction and computational density. Processors give up raw computational capacity for the ability to support large and irregular computations robustly, while FPGAs give up the ability to switch rapidly among diverse tasks to maximize available compute density and spatial parallelism. This chapter develops a simple model of programmable devices and uses it to illustrate the gross design space, which includes processors and FPGAs, the trade-offs each makes, and the consequences of those trade-offs.

36.1 GENERAL COMPUTATIONAL ARRAY MODEL

Let us start by focusing exclusively on a capabilities viewpoint, ignoring, for the moment, costs. *What would be good to have for a general-purpose programmable computing architecture?*

Copyright © 2008 by André DeHon. Published by Elsevier Inc.

The most general and flexible programmable architecture we might build would have:

- Computational operators (e.g., programmable gates) that compute an output bit from some number of input bits
- Full, bit-level interconnect among computational operators
- Local data storage for each bit operator
- The ability to issue a unique instruction to each bit-level computational operator on every cycle; this instruction should indicate:
  - Which computational function the operator should perform on each cycle
  - Where the inputs for the operator should come from, including both spatially from any other operator and temporally from local memory
  - Where the output of the operator on this cycle should go into local memory

Figure 36.1 shows a diagram of this architecture. For this simple model, we assume that all the programmable blocks are identical. We call the instruction that controls each programmable block (including interconnect and memory, as just summarized) a *primitive instruction*, or *pinst* for short (see Figure 36.2). With an array of *N* blocks, the full instruction word issued on every cycle to control the computational array is the composition of *N* pinsts.

This array provides a computational capacity of *N*-bit operations (*bitops*) on each cycle. We have great flexibility in using this array since every bitop can have a unique pinst on every cycle. So, if we need to process an irregular collection of operations, such as a 17-bit add, an 8-bit subtract, a 13-bit exclusive-or (XOR), the next state evaluation on a 23-state finite-state machine (FSM), and a 5-bit shift left by 3, we can direct each bitop independently to keep all bitops performing exactly the operations needed for the computation. Further, if the following cycle needs a very different set of operations, such as a 9-bit multiply by the constant 27, a 12-bit AND, the next state evaluation on a 23-state FSM,

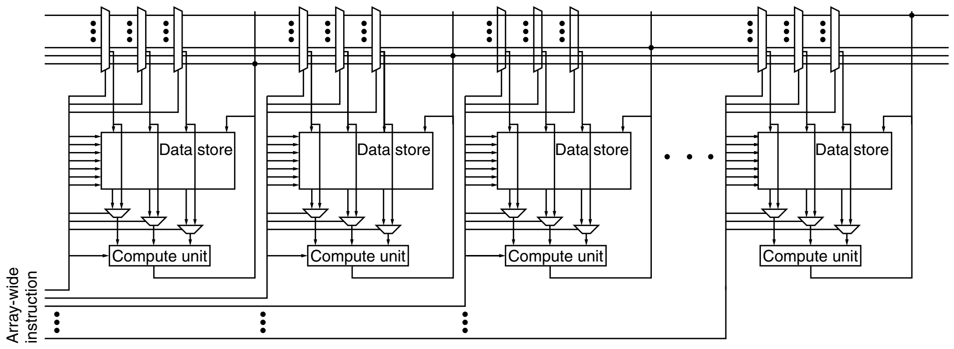

FIGURE 36.1 ■ The general computational array model.

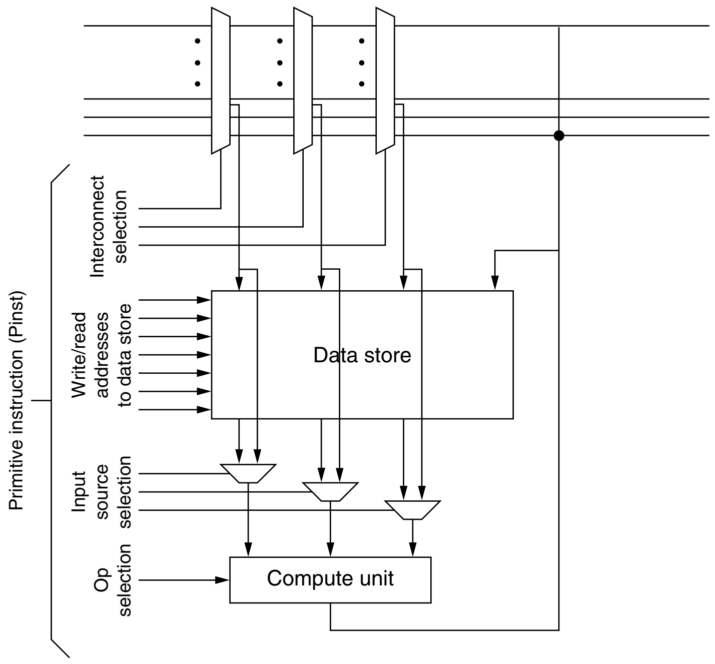

FIGURE 36.2 ■ Primitive instruction (pinst) for programmable bitops.

and an 11-bit shift right by 2, we can issue the next array-wide instruction to control the computational array accordingly.

We get to use all the bitops all the time. Mapping designs to this array is simply a matter of scheduling the bit-level computational needs onto the *N*-bit operations provided by the array. With this full ability to control the cycle-by-cycle operation of each bitop independently, scheduling is relatively easy. (Strictly speaking, optimal scheduling remains NP-hard, but it can be approximated within a factor of 2 of optimal using a variant of Johnson's Algorithm [1].) So, why is it that we do not have a popular architecture that provides this model?

36.2 IMPLICATIONS OF THE GENERAL MODEL

From a purely logical standpoint, we cannot fault the general computational array model. However, we must implement any architecture in a physical computational medium (e.g., out of a number of discrete vacuum tubes or transistors, on a silicon die, ultimately out of molecules and atoms). To support the architecture, we must commit physical resources. Those resources have a cost in terms of area, delay, and energy. The general computational array model turns out to be extravagant—so much so that we are generally willing to compromise its power to build more practical architectures.

This section illustrates two ways in which the instruction organization of the general model is unreasonably expensive. The focus here is on silicon VLSI implementations, and we discuss the sizes and areas of components in VLSI. To make the discussion general, resource areas are measured in terms of technology-normalized units. In particular, we will measure widths in units of F—the minimum feature size in a VLSI process; as a consequence, areas are measured in units of F^2. VLSI technologies are normally named by their minimum feature size, so when we talk about a 45 nm technology, we are talking about a technology with $F = 45$ nm. Ideally, when we scale from a larger technology to a smaller technology, everything scales as F. Features 900 nm wide in a 90 nm technology are $10F$ wide and should become 450 nm wide in a 45 nm technology. Features do not always scale perfectly linearly like this, but they scale close enough for illustrative purposes. Details and estimates on how the industry expects silicon technology to scale are summarized by the ITRS [2]; the industry collaborates to produce an updated or revised version of this document annually.

36.2.1 Instruction Distribution

This section starts by considering the resource implications of delivering a separate pinst to every bitop. We assume the following:

- The bitops are arranged in a dense $\sqrt{N} \times \sqrt{N}$ array (see Figure 36.3).
- The area required for each bitop, including compute, storage, and interconnect, is $A_{bop} = 250{,}000F^2$; we further assume that the bit operator itself is laid out as a square $500F$ on a side. This size assumes that the interconnect has also been designed in a more restrictive way than the most general model (see Section 36.1), perhaps resembling something closer to traditional FPGA interconnect capabilities.
- The metal pitch available for distributing an instruction bit is $W_{metal} = 4F$. The minimum pitch possible in a given technology is $2F$ because we need to leave one feature size worth of space between features so that they do not short together. The smallest feature sizes tend to be polysilicon for transistor gate widths, with metal pitches being a little wider. A modern VLSI process has many metal layers, and the ones higher in the stack (farther from the silicon base) tend to be wider.
- We have one complete horizontal metal layer and one complete vertical metal layer available to distribute instructions. As noted, modern VLSI processes generally have many metal layers; for example, an $F = 65$ nm process might have 11 metal layers. Some of the layers will be needed for local wiring in the cell, some for power and clock distribution, and some for interconnect. Dedicating two complete metal layers to instruction distribution is extravagant even with 11 metal layers.
- Each pinst requires $I_{bits} = 64$ to specify its instruction. This may seem small if we think about how many bits are required per 4-LUT in an FPGA, or large if you think about 32-bit processor instructions. Encoded densely, FPGA configurations could be much smaller [3]. The capabilities of the pinst might be closer to two processor instructions than one.

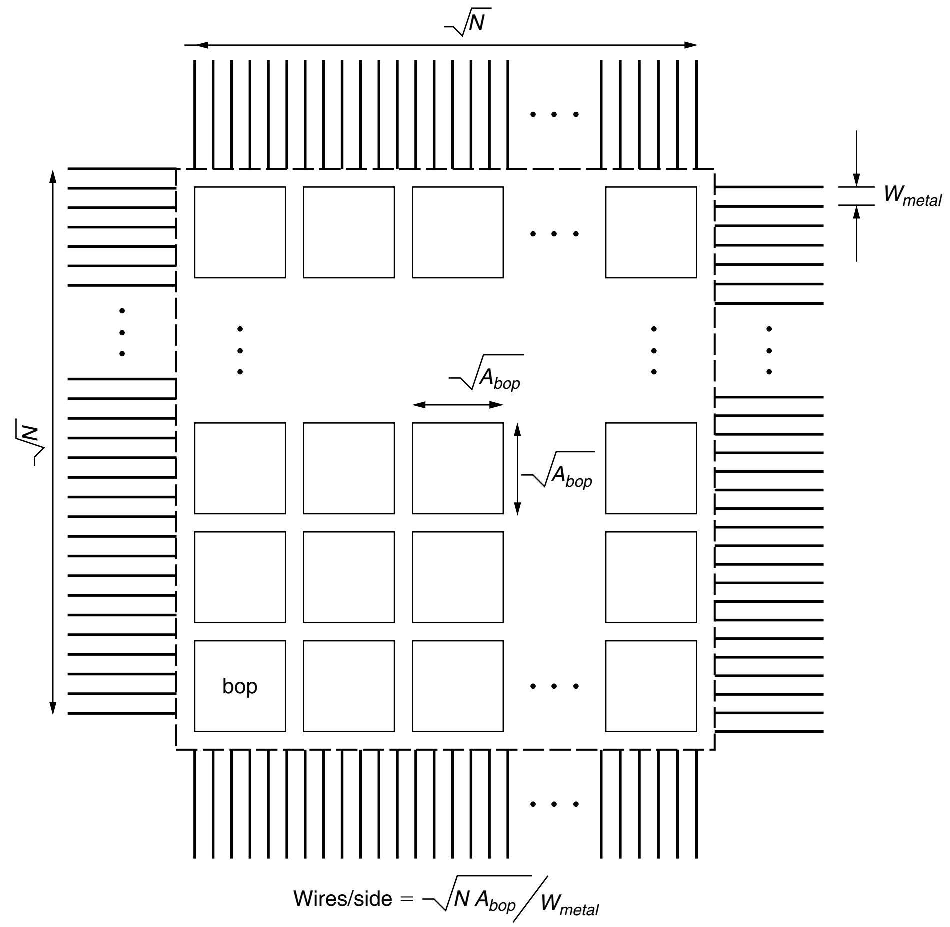

FIGURE 36.3 ■ Wiring for instruction distribution.

As we will see, the preceding assumptions only affect the particular quantitative conclusion we reach. The qualitative effect remains even if we assume two or four times as many metal layers, half the metal pitch, more compact instruction encodings, or larger bitop cell sizes.

If the instructions must all come into the computational array, then the total wiring capacity available for instruction distribution is equal to the perimeter of the array.

$$A_{side}(N) = \sqrt{N} \times \sqrt{A_{bop}} \tag{36.1}$$

$$L_{perimeter}(N) = 4 \times A_{side}(N) \tag{36.2}$$

Note that the two metal layers allow the connections on the top and bottom layers to cross over each other to reach into the array. However, if the lower layer is completely dense, we will have trouble making connections between the upper layer and the bit operations (i.e., we need to reserve space for vias through the lower level). To keep the math simple, general, and illustrative, we will not model that effect, which will only tend to make the problem more severe than the simple model indicates.

To feed the N-bit operators into the array, we need:

$$I_{total\_bits}(N) = N \times I_{bits} \tag{36.3}$$

$$L_{instr\_dist}(N) = W_{metal} \times I_{total\_bits}(N) \tag{36.4}$$

For the distribution to be viable, we need:

$$L_{perimeter}(N) > L_{instr\_dist}(N) \tag{36.5}$$

Substituting into the previous equations, this results in:

$$4 \times \sqrt{N} \times \sqrt{A_{bop}} > W_{metal} \times N \times I_{bits} \tag{36.6}$$

$$\frac{4 \times \sqrt{A_{bop}}}{W_{metal} \times I_{bits}} > \sqrt{N} \tag{36.7}$$

$$N < \left(\frac{4 \times \sqrt{A_{bop}}}{W_{metal} \times I_{bits}}\right)^2 \tag{36.8}$$

Using the preceding assumptions:

$$N < \left(\frac{4 \times 500F}{4F \times 64}\right)^2 = 61 \tag{36.9}$$

This says that we cannot afford to feed more than about 60 bit-processing units without saturating available instruction distribution bandwidth. If we want to support more bit-processing elements, we must increase the perimeter and effectively make the bitops larger. Rearranging equation 36.6 with A_{bop} as the variable:

$$\sqrt{A_{bop}(N)} > \frac{W_{metal} \times \sqrt{N} \times I_{bits}}{4} \tag{36.10}$$

$$A_{bop}(N) = \left(\frac{W_{metal} \times \sqrt{N} \times I_{bits}}{4}\right)^2 \tag{36.11}$$

$$A_{bop}(N) = 4096 \times NF^2 \tag{36.12}$$

That is, the area of each bitop needs to grow linearly with N, meaning that the array area is actually growing quadratically with N.

Equivalently, we can recognize this effect as a difference between the growth rate of the area and the perimeter. If we assume the bitop area is constant, then the total area in the array is growing linearly in the number of bitops. However, the perimeter of the array is only growing as the square root of the array area. So it is not surprising that we reach a point where the array's need for instructions, which is also growing linearly with bitops, exceeds the ability to feed instructions into the array that grows only as the square root of the number of bitops in it. The particular assumptions used for this example starkly illustrate that this effect is already an issue for very small arrays. You can substitute your favorite assumptions about instruction bits, metal pitch, metal layers, or bit-operator area, but the qualitative conclusion remains as follows:

If we support this model, either we are limited in the size of the arrays we can build, or instruction distribution wiring ends up dominating all other resources and forces us to scale only as the square root of the area we spend on the computational array.

36.2.2 Instruction Storage

The previous section illustrated that instruction distribution from outside the computational array is not scalable to large computations. Alternately, consider storing the instructions inside the array. In particular, each bitop could include an instruction memory that holds its instruction (see Figure 36.4). We would

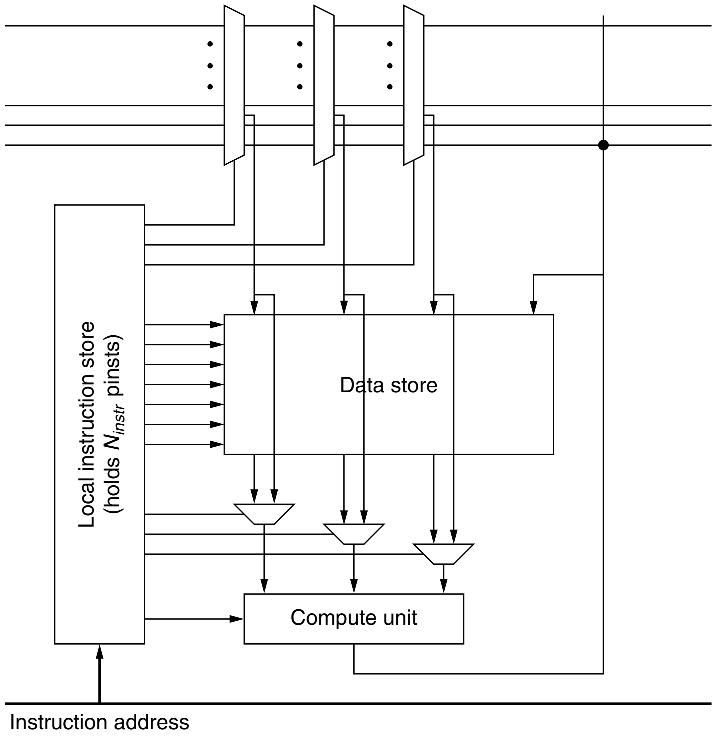

FIGURE 36.4 ■ A bitop with local instruction memory.

then only need to broadcast an address into the array, and each bitop could translate that through the instruction memory to its instruction. Even a 64-bit address is small compared to $L_{perimeter}(1)$, so this solution does not challenge wiring capacity. However, it does raise the question of how large the instruction memory should be to begin to approximate the general model.

In any case, storing the instructions requires area. So we should assess the cost of storing these instructions. Assume that the instruction memory lives in SRAM, and that the area of an SRAM cell to hold an instruction bit is $A_{bit} = 200F^2$. This means that the area per instruction is:

$$A_{pinst} = A_{bit} \times I_{bits} \tag{36.13}$$

$$A_{pinst} = 200F^2 \times 64 = 12{,}800F^2 \tag{36.14}$$

The total area per bitop is now:

$$A_{bitop\_w\_imem} = A_{bop} + N_{instrs} \times A_{pinst} \tag{36.15}$$

$$A_{bitop\_w\_imem} = 250{,}000F^2 + N_{instrs} \times 12{,}800F^2 \tag{36.16}$$

Equation 36.16 now tells a very interesting story. The area required to store a single instruction is small compared to the area required for compute and interconnect in the bit operator (one-twentieth the area). If we store 20 instructions locally, we place half of the area into instruction memory. When we store 200 instructions locally, the instruction memory area ends up dominating (i.e., is 10 times the size of) the area required for computation. That is, given fixed area, the design with 200 instructions will only fit one-tenth the number of bitops as the design with a single local instruction.

Unless we can limit the number of different, array-wide instructions we need to issue, the instruction memory needed to approximate the general model will end up dominating the computational area. Taken together with the result on instruction distribution, these examples illustrate why the general model is not typically supported:

To support the general model, instruction resources would dominate all other resources, forcing limited computational density.

We are left with the choice of either accepting very low computational density or looking for compromises in the general model that will allow us to avoid the huge instruction expense it implies.

36.3 INDUCED ARCHITECTURAL MODELS

If the general model was viable, we would not have the varied set of computer architectures that exist. That is, computer architectures arise because (1) the general model is too expensive, and (2) there is structure in typical computational tasks that permits more economical implementations. Having identified that it is unreasonable to support the general computational array model, we ask: Which structure exists in typical computations that can be exploited to provide a more economical implementation?

36.3.1 Fixed Instructions (FPGA)

If the instructions never change, we do not need to distribute them into the computational array, nor do we need to allocate instruction memory area to store more than a single instruction. We still allow each bitop a pinst, so each can perform a unique operation; however, we do not allow the pinst to change from cycle to cycle. Unchanging instructions is an extreme form of temporal locality, where computation remains the same over time. This allows us to build large arrays and keep the computation dense. If we need to, or can arrange to, perform the same computation on every cycle, then we use the array efficiently. This restriction on the general model effectively gives us an FPGA or spatially reconfigurable architecture. In Chapter 5, Section 5.2, we saw many system architectures that illustrate how we might organize computation to enhance this kind of structure.

36.3.2 Shared Instructions (SIMD Processors)

Another structure common to applications is SIMD datapaths (see Single program, multiple data subsection of Section 5.2.4)—that is, it is common for us to identify sequences of bit-level operations that are the same across a number of data bits. The most common case is word-wide operations, such as multibit adds or bitwise logical operations (e.g., OR, AND, XOR). At a higher level, we would perform a number of identical word-wide operations on different data (e.g., performing a component-wise multiplication on the elements of two arrays as part of a dot product). Here we perform the same operation across many bitops. Rather than providing a unique instruction for each bitop, we can arrange to share a single instruction across a large number of bit operators, amortizing the instruction distribution or storage expense.

In the extreme, we would distribute a single instruction to all the bitops in the array. This is the opposite of the simplification used in the FPGA. Here, all bitops in the array must perform the same operation on a given cycle, but this operation may change from cycle to cycle.

We can view conventional, word-wide processors as exploiting this idea. A processor instruction typically only tells the datapath to do one homogeneous thing—that is, the processor instruction asks every bit in the arithmetic logic unit (ALU) bit slice to perform the same computation (e.g., perform a full adder bit, perform an XOR, perform a shift). For example, a 32-bit processor datapath could perform many more operations if each individual bit slice of the ALU could operate independently; instead, ALUs are constrained to operate in SIMD fashion to keep the cycle-by-cycle instruction size small.

In the general computational array model, we saw that the instruction memory took up the same area as the computation when we stored only 20 instructions in the array (equation 36.16). If we instead share each instruction across

$W_{simd} = 32$ bitops to form a SIMD datapath, it takes 625 instructions for the instruction memory to reach parity with the computation—that is:

$$A_{bitop\_w\_imem}(W_{simd}, N_{instrs}) = A_{bop} + \left(\frac{N_{instrs}}{W_{simd}}\right) \times A_{pinst} \tag{36.17}$$

$$A_{bitop\_w\_imem}(N_{instrs}, 32) = 250,000F^2 + N_{instrs} \times 400F^2 \tag{36.18}$$

From these illustrations, we can see how the more familiar FPGA and processor architectures fall out as simplifications of the general computational array model that exploits different kinds of structures that exist in typical computations.

36.4 MODELING ARCHITECTURAL SPACE

The demonstrations in Sections 36.2 and 36.3 highlight the fact that choices about instruction architecture can have a first-order impact on the area, and hence density, of programmable computing components. We can take this a step farther and build models of the density, and ultimately relative efficiency, of architectural design points.

Table 36.1 summarizes where some familiar architectures fall in the (W_{simd}, N_{isntr}) architectural space. Nonetheless, remember that we are using a deliberately simple model and that many other effects and issues are associated with each architecture, some of which are mentioned in Section 36.4.3.

36.4.1 Raw Density from Architecture

Using equation 36.17, we can plot the relative densities of each bit operator as a function of the local instruction memory, N_{instr}, and the SIMD instruction width, W_{simd}. Figure 36.5 shows plots of the computational density for the instruction memory from 1 to 16,384 and the instruction width from 1 to 1024. Here, note that peak densities vary over three orders of magnitude. As we increase instruction depth (N_{instr}), we shift area into instructions rather than compute, often significantly reducing computational density. Wide-word architectures can reduce the memory costs at a particular instruction depth, but there also may be significant computational density reductions as instruction depth grows.

TABLE 36.1 ▪ Placement of sample architectures in (W_{simd}, N_{instr}) space

Architecture	W_{simd}	N_{instr}	Reference
FPGA	1	1	
GARP fabric	2	4	Chapter 2, Section 2.1.1
KiloCore256	8	16	Chapter 2, Section 2.1.2
MIPS-X	32	512	[4]
IA-64 (Montecito)	64	200,000	[5]
Cell SPU	128	65,536	[6]

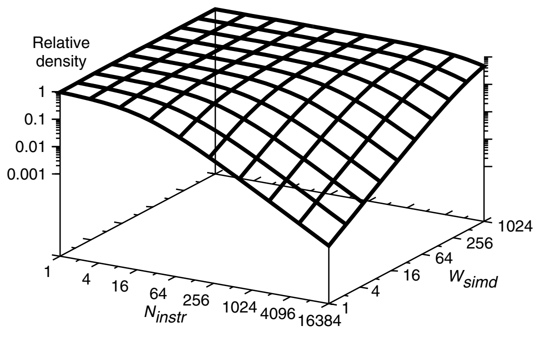

FIGURE 36.5 ■ Relative peak computational density from the model (normalized to the density of $N_{instr} = 1$, $W_{simd} = 1024$ design points).

36.4.2 Efficiency

The previous section showed peak raw densities achievable at various architectural points. If peak raw density was all that mattered, we would build SIMD designs with shallow instruction memories, as Figure 36.5 illustrates. However, it is seldom the case today that we can keep the millions of SIMD bit-processing elements we might be able to put on a die performing useful computations. When we cannot match the structure assumed by the architecture, the yield is only a fraction of the potential density—that is, another architecture, perhaps one with lower peak density, often can deliver more net density to the application. In particular, the architectural point whose structure assumptions exactly match the application will deliver the highest net density on that application. This leads to an interesting set of questions:

- How does the efficiency of an architecture fall off as it becomes mismatched to the structure of the application?
- How does the net density compare between various matched and mismatched architectures?

Since there is a model for the area of architectural design points in the (W_{simd}, N_{instr}) design space (equation 36.17), we can use that to measure efficiency. In particular, it is possible to measure the efficiency of an architecture design point (Arch(W_{simd}, N_{instr})) processing applications with a particular structure (App(W_{app}, L_{path})) as the ratio of the area of the architecture that exactly matches the application structure to the area of the point being evaluated:

$$\text{Efficiency}\left(\text{Arch}(W_{simd}, N_{instr}), \text{App}\left(W_{app}, L_{path}\right)\right) = \frac{Area\left(\text{Arch}\left(W_{app}, L_{path}\right), \text{App}\left(W_{app}, L_{path}\right)\right)}{Area\left(\text{Arch}(W_{simd}, N_{instr}), \text{App}\left(W_{app}, L_{path}\right)\right)} \quad (36.19)$$

TABLE 36.2 ■ Sample applications in the (W_{app}, L_{path}) space

Application	W_{app}	$L_{critpath}$	L_{path}	Comments
Conway's Game of "Life"	1	1	1	Bit-level CA [7]
Error correcting codes	1	1	1–10,000	At memory interface, need one per cycle; on audio-rate, real-time data can be low throughput
Entropy coding	1	1–10	1–10,000	(similar to previous)
Video processing of pixel data	8	1–6	12	1024×1024 at 30 frames per second on a 500 MHz cycle can afford approximately 12 cycles per pixel
CD audio	16	1–10	10,000	44 kHz real-time vs. 500 MHz cycle
SPIHT image compression	16	10	10+	Chapter 27
FDTD	35	1–5	1–5	Chapter 32

To characterize the structure of the architecture separately from the structure of the application, equation 36.19 keeps W_{simd} and N_{instr} as parameters characterizing the architecture and adds the dual parameters W_{app} and L_{path} to characterize the application structure. W_{app} is simply the natural SIMD datapath width of the application, while L_{path} is the path length of the application (see the Mismatch in N_{instr} subsection).

For illustrative purposes, Table 36.2 summarizes where several applications appear in the (W_{app}, L_{path}) space. The area of the mismatched design is always larger, so the efficiency metric in equation 36.19 effectively tells us how much lower the mismatched point's net density is than the matched point's net density.

To develop the intuition and keep the explanation simple, we stay with the assumption that applications have homogeneous structure (i.e., single-characteristic W_{app} and L_{path}). One of the reasons we are interested in how well an architecture deals with different, mismatched structures is that a real application will typically contain heterogeneity in the structure it exhibits.

Mismatch in W_{simd}

What happens when the application width W_{app} is mismatched to the architectural width W_{simd}?

- $W_{simd} > W_{app}$: Here we do not have as fine-grained control of the bit operators as the application requires. Consequently, bitops go unused. In particular, we will actually need a larger array so that we match the instruction control needs of the application. For example, if $W_{app} = 5$ and $W_{simd} = 8$, then three bitops in every architectural SIMD datapath will go idle. To satisfy the application requirements, we end up needing $\frac{W_{simd}}{W_{app}} = \frac{8}{5} = 1.6$ times as many physical bitops as the application actually requires.

- $W_{simd} < W_{app}$: There are two effects that can work to make implementations in this architecture larger than the optimally matched architecture:
  1. We have finer-grained control, but may still need more physical bit operators because of granularity problems. For example, when $W_{app} = 8$ and $W_{simd} = 5$, we need $\left\lceil \frac{W_{app}}{W_{simd}} \right\rceil = 2$ groups of W_{simd} bitops to cover each application group, or $\left\lceil \frac{8}{5} \right\rceil \times W_{arch} = 10$ bitops, of which only $W_{app} = 8$ are doing useful work.
  2. Since we have more control than necessary for the application, the area of each bitop is larger than necessary in order to accommodate additional instruction memory; this extra instruction memory holds redundant information. Continuing the $W_{app} = 8$ and $W_{simd} = 5$ example, each bit operator effectively pays for $\frac{W_{app}}{W_{simd}} = \frac{8}{5} = 1.6$ times as many instructions as necessary for the application.

Assuming that instruction storage depth is matched to application path length ($N_{instr} = L_{path}$) to focus on the width mismatch, we can show this in an area model as:

$$\begin{aligned} &\mathit{Area}\left(\text{Arch}\left(W_{simd}, L_{path}\right), \text{ App}\left(W_{app}, L_{path}\right)\right) \\ &\quad = \left(\frac{W_{simd}}{W_{app}}\right) \times \left\lceil \frac{W_{app}}{W_{simd}} \right\rceil \times A_{bitop\_w\_imem}\left(W_{simd}, L_{path}\right) \\ &\quad = \left(\frac{W_{simd}}{W_{app}}\right) \times \left\lceil \frac{W_{app}}{W_{simd}} \right\rceil \times \left(A_{bop} + \left(\frac{L_{path}}{W_{simd}}\right) \times A_{pinst}\right) \end{aligned} \tag{36.20}$$

This allows us to compute the efficiency of the mismatched SIMD datapath width at a matched L_{path} as:

$$\begin{aligned} &\text{Efficiency } [L_{path}]\left(W_{simd}, W_{app}\right) \\ &\quad = \frac{\left(A_{bop} + \left(\frac{L_{path}}{W_{app}}\right) \times A_{pinst}\right)}{\left(\frac{W_{simd}}{W_{app}}\right) \times \left\lceil \frac{W_{app}}{W_{simd}} \right\rceil \times \left(A_{bop} + \left(\frac{L_{path}}{W_{simd}}\right) \times A_{pinst}\right)} \end{aligned} \tag{36.21}$$

Figure 36.6 shows plots of the efficiency from equation 36.21 versus W_{app} for a collection of W_{simd}'s and L_{path}'s. Perhaps more significant than the large density range shown in Figure 36.5, we see that SIMD width mismatches can cost us orders of magnitude in net density delivered to an application. Interestingly, we see some SIMD width selections that do not show orders of magnitude efficiency losses (e.g., $W_{simd} = 1$ for $L_{path} = 1$, $W_{simd} = 3$ for $L_{path} = 64$, $W_{simd} = 32$ for $L_{path} = 640$). These robust points occur when the instruction area is equal to the compute and interconnect area. That is:

$$A_{bop} = \left(\frac{L_{path}}{W_{simd}}\right) \times A_{pinst} \qquad (36.22)$$

In these cases, half the area is allocated to storing instructions and half to compute. For illustration, consider the $L_{path} = 640$ and $W_{simd} = 32$ case. Here, if we are processing $W_{app} = 1$ data, then we use only one-thirty-second of the compute

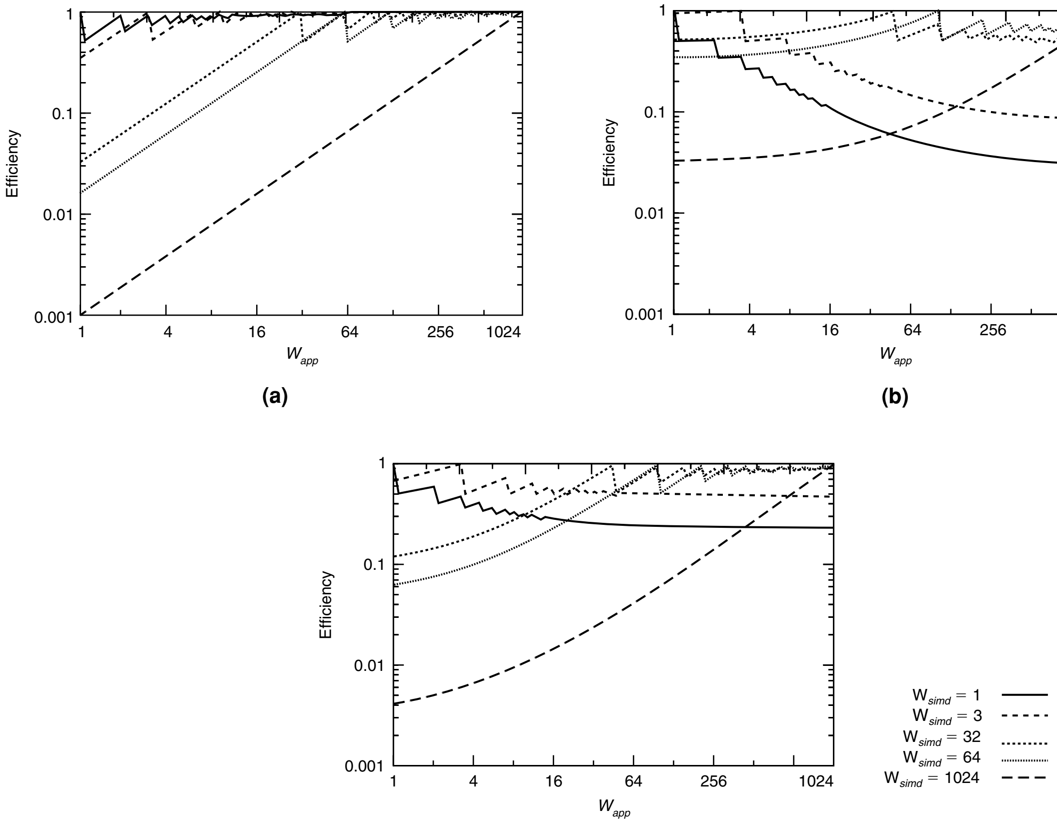

FIGURE 36.6 ■ Efficiency as a function of W_{app} for various L_{path} values: (a) $L_{path} = 1$, (b) $L_{path} = 640$, and (c) $L_{path} = 64$.

area. However, we are able to use all the memory area; a matched architecture can, at most, be half the size of this design point since it still requires the 640 instructions, even if they drive a smaller datapath. At the opposite extreme, if $W_{app} = 16,384$, we can use all the compute operators but we underutilize the instructions. Here, a matched architecture could have used a factor of 512 lower instruction area; however, since half the area is in compute, the matched architecture is, at best, only half the size of this robust point.

It should be clear that this observation holds for any choice of W_{app} when the area is allocated evenly between compute and instruction memory. In contrast, if we make $W_{simd} = 1$ for this $L_{path} = 640$ case, then 97 percent of the area goes into memory; if this $W_{simd} = 1$ architecture now has a task with $W_{app} = 16,384$, it is much larger (at least 33 times larger) than a design with matched width, which can put significantly less area into instruction memory.

If we can design to a single application width, or a small range of widths, it is best to select a matched width, or the width that provides the highest average efficiency over the range. However, if we don't have tight bounds on the application width, these robust points show how we can select organizations that remain fairly efficient for any application width.

Mismatch in N_{instr}

A similar phenomenon occurs when N_{instr} does not match the structure of the application. First, we need to understand L_{path}—the application demand for N_{instr}. In particular, let us consider an inner loop in a kernel or the computation required for each invocation of a transform operator (see Transform or object subsection of Section 5.1.2). To compute each inner loop iteration, or each operator invocation, we need to evaluate a set of N_{ops} bitops. In general, there may be a set of cyclic sequential dependencies, or a critical path, of depth $L_{critpath}$ among the bitops in the computation that prevent us from starting the next iteration of the loop or invocation of the operator until the $L_{critpath}$ array cycles have completed. For example, consider the loop body of a saturated accumulation:

$$y[i] = \max(\min(x[i] + y[i-1], 255), 0)$$

Before performing the next addition to compute $y[i+1]$ from $y[i]$, we must complete the computation of $y[i]$, including both the addition and the selection of maximum or minimum bound limits (see Figure 36.7).[1] Assume the following:

- The addition requires a path length of six sequential bitops.
- The comparisons can be performed in parallel.
- Each comparison requires a path of three sequential bitops.
- The final selection requires a single bitop.

The critical path $L_{critpath}$ is 10 for this computation. With a path length of $L_{critpath}$, we can schedule the N_{ops} required to evaluate the application into $L_{critpath}$ cycles

[1] With care, this actually can be avoided using sophisticated transformations [8].

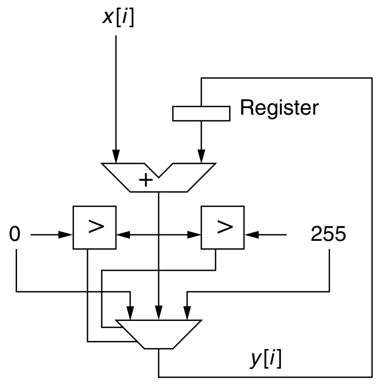

FIGURE 36.7 ■ Saturated accumulator cyclic dependency.

on the array without slowing down the application, the sequentially dependent paths guarantee that it will always take at least $L_{critpath}$ cycles to perform the operation.

The application may not actually demand that the computation be performed every $L_{critpath}$ cycle. Perhaps the data throughput is lower and new samples, $x[i]$, are arriving every 20 ns while the array cycle time is 1 ns. Here, evaluating with $L_{critpath} = 10$ leaves the array sitting idle for 10 cycles before the next input sample is available to compute. Consequently, it would be possible to schedule to $L_{path} = 20 > L_{critpath}$ and cut the number of bitops needed by at least a factor of 2. In this way, the loop or transform body is efficiently implemented by scheduling the computations onto a minimum number of bitops in a period of L_{path} cycles, with each operator potentially getting a unique instruction on each cycle $N_{instr} = L_{path}$. For examples, see Table 36.2, which summarizes the throughput L_{path} required in a few applications.

Now consider the two mismatched cases:

- $N_{instr} > L_{path}$: In this case, by scheduling the computation into L_{path} cycles, $(N_{instr} - L_{path})$ instruction memory slots in each bitop go unused. The matched architecture is smaller because it does not spend area on these unused instruction memories. In the aforementioned saturated accumulation, if $L_{path} = 20$ and an array with $N_{instr} = 100$ is used, then 80 instruction slots go unused.
- $N_{instr} < L_{path}$: In this case, we cannot necessarily reuse each bit operator in L_{path} in different ways on each of the L_{path} cycles. Since we can only use each operator in N_{instr} ways, to solve the entire problem we may need a total of $\left\lceil \frac{L_{path}}{N_{instr}} \right\rceil$ times as many bitops to perform the computation. Continuing with the example, if $N_{instr} = 5$ and there is an $L_{path} = 20$, we may need four times as many bitops as the optimally matched architecture. The total amount of memory is the same between these cases; however, an $N_{instr} = 5$ architecture pays for four times as many

compute blocks (A_{bop}). There is also a granularity effect here; for example, we still need four times as many bitops even when $N_{instr} = 6$.

Assuming that the datapath width is matched ($W_{simd} = W_{app}$), allows us to focus on the instruction mismatch; we can show this in an area model as:

$$\begin{aligned} &\mathit{Area}\left(\text{Arch}\left(W_{app}, N_{instr}\right), \text{ App}\left(W_{app}, L_{path}\right)\right) \\ &= \left\lceil \frac{L_{path}}{N_{instr}} \right\rceil \times A_{bitop\_w\_imem}\left(W_{app}, N_{instr}\right) \\ &= \left\lceil \frac{L_{path}}{N_{instr}} \right\rceil \times \left(A_{bop} + \left(\frac{N_{instr}}{W_{app}}\right) \times A_{pinst}\right) \end{aligned} \tag{36.23}$$

This allows us to compute the efficiency of the mismatched instruction store at a matched W_{app} as:

$$\text{Efficiency } [W_{app}](N_{instr}, L_{path}) = \frac{\left(A_{bop} + \left(\frac{L_{path}}{W_{app}}\right) \times A_{pinst}\right)}{\left\lceil \frac{L_{path}}{N_{instr}} \right\rceil \times \left(A_{bop} + \left(\frac{N_{instr}}{W_{app}}\right) \times A_{pinst}\right)} \tag{36.24}$$

Figure 36.8 plots the efficiency from equation 36.24 versus L_{path} for a collection of N_{instrs}'s and W_{apps}'s. Again, note that instruction store mismatches can cost orders of magnitude in net density. We also see robust points here where the net density remains within 50 percent of the matched architecture. The effect is the same as for datapath width mismatch (see previous section), and the efficient points are governed by an analogous equation:

$$A_{bop} = \left(\frac{N_{instr}}{W_{app}}\right) \times A_{pinst} \tag{36.25}$$

For any of these robust points, at the minimum value, $L_{path} = 1$, we are using all the compute area and only a fraction of the instruction memory area, so an optimally matched architecture could, at best, be implemented in half the area. Similarly, for arbitrarily large L_{path}, if $N_{instr} < L_{path}$, all the instruction memory area is used to hold instructions, but this may leave the compute area idle most of the time. Here, again, with only 50 percent of the area in compute, the design is, at most, twice the size of an optimally matched architecture with less area allocated to computation. In contrast, if we put 90 percent of the area into compute, then we could end up wasting 90 percent of the area in scenarios where $L_{path} >> N_{instr}$; matched architectures can be an order of magnitude smaller in such cases. Similarly, if 90 percent of the area is put into instruction memory, we can end up wasting almost 90 percent of the area when L_{path} is small.

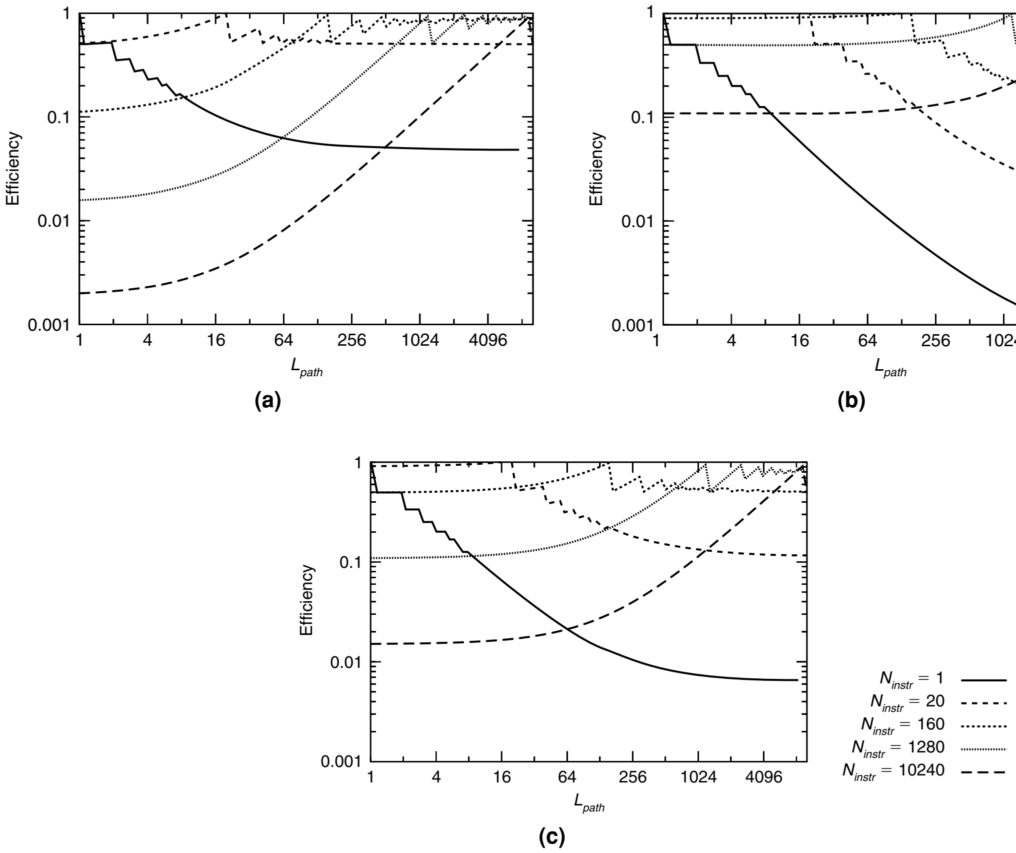

FIGURE 36.8 ■ Efficiency as a function of L_{path} for various W_{app} values: (a) $W_{app} = 1$, (b) $W_{app} = 64$, and (c) $W_{app} = 8$.

Composite effects

Combining the effects of SIMD width mismatch and local instruction storage mismatch, we get the total efficiency:

$$\text{Efficiency}\ \left(\text{Arch}(W_{simd}, N_{instr}), \text{App}\left(W_{app}, L_{path}\right)\right)$$

$$= \frac{\left(A_{bop} + \left(\frac{L_{path}}{W_{app}}\right) \times A_{pinst}\right)}{\left(\frac{W_{simd}}{W_{app}}\right) \times \left\lceil \frac{W_{app}}{W_{simd}} \right\rceil \times \left\lceil \frac{L_{path}}{N_{instr}} \right\rceil \times \left(A_{bop} + \left(\frac{N_{instr}}{W_{simd}}\right) \times A_{pinst}\right)} \tag{36.26}$$

Unfortunately, if both the SIMD width and the local instruction storage mismatch, it is not possible to pick a robust point as we did in previous sections.

Returning to equations 36.22 and 36.25, we note that the robust points occur when we can match the instruction storage area, $\left(\frac{N_{instr}}{W_{simd}}\right) \times A_{pinst}$, and the computation and interconnect area, A_{bop}. However, when both W_{app} and L_{path} vary, even when the area is matched, we can have cases where the allocation of width versus storage size within that area can prevent us from using the computational units efficiently.

Efficiency of processors and FPGAs

The previous section suggests that we will not find an architectural point in this (W_{simd}, N_{instr}) design space that is efficient across a wide range of application structures. To understand where processors and FPGAs are efficient, we can use the composite efficiency relation (equation 36.26) and estimate how efficient they each can be across a portion of the design space (see Figure 36.9). Here the FPGA is naturally modeled with $N_{instr} = 1$ and $W_{simd} = 1$. We model a processor as $W_{simd} = 64$ and $N_{instr} = 16{,}384$.

Figure 36.9 shows starkly that the FPGA and processor are both designed for different points in the application space. Notice that each can be less than 1 percent efficient in some portions of the space. Further, we note that *in the places where the processor is very inefficient (< 1 percent), the FPGA is highly efficient; the reverse is true as well.* This effect, coupled with the heterogeneous nature of applications, explains why it is often useful to have reconfigurable systems that mix FPGA or reconfigurable fabrics along with processors (e.g., Instruction augmentation subsection of Section 5.2.2 and Chapter 26).

36.4.3 Caveats

As noted in the introduction to this chapter, we are deliberately using a simple model to illustrate key effects in instruction organization. There are many other application structural opportunities and architectural variables that can also have a large effect on resource balance and efficiency, including interconnect richness (e.g., [9]) and organization, data storage and memory hierarchy capacities, bandwidth and latencies, threads of control, dynamic instruction selection, and integration of hardware functional units (e.g., multipliers [10,11]

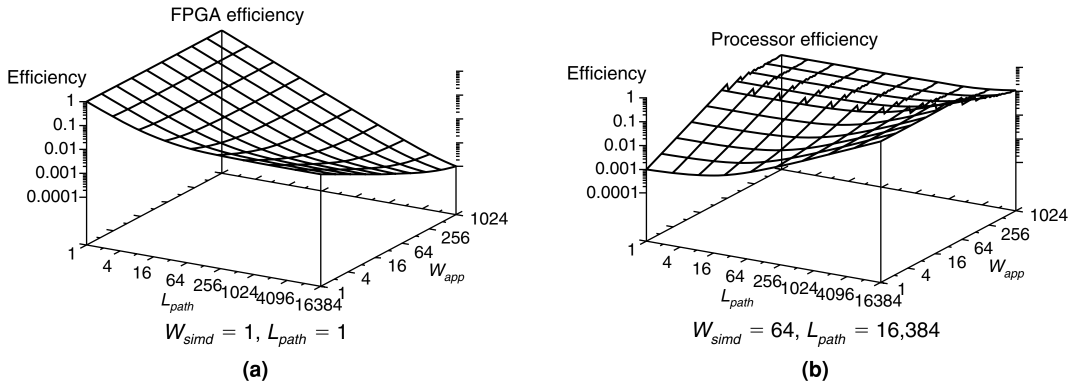

FIGURE 36.9 ■ Efficiency of FPGA-like (a) and processor-like (b) designs across both L_{path} and W_{app}.

and floating-point units [12]). In processors, the SIMD control of ALUs is coupled with fast logic to support carries in arithmetic (e.g., [13]), which serves to reduce $L_{critpath}$; FPGAs also employ fast cascade structures for similar reasons (e.g., [14], Chapter 1) but do not tie them to SIMD datapaths. Nonetheless, the simple model shows that these instruction organization decisions can have a significant impact on computational density, and it illustrates why FPGAs can be more efficient than processors for important classes of applications.

36.5 IMPLICATIONS

36.5.1 Density of Computation versus Description

From this model, we can clearly see a trade-off between computational density and instruction density. Equation 36.16 illustrates that the instruction store area for a single bitop can be an order of magnitude smaller than the computation to support it. This means an $N_{instr} = 1$ design stores instructions an order of magnitude less densely than an $N_{instr} = 200$ design, and an $N_{instr} = 200$ design packs computation an order of magnitude less densely than an $N_{instr} = 1$ design.

When the goal is to simply pack a large, irregular computation into a small area, we are best off focusing on instruction density; this minimizes the area for the implementation, at the expense of lower performance. When the goal is to perform the computation at high throughput, designs with high computational density allow us to meet the throughput with the least area.

36.5.2 Historical Appropriateness

When we first started building programmable integrated circuits, the premium for describing large computations was high. The capacity on a single integrated circuit was very low when they were built with $F = 3\,\mu\text{m}$ technology. In the mid-1980s, with $N_{instr} = 1$ and $W_{simd} = 1$, we could put only 64 bitops on a die [15], limiting computations to those that could be described by 64 instructions. At roughly the same time, one could put $N_{instr} = 512$ instructions on the die along with 32 bitops controlled in an SIMD fashion by a single pinst on each cycle ($W_{simd} = 32$) [4]. The struggle at this point in history was to fit an entire computational kernel onto a single die, and the deep instruction, word-wide processor design could begin to fit interesting kernels while the FPGA designs could fit only the most trivial computations.

By 2005, however, with $F \leq 0.1\,\mu\text{m}$, the landscape had changed. Moore's Law process scaling has given us more than a 10,000-fold increase in capacity per integrated circuit. Modern processors, still built with ever-deeper memories, have large enough instruction stores to contain large applications. At the same time, FPGAs hold hundreds of thousands of active bitops. Even kernels with thousands of 64-bit-wide operations can fit spatially on the FPGA and exploit the higher computational density.

The question with today's silicon is less "Can we get the application to fit on the die?" and more "How do we turn the available die area into performance?" Consequently, as we continue to scale feature sizes, the fraction of tasks where high instruction density remains the premium is shrinking, while the fraction where the application fits on the die and high computational density offers a benefit is increasing.

36.5.3 Reconfigurable Applications

Understanding why FPGAs can be efficient and where they are most efficient (e.g., Figure 36.9) provides additional insight into where we should use FPGAs and how to fully exploit their strengths. Certainly, if the task has low throughput requirements (i.e., large L_{path}), then FPGAs are often not an efficient implementation. The FPGA is efficient when we operate at minimum path length, preferably $L_{path} = 1$, where we are performing the same operation over and over and keeping all the bitops active during the operation. For FPGAs with a variable clock cycle, we want to keep the cycle time to the minimum, maximizing the reuse rate of each operation. This underscores why retiming operations such as pipelining and C-slow (see Chapter 18) are important for optimizing FPGA efficiency, as well as behavioral transformations that reduce $L_{critpath}$.

When L_{path} is large simply because of a low throughput demand, we can often turn the SIMD structure, W_{app}, into additional operation regularity. In particular, when $W_{app} > 1$, that is an indication that a number of bit-level operators do perform the same operation. By moving this regularity into time rather than space, we can reduce the number of unique instruction combinations needed and hence reduce the N_{instr} required. For example, if $W_{app} = 16$ and $L_{path} >> L_{critpath}$, we can implement the SIMD datapath bit serially so that the necessary instruction storage depth is a factor of 16 smaller ($N'_{instr} \approx \frac{L_{path}}{W_{app}}$). As shown in Figure 36.10, this can increase the FPGA's domain of efficiency.

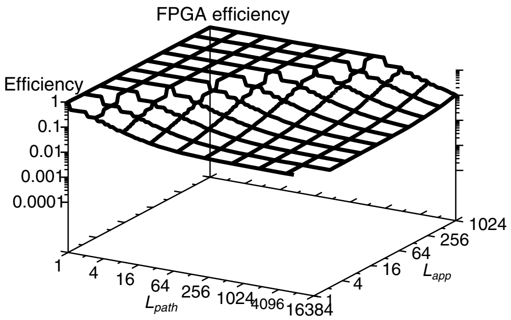

FIGURE 36.10 ■ FPGA efficiency when datapath regularity can be used to increase temporal regularity.

References

[1] D. S. Hochbaum, ed. *Approximation Algorithms for NP-Hard Problems,* PWS Publishing, 1997.

[2] International technology roadmap for semiconductors. *http://www.itrs.net/Links/2005ITRS/Home2005.htm*, 2005.

[3] A. DeHon. Entropy, counting, and programmable interconnect. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, ACM/SIGDA, 1996.

[4] M. Horowitz, J. Hennessy, P. Chow, G. Gulak, J. Acken, A. Agarwal, C.-Y. Chu, S. McFarling, S. Przybylski, S. Richardson, A. Salz, R. Simoni, D. Stark, P. Steenkiste, S. Tjiang, M. Wing. A 32b microprocessor with on-chip 2 Kbyte instruction cache. *IEEE International Solid-State Circuits Conference, Digest of Technical Papers*, IEEE, 1987.

[5] C. McNairy, R. Bhatia. Montecito: A dual-core, dual-thread Titanium processor. *IEEE Micro* 25(2), 2005.

[6] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki. Synergistic processing in cells multicore architecture. *IEEE Micro* 26(2), 2006.

[7] M. Gardner. The fantastic combinations of John Conway's new solitaire game "Life." *Scientific American* 223, 1970.

[8] K. Papadantonakis, N. Kapre, S. Chan, A. DeHon. Pipelining saturated accumulation. *Proceedings of the International Conference on Field-Programmable Technology*, 2005.

[9] A. DeHon. Balancing interconnect and computation in a reconfigurable computing array (or, why you don't really want 100% LUT utilization). *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 1999.

[10] A. DeHon. The density advantage of configurable computing. *IEEE Computer* 33(4), 2000.

[11] I. Kuon, J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26(2), 2007.

[12] M. J. Beauchamp, S. Hauck, K. D. Underwood, K. S. Hemmert. Embedded floating-point units in FPGAs. *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2006.

[13] R. P. Brent, H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers* 31(3), 1982.

[14] S. Hauck, M. M. Hosler, T. W. Fry. High-performance carry chains for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8(2), 2000.

[15] W. S. Carter, K. Duong, R. H. Freeman, H.-C. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo, S. L. Sze. A user programmable reconfigurable logic array. *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1986.