The need for instructions to control device operation is one distinguishing feature of general-purpose computing devices. These instructions give general-purpose devices their flexibility to solve a variety of problems. At the same time, instructions require dedicated resources for storage and delivery.

General-purpose computing architectures must address a number of important questions:

1. How are general-purpose processing resources controlled?
2. How much area is dedicated to holding the instructions which control these resources?
3. How many resources are controlled with each instruction?
4. How much bandwidth is provided for instruction distribution?
5. How frequently can instructions change?

There are many, different possible answers to these questions and the answers, in large part, distinguish the various general-purpose architecture categories which we reviewed in Chapter 4 (*e.g.* word-wide uniprocessor, SIMD, MIMD, VLIW, FPGA, reconfigurable ALU). The answers also play a large role in determining the efficiency with which the architecture can handle various applications.

In this chapter, we look at the problem of instruction control and the resources involved. We start by looking at the extreme case where every bit operation is given a unique instruction on every cycle. This example illustrates that instruction distribution resource requirements can be quite large – dominating other areas in a device. To combat these requirements, traditional architectures have placed various, stylized restrictions on instruction distribution in order to contain its resource requirements. Each of these restrictions also limits the realm of efficiency of the architecture. We review these restrictions and their effects on device utilization efficiency in Section 8.3. Of course, the opportunity to compress instruction distribution requirements depends on the inherent *compressibility* of the instruction stream suggesting that some computations will remain description limited while others are compute limited (Section 8.4). We also look at the issue of instruction stream control (Section 8.5). Finally, Section 8.6 organizes the architectural parameters reviewed in this chapter into an expanded taxonomy for multiple data processing architectures.

8.1 General Case Example

Consider that we have N_p bit processing elements. Each of these elements may be a 4-LUT as in the previous section or a one bit ALU. We want to provide a different instruction to each processing element on every cycle of operation. From Section 4, we see that 100 MHz+ operating frequencies are readily achievable today, with many devices already achieving higher frequencies. We will consider a 200 MHz operating frequency as one that will be easily achievable in the very near future. We saw from Section 7.8 that each 4-LUT needs 40-50 bits to describe its network configuration and 16 bits to control the logic function. We will thus assume each LUT requires a 64-bit instruction to control it.

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]

Let us further assume that we distribute the instruction, one per clock cycle, from all four sides of the array of N_p processing elements densely using 2 layers of metal with an 8λ wire pitch. Of course, the assumption that we dedicate two full metal layers to instruction distribution is extreme, but even making this best case assumption, we will see that the resources required for full instruction control can dominate all other concerns. For the sake of easy comparison, we will target a $1024\lambda \times 1024\lambda = 1\text{M}\lambda^2$ array element, which is on par with large, conventional 4-LUT FPGA implementations (Table 7.1).

Across the width of one processing element, we can run instruction distribution busses to control two such elements:

$$2 \times \underbrace{64}_{\text{bits/LUT}} \times \underbrace{8\lambda}_{\text{wire pitch}} = 1024\lambda$$

This means, we can support an array with $2\times$ as many processing elements (N_p) as it has edge widths. That is:

$$2 \times 4 \times \sqrt{N_p} = N_p$$

From which we conclude $N_p = 64$.

At this point, we have fully saturated the i/o bandwidth into the compute array. Any further increase in the number of elements supported in the array must be accompanied by a corresponding increase in LUT size. That is:

$$\begin{aligned} \frac{\sqrt{A_{LUT}}}{64 \times 8\lambda} \times 4 \times \sqrt{N_p} &= N_p \\ \frac{\sqrt{A_{LUT}}}{128\lambda} &= \sqrt{N_p} \\ A_{LUT} &= 16384\lambda^2 N_p \end{aligned} \tag{8.1}$$

Consequently, LUT area increases linearly with the number of processing elements supported in the array. *The instruction distribution bandwidth requirement is the dominante size effect determining the density with which computational elements can be built.* Notice that the LUT area growth rate dictated by instruction bandwidth is larger than the interconnect growth rate for any value of $p < 1$ and, ultimately, both interconnect and instruction distribution compete for the same limited resource – wire bandwidth.

Further, we can calculate the actual instruction bandwidth requirements. At 200 MHz and 64-bit instructions, each LUT requires 1.6GBytes/s of instruction distribution. For the $N_p = 64$ case above, this amounts to over 100GBytes/s.

This kind of bandwidth could not, reasonably, be supported from off chip with any contemporary technology, necessitating on-chip instruction memory. At $1000\lambda^2$ per SRAM cell, 16 64-bit instructions will occupy the same space as each $1\text{M}\lambda^2$ processing element. If more than 16 unique instruction sets are required, instruction memory will occupy more area than the processing elements and interconnect.

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]

8.2 Bits per Instruction

In the previous section we assumed 64 bits per instruction. This seems to be a reasonable, ballpark estimate of the number of bits required to describe a single bit operation, including interconnect.

Processors Modern processors generally employ 32 bits per instruction. However, as we saw in Section 4.1, about half of the instructions issued by a microprocessor are interconnect operations. Particularly, when we looked at gate evaluations, we saw that each processor instruction describes, on average, about 0.5-0.6 gate evaluations.

FPGAs Modern FPGAs use 120-200 bits per 4-LUT. We pointed out in the previous chapter that this was due to non-sparse encoding, and much denser encodings were possible. Simply using the crossbar bound from Section 7.8.1, we see that we can handle a 4000 4-LUT device with 48 bits of interconnect description and 16 bits of logic description. VEGA, a heavily multicontexted FPGA, with 85 bits per instruction, comes closer to this range [JL95].

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]

8.3 Compressing Instruction Stream Requirements

Section 8.1 showed us that we cannot afford to have full, independent, cycle-by-cyle control of every bit operation without instruction storage and distribution requirements dominating all other resource requirements. Consequently, we generally search for application characteristics which allow us to describe the computation more compactly. In this section, we review the most common techniques generally employed to reduce instruction size and bandwidth. We see that every architecture reviewed in Chapter 4 exploits one or more of these compression techniques.

8.3.1 Wide Word Architectures

Processors do not, commonly, operate on single bit data items. Rather, sets of w bit elements ($w = \{8, 16, 32, 64\}$) are grouped together and controlled by a single instruction in SIMD style. This has the effect of reducing instruction bandwidth requirements and instruction storage requirements by a factor of w. This compression scheme takes advantage of the fact that we commonly do want to operate uniformly on multibit quantities. We can, therefore, effectively amortize instruction resources across multiple bit processing elements.

Returning to our opening example, we can support w^2 more processing elements before reaching the same point of wire saturation:

$$\begin{aligned} 2 \times 4 \times \sqrt{N_p} &= \frac{N_p}{w} \\ 8w &= \sqrt{N_p} \\ N_p &= 64w^2 \end{aligned}$$

The utilization efficiency of the resulting architecture depends on the extent to which all operations are w-bit operations, or even multiples thereof. When smaller operations, $w_d < w$, are required, $w - w_d$ bit processing units will sit idle while only w_d units provide useful work.

8.3.2 Broadcast Single Instruction to Multiple Compute Units

SIMD and vector machines take this instruction sharing one step further. They arrange so that multiple functional units operating on nominally different words share the same instruction. This allows them to scale up the number of bit operators without increasing the word granularity or instruction bandwidth. It does, however, increase the operation granularity. To remain efficient now, the application requires n_v w-bit operations, where n_v, is the number of word-wide datapaths controlled by each instruction.

8.3.3 Locally Configure Instruction

Reconfigurable architectures, such as FPGAs, take advantage of the fact that little instruction bandwidth is needed if the instructions do not change on every cycle. Each bit processing element gets its own, unique, instruction which is stored locally. However, this instruction cannot change from cycle to cycle. A limited bandwidth path is used to change array instructions when necessary.

There are two viewpoints from which to approach the efficiency of this restriction:

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]

1. **Frequency of Instruction Change** – Given a lower bandwidth path, I_{BW}, and N_p compute elements with n_{ibits} long instructions, each context reload will take $\frac{n_{ibits} \cdot N_p}{I_{BW}}$. If the inter arrival time between reloads is n_{cycle}, the efficiency of operations is equal to the fraction of time spent computing versus total compute and reload time:

$$\text{Efficiency} = \frac{n_{cycle} \cdot t_{cycle}}{(n_{cycle} \cdot t_{cycle}) + \left(\frac{n_{ibits} \cdot N_p}{I_{BW}}\right)}$$

   This, of course, assumes that every processor is doing useful work on each cycle.

2. **Task Critical Path Length** – Alternately, if we assume the computing array is sufficiently large to perform the task, the efficiency is the fraction of compute elements performing useful work on each cycle. If the device has N_p processors and must perform a task with N_{bit_ops} which has a critical path of length L_{crit}, then the efficiency is the number of useful bit operations divided by the total number of processors and the critical path length:

$$\text{Efficiency} = \frac{N_{bit_ops}}{L_{crit} \times N_p}$$

For the frequency of change case, *partial reconfiguration* can reduce the reload time by allowing individual processing units to change instructions without requiring an entire reload of all instructions in the array. This can increase reload efficiency if a large fraction of the instructions does not change. Partial reconfiguration can also allow portions of the array to change their instructions while other portions of the array continue to operate. Modern FPGAs from Plessey [Ple90], Atmel [Atm94], and Xilinx [Xil96] support partial reconfiguration for these reasons.

8.3.4 Broadcast Instruction Identifier, Lookup in Local Store

A hybrid form of instruction compression is to broadcast a single instruction identifier and lookup its meaning locally. This allows us to use a moderately short, single "instruction" across the entire array in a manner similar to SIMD instruction broadcast. Each processing element performs a local lookup from the broadcast instruction identifier to generate a full length instruction. DPGAs (Chapter 10), PADDI [CR92], and VLIW machines with an independent cache for each functional unit, exhibit this kind of hybrid control.

This technique is similar to a dictionary compression scheme where the set of entries at each "instruction" address makes up one element in the dictionary. The instruction address is the encoded symbol which can now be transmitted into the array with minimal bandwidth. The key benefit here is that the parallel instruction sets can be tailored to each application in the same way the dictionary can be tailored to a message or message type.

Efficiency in this scheme is similar to the task critical path length case above. The difference being, that each single processor need not be dedicated to a single instruction. With local instruction stores each holding n_{inst} instructions, an array of N_p processing nodes can perform up to $n_{inst} \cdot N_p$ different bit operations. In the single instruction configuration case, a critical path of $L_{crit} > 1$ implied a peak, achievable efficiency of $\frac{1}{L_{crit}}$. In this case, the peak, achievable efficiency is $\min\left(\frac{n_{inst}}{L_{crit}}, 1\right)$.

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]

Viewed in terms of instruction change frequency, the additional local configurations can serve as a cache, diminishing the need to fetch array instruction from outside the array. Like a cache if each instruction set required is unique, it will provide no benefit. However, when an array instruction can be kept in the array and used several times before being replaced, we reduce the required instruction bandwidth. Use of a loaded instruction can occur at the operational cycle rate rather than at the bandwidth limited reload rate.

With multiple configurations it is possible to arrange for instruction reload to occur as a background task operating in parallel with operation. If the reload time, $\frac{n_{ibits} \cdot N_p}{I_{BW}}$, is less than the run time within the balance of the loaded instruction memory, $n_{cycle} \cdot t_{cycle}$, and the next instruction can be predicted sufficiently in advance, reload time can be completely overlapped with computation.

8.3.5 Encode Length by Likelihood

Since it is unlikely that all instructions will be used with equal frequency, one can break instructions into a series of smaller words, giving common instruction short encodings. If we huffman encode our instruction into a series of s-bit words, we can, potentially, reduce the instruction distribution bandwidth by a factor of $\frac{s}{\log_2(|\text{instructions}|)}$; that is, $\frac{s}{64}$ if we assume 64 bit instructions.

The efficiency now depends on the expected number of s-bit words required to construct a single, logical instruction. If the instruction stream entropy is low, this kind of encoding can be very efficient — asymptotically approaching one symbol per instruction. Counterwise, if the instruction stream entropy is high – or even flat, it may take a full $\frac{\log_2(|\text{instructions}|)}{s}$ cycles to built up a single instruction. Worse, if the instruction frequency is substantially different from the instruction frequency for which the encoding was optimized, it can actually take more cycles, on average, to build an instruction. Of course, the huffman encodings could be variable, as well, to avoid this mismatch, but the space required to handle programmable huffman encodings will likely exceed the area of several computational units.

8.3.6 Mode Bits for Early Bound information

All of the bits in an instruction do not always need to change at once — or portions of an instruction may change at different rates. Rather than include the infrequently changing portions of the instruction in the word which is broadcast from cycle to cycle, these portions can be factored out of the broadcast instruction and explicitly loaded with new values only when they need to change. This allows us to describe richer instructions with less bandwidth. These locally configured instructions can be seen as a special case of the previous section on likelihood encoding — but in this case we exploit the low frequency of change rather than simply the low frequency of occurrence of some instructions.

Mode bits such as these are used to define operational modes in several architectures. Floating point coprocessors often use mode bits to define rounding modes. Segmented SIMD architectures such as Abacus [BSV+95] and the dynamic computer groups of [KK79] use mode bits to define segmentation of the SIMD datapaths.

Bandwidth savings depends on the number of bits factored out of the broadcast instruction stream. Efficiency depends on the frequency with which non-broadcast instruction values need to

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]

change. Typically, it takes an instruction cycle to load each mode value – which is an instruction cycle which does not serve a purpose towards execution.

8.3.7 Themes

Two major themes emerge from the techniques listed here:

1. **Granularity** – How many resources are controlled by a each instruction? From a resource cost standpoint, this is the motivation behind word-wide datapaths, SIMD, and vector processors. processing
2. **Local Configuration Memory** – How many instructions are stored locally per active computing element? Similarly, this is the motivation behind configurable architectures and local memories.

In the next chapter, we will look effects which these techniques have both on resource requirements and on utilization efficiency.

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]

8.4 Compressibility

Of course, we can only succeed in compressing the instruction bandwidth when there is structure to the task description for us to exploit. If the task descriptive complexity really is as large as implied in Section 8.1, we are instruction bandwidth limited, and instruction distribution does determine achievable, computational density.

This suggests we have two extremes in the characterization of computing tasks:

1. **Descriptive Complexity Limited** – the instruction bandwidth to describe the computation limits the rate of execution.

2. **Compute Limited** – the active computing elements performing the required computation limit the rate of execution.

Regular tasks such as signal and stream processing, systolic computations, and computational inner loops are typically compute limited. Irregular, run once, tasks such as initialization, cleanup, and exception handling are typically descriptive complexity limited. Of course, applications tend to have a mix of both elements. It has long been observed that only a small fraction of the code in a typical application accounts for most of the computational time [Knu71]. The regions composing this small fraction are heavily reused, allowing the computation to be described compactly. The code outside of the heavily used fraction, does not benefit from the heavy reuse amortization and will tend to be more description limited.

As with most compression schemes, the amount of compression achievable, in practice, also depends heavily on the frequency of repetition and storage space available. For example, if a task performs a sequence of one million, unique operations, then restarts the sequence, the stream is very repetitive, and an infinite sequence of such such repetitions contains a constant amount of information. However, unless we have space to hold all one million instructions on chip, we will not be able to take advantage of this regularity and low information content in order to compress instruction bandwidth requirements. Further, holding one million instructions on chip is a large cost to pay for instruction storage, even by today's standards.

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]

8.5 Control Streams

In Sections 8.1 and 8.3, we viewed the set of processing elements as having a single, large, array-wide, instruction. In general, the array-wide instruction context may be decomposed into a number of independent instruction streams. This decomposition does not change the aggregate instruction bandwidth which may be required into the array, but it may change the number of distinct contexts used by the array and hence the requirements for instruction distribution and storage.

Let us assume, as in the case of Section 8.3.4, that each processing element has a local store of n_{inst} instructions. Let us also assume we have a series of m independent tasks, each composed of at most n_{task} instructions. The total number of distinct, array-wide contexts may be as large as $(n_{task})^m$, since the tasks are independent and any combination of instructions is possible. If each of the tasks is controlled separately, we need only $n_{inst} = n_{task}$ instructions to describe and control the tasks. If we must control the m tasks with a single instruction stream, that stream requires all $(n_{task})^m$ contexts and hence a larger number of instructions, $n_{inst} = (n_{task})^m$, are required.

This example demonstrates that there is a control granularity which is a distinct entity from the operation granularity introduced in Sections 8.3.1 and 8.3.2. As with operation granularity, we can compress instruction control requirements by sharing the control among a number of operating units. However, if we control too many units by the same control stream, we are forced to use the device inefficiently. In the worst case, we may pay an efficiency or compaction penalty in proportion to the product of the instruction sets of the independent operations which must be combined into a single control stream.

The separate streams of control are, of course, what distinguishes MIMD architectures (Section 4.9), as well as MSIMD (*e.g.* [Bri90, Nut77]) or MIMD multigauge [Sny85] architectures.

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]

Control Threads (PCs)	**Instructions per Control Thread**	**Instruction Depth**	**Granularity**	**Architecture/Examples**
0	0	0	n/a	Hardwired Functional Unit (*e.g.* ECC/EDC Unit, FP MPY, Hardware Systolic)
	n	1	1	FPGA, Programmable Cellular Automata
			w	Reconfigurable ALUs Programmable Systolic Datapath Arrays
1	1	c	$n_v \cdot 1$	Bitwise SIMD
			w	Traditional Processors
			$n_v \cdot w$	Vector Processors
	n	c	1	DPGA
		8	16	PADDI
		c	w	VLIW
m	1	c	$n_v \cdot w$	MSIMD
m	1	c	1	VEGA
		8	16	PADDI-2
		c	w	MIMD (traditional)

Table 8.1: Instruction Control Taxonomy

8.6 Instruction Stream Taxonomy

Table 8.1 categorizes the various architectures we have reviewed in Chapter 4 according to the granularity (w,n_v), local instruction storage depth (c), number of distinct instructions per control thread (n), and number of control threads (m) supported. This taxonomy elaborates the multiple data portion of Flynn's classic architecture taxonomy [Fly66] by segregating instructions from control threads and adding granularity.

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]

8.7 Summary

In this section, we have seen that the requirements for instruction distribution and storage can dominate all other resources on general-purpose computing devices, dictating the size and density of computing elements. A major distinguishing feature of modern, general-purpose architectures is the way in which they compress the requirements for instruction control. Traditional microprocessors, SIMD, and vector machines reduce the requirements by sharing a single instruction across many bits or words. FPGAs and programmable systolic arrays reduce requirements by maintaining the same instruction from cycle to cycle. VLIW-like architectures use small, local instruction stores addressed by short addresses so that limited instruction distribution bandwidth can effect cycle-by-cycle changes in non-uniform instructions. Each of the techniques used to reduce instruction control resources comes with its own limitations on achievable efficiency should the needs of the application not meet the stylized form in which the instruction bandwidth reduction is performed. Some instruction sequences are more compressible than others, suggesting we have a continuum of task descriptive complexities such that some tasks are, by nature, instruction bandwidth limited while others are parallel computing resource limited. In this chapter we reviewed both the nature of the resource reductions and the efficiency limits which arise from these techniques. In the following chapter, we will combine these effects with our size and growth observations from Chapters 4 and 7 to model the size and efficiency of reconfigurable computing architectures.

[See <http://www.ai.mit.edu/people/andre/phd/phd.html> for complete document.]