Extract from: Reconfigurable Architecture for General Purpose Computing

4. Empirical Review of General Purpose Computing Architectures in the Age of MOS VLSI

Here we review various general-purpose computing architectures by taking an empirical look at their implementations during the past decade. In this section we draw from the whole realm of general-purpose architectures – not just those which fit directly into our *RP*-space. This makes a larger set of design points avilable for review, but also introduces considerably more variation in architectures than we will focus on in later chapters. We look primarily at general-purpose capacity in this section, generally ignoring the effects of specialized functional units. The following chapter will look at the effects of custom multipiers, the most common specialized functional unit added to nominally general-purpose computing devices. The focus here is on integrated, single IC, computational building blocks to keep the comparison as consistent as possible across such a wide variety of architectures. Additionally, we focus entirely on MOS VLSI implementations since most of these architecture have had multiple MOS VLSI implementations and the effects of MOS feature size device scaling are moderately well understood.

4.1 Processors

We start by looking at a simple RISC-style processor.

Model The pedagogical processor model (Figure 4.1) is composed of:

- n-bit ALU (two read, one write port from register file) [for the sake of comparison, we do include multiple ALU processors in the empirical review]
- memory (Register File, Instruction/Data Cache)
- control (sequencer, cache control, load/store unit, etc.)

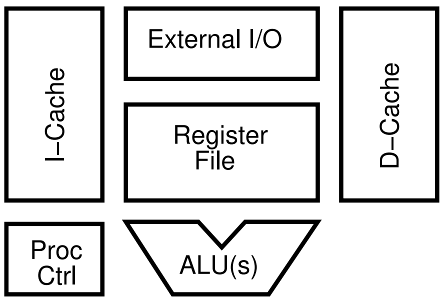


Figure 4.1: Basic Organization for a Processor

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

Instruction	Gate Evaluation Capacity	Explanation
LD, ST, MOVE	0	overhead allowing reuse and interconnect
ADD, SUB, CMP	$2n$	Each full adder bit is 2 gate evaluations
AND, OR, XOR,	n	One gate evaluation per bit
BEQ,BNE	$\approx \frac{n}{3}$	n-bit AND reduction $\frac{n}{4} + \frac{n}{16} + \cdots + 1$
B, CALL, RETURN	0	flow control overhead
SHIFT	0	interconnect

Table 4.1: Basic ALU Operations and Capacities

The ALU is the sole source of general-purpose capacity. Table 4.1 shows the traditional ALU operations provided by the ALU along with the computational capacity provided by each operation. An n-bit ALU provides n ALU bit operations. For this simple processor, no multiplier or specialized coprocessor is included. We will look at hardwired multiply implementations separately in Chapter 5 as an example of such specialized coprocessors. Each ALU operation operates in one processor clock cycle.

Capacity Provided We extract a maximum of $2n$ gate evaluations (n ALU bit operations) per cycle. Modern processors are achieving cycle times as low as 2-5ns with $n = 128$. The fastest, single-ALU processors today thus offer a peak capacity around 84 gate-evaluations/ns. Table 4.2 compares several processor implementations over the past decade. Results are summarized there in terms of ALU bit ops since that is the native, and hence most accurate, unit for processors. From Table 4.2, we see that conventional processors have provided a peak functional density of 3-9 ALU bit operations/λ^2s over the past decade. We see from Table 4.1 and some simple weightings below that an ALU bit op is somewhere between one half and two 3-LUT gate evaluations.

It is interesting, and perhaps a bit unexpected, to note how consistent this capacity density has been over time. We might have expected:

1. delays to improve with process such that $\lambda^2\tau$ would be a better measure of process-normalized capacity than λ^2s [τ is the delay parameter for a process which nominally amounts to the intrinsic RC delay for gates. One τ is the delay required for one inverter to drive a single inverter of equal size.]

2. architectural or circuit design improvements to have increased functional density over time

Given the displayed consistency, we may be seeing compensating effects from:

1. velocity saturation in the CMOS devices, especially submicron CMOS prevents the expected τ scaling

2. decreasing relative performance of on-chip interconnect with scaling

Extract from: Reconfigurable Architecture for General Purpose Computing

Year	Design	Organization	Die Size	λ	λ^2 area	cycle	$\frac{\text{ALU bit ops}}{\lambda^2 s}$
1984	RISC II [SKPS84]	1×32 (100%)	4.3mm×7.7mm	1.5μ	15M	330 ns	6.5
1984	MIPS [RPJ+84]	1×32 (100%)	5.5mm×6.1mm	1.5μ	15M	250 ns	8.5
1987	MIPS-X [HHC+87]	1×32 (100%)	8mm×8.5mm	1.0μ	68M	50 ns	9.4
1987	PA-RISC [YFJ+87]	1×32 (100%)	8.4mm×8.4mm	0.75μ	125M	66 ns	3.9
1988	SPARC [QC88, TFT+85]	1×32 (100%)	12.1mm×12.7mm	0.75μ	273M	60 ns	2.0
1990	PA-RISC [TLB+90]	1×32 (100%)	14mm×14mm	0.5μ	784M	11 ns	3.7
1990	SPARC [MMN+90]	1×64 (75%) IEEE FPU (25%)	14.9mm×15.1mm	0.4μ	1.4G	25 ns	2.4
1992	SuperSparc [ANAB+92]	2×32 (82%) IEEE FPU (18%)	16mm×16mm	0.4μ	1.6G	25 ns	2.0
1992	Alpha [DWA+92]	1×64 (81%) IEEE FPU (19%)	16.8mm×13.9mm	0.38μ	1.7G	5 ns	9.5
1994	PA-RISC [RDB+94]	2×64 (88%) IEEE FPU (12%)	14mm×15mm	0.28μ	2.8G	7 ns	7.4
1994	MIPS [SYN+94]	1×32 (100%)	7.9mm×8.8 mm	0.2μ	1.7G	2 ns	9.1
1995	PowerPC [BBB+95]	2×64 (87%) IEEE FPU (13%)	18.2mm×17.1mm	0.25μ	5G	7.5 ns	3.9
1995	UltraSparc [CDd+95]	2×64 (84%) 2×FP/GFU (16%)	17.7mm×17.8mm	0.25μ	5G	6 ns	5.0
1995	SPARC V9 [SPA+95] [CDF+95]	4×64 (80%) IEEE FPU (20%)	297mm^2	0.2μ	7.4G	6.5 ns	6.6
1995	Alpha [BAB+95]	2×64 (90%) 2×IEEE FPU (10%)	16.5mm×18.1mm	0.25μ	4.8G	3.3 ns	9.0
1996	MIPS [KDS+96]	1×32	9.1mm×8.3mm	0.25μ	1.2G	10 ns	2.6
1996	PA-RISC [LLNK96]	2×64 (80%) IEEE FPU (20%)	17.7mm×19.1mm	0.25μ	5.4G	4 ns	7.4
1996	ARM [MWA+96]	1×32	7.8mm×6.4mm	0.18μ	1.6G	5 ns	4
1996	Alpha [GBB+96]	2×64 (95%) 2×IEEE FPU (5%)	14.5mm×14.4mm	0.18μ	6.8G	2.3 ns	8.6

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{iALU}}{\frac{Area}{\lambda^2} \times P_{fraction} \times t_{cycle}}$$

$P_{fraction}$ is the fraction of the die not including any specialized coprocessors – primarily omitting any FPUs.

Table 4.2: Survey of Processor Capacity

Year	Design	Ref.	$F_{density}$	$I_{density}$	$D_{desnsity}$
1984	RISC II	[SKPS84]	6.5	0	3.0×10^{-4}
1984	MIPS	[RPJ+84]	8.5	0	3.4×10^{-5}
1987	MIPS-X	[HHC+87]	9.4	7.5×10^{-6}	1.5×10^{-5}
1987	PA-RISC	[YFJ+87]	3.9	0	8.2×10^{-6}
1988	SPARC	[QC88]	2.0	0	1.9×10^{-5}
1990	PA-RISC	[TLB+90]	3.7	0	1.3×10^{-6}
1990	SPARC	[MMN+90]	2.4	1.5×10^{-6}	1.9×10^{-5}
1992	SuperSparc	[ANAB+92]	2.0	3.9×10^{-6}	1.0×10^{-4}
1992	Alpha	[DWA+92]	9.5	1.2×10^{-6}	4.1×10^{-5}
1994	PA-RISC	[RDB+94]	7.4	0	7.5×10^{-6}
1994	MIPS	[SYN+94]	9.1	1.5×10^{-7}	5.3×10^{-6}
1995	PowerPC	[BBB+95]	3.9	1.9×10^{-6}	6.0×10^{-6}
1995	UltraSparc	[CDd+95]	5.0	9.7×10^{-7}	3.3×10^{-5}
1995	SPARC V9	[SPA+95]	6.6	0	1.1×10^{-5}
1995	Alpha	[BAB+95]	9.0	$4.8\text{-}57\times10^{-7}$	$1.5\text{-}18\times10^{-5}$
1996	MIPS	[KDS+96]	2.6	8.4×10^{-7}	2.8×10^{-5}
1996	PA-RISC	[LLNK96]	7.4	0	4.7×10^{-7}
1996	ARM	[MWA+96]	4	2.5×10^{-6}	8.0×10^{-5}
1996	Alpha	[GBB+96]	8.6	$3.1\text{-}38\times10^{-7}$	$1.0\text{-}12\times10^{-5}$

Table 4.3: Processor Capacity Summary

3. increasing chip size implies increasing wire runs – the area occupied by long interconnect wires is not scaling with λ^2
4. increasing use of CAD – designers are giving up some density in order to manage the complexity associated with the larger and larger designs
5. increasing gap between on-chip and off-chip performance necessitates dedicating more on-chip area to non-active resources, particularly memory, to compensate for the i/o bottleneck
6. increasing area being dedicated to control and state management to prevent control and data dependent stalls in the instruction stream

This peak computational density assumes that every operation on each n-bit ALU performs an n-bit compute operation and the processor completes one instructions per ALU on every cycle. In practice, a significant number of processor cycles are not spent executing compute operations.

- **Limited instruction and data bandwidth** – coupled with long latencies to off-chip resources, limited bandwidth can cause the processor to stall waiting on the information which it needs to determine the course of the computation or the data it needs to operate upon. As a result, the number of instructions completed per cycle is always less than the number of ALUs available.
- **Abstraction overhead and data movement consume capacity** – procedure calls, data marshaling, traps, and data conversion consume capacity without, directly, providing capacity

Application	$\frac{\text{Gate Evaluations}}{\text{Datapath Bit}}$
GCC	0.5
TeX	0.5
US Steel	0.8
Average	0.6

Table 4.4: Average Gate Evaluations/Datapath Bit

to the problem. Operations which simply move data around do not provide capacity to the problem either. Since we have a mix of the instructions shown in Table 4.1, we will never achieve the peak condition where every operation provides $2n$ gate-evaluations to the computational task.

Quantitative studies tell us, for given systems or application sets, average values for the number of gate evaluations per instruction and the number of instructions executed per clock cycle. This gives us an expected computational capacity:

$$\mathrm{E}(\text{Functional Density}) = \frac{\text{Gate Evaluations}}{\text{Datapath Bit}} \times \frac{\text{Datapath Bits}}{\text{Instruction}} \times \frac{\text{Instructions}}{\text{Issue Slot}} \times \frac{\text{Issue Slots}}{\text{Clock Cycle}} \times \frac{1}{\text{area} \times t_{cycle}}$$

For example, while HaL's SPARC64 should be able to issue 4 instructions per cycle, in practice it only issues 1.2 instructions per cycle on common workloads [EG95]. Thus $\frac{\text{Instructions}}{\text{Issue Slot}} \approx 30\%$, resulting in a 70% reduction from expected peak capacity.

Assuming the integer DLX instructions mixes given in Appendix C of [HP90] are typical, we can calculate $\frac{\text{Gate Evaluations}}{\text{Datapath Bit}}$ by weighting the instructions by their provided capacities from Table 4.1. In Table 4.4 we see that one ALU bit op in these applications is roughly 0.6 gate-evaluations.

If this effect is jointly typical with the instructions per issue slot number above, then we would yield at most $0.3 \times 0.6 \approx 0.2$ of the theoretical, functional density. For the HaL case, this reduces 6.6 ALU Bit Ops$/\lambda^2$s to 1.3 gate-evaluations$/\lambda^2$s.

There are, of course, several additional aspects which prevent most any application from achieving even this expected peak capacity and which cause many applications to not even come close to it.

1. **Coarse-grained datapath/network** – the word-oriented datapaths in conventional processors prevents efficient interaction of arbitrary bits – *e.g.* A simple operation like XOR-ing together two fields in a data word requires a mask, shift, and XOR operation even though the whole operation may effect only a few gate evaluations. Returning to the HaL example above, we note that the processor has a 64-bit datapath. When running code developed for 32-bit SPARCs, half of the datapath is idle, further reducing the yielded capacity by 50% to 0.7 gate-evaluations$/\lambda^2$s.

//R1 x_i base pointer	
//R2 avg_i base pointer	
//R3 working sum	
//R6 buffer top	
ld -32[R1],R4	*//value dropping out of window*
sub R4,R3,R3	*//subtract out*
ld 0[R1],R4	*//value entering window*
add R4,R3,R3	*//add in*
asr R4,R5,#3	*//divide by 8*
st [R2],R5	*//save new average*
addi R1,#4,R1	*//increment x ptr*
ble R1,R6,top	*//branch to top*
addi R2,#4,R2	*//increment avg ptr (delay slot)*

Figure 4.2: Inner Loop of Processor Implementation for Windowed Average

2. **Limited control of ALU capacity** – the n-bit ALU mostly functions as a collection of n-bit processors controlled in SIMD fashion. Full capacity is only extracted when all n bits arranged in a dataword need the same operation. Data with smaller ranges (less than 2^n) do not make full use of the capacity. Inhomogeneous operations (*e.g.* ADD the low 8-bits, XOR bits 10-8, use this constant for bits 15-11, AND in bits 19-16, and OR together the remaining bits) must be decomposed into sets of homogeneous operations.

Example: Average Calculation Consider a windowed average calculation performed on a processor:

$$avg_i = \left(\frac{1}{8}\right) \cdot (x_{i-3} + x_{i-2} + x_{i-1} + x_i + x_{i+1} + x_{i+2} + x_{i+3} + x_{i+4})$$

Figure 4.2 shows a possible inner loop of the windowed average calculation on a standard RISC processor. Assuming one instruction per instruction slot, this sequence takes 9 instruction slots to perform two potentially 32 bit adds – for a total of 128 gate evaluations. The loads, stores, and shifts are all data movement operations and do not contribute to the actual computational task at hand. The branch and increments are control overhead. Assuming this operation is performend on a MIPS-X processor at 1 CPI, we yield a functional density:

$$F_{d_{yield}} = \frac{\text{128 gate evaluations}}{\text{68M}\lambda^2 \times \text{9 cycles} \times \text{50ns}} = 4.2 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

Example: Parity Calculation Consider calculating the parity of a 32-bit word.

$$p = d_{31} \otimes d_{30} \otimes \cdots \otimes d_0$$

```
//R1 d input
//R3 parity output
  asr R1,R2,#16    //align half words
  xor R2,R1,R3     //16b xor
  asr R3,R2,#8     //align bytes
  xor R2,R3,R3     //8b xor
  asr R3,R2,#4     //align nibbles
  xor R2,R3,R3     //4b xor
  asr R3,R2,#2     //align 2 bits
  xor R2,R3,R3     //2b xor
  asr R3,R2,#1     //align final bits
  xor R2,R3,R3     //final xor
```

Figure 4.3: Processor Implemention for Parity Computation

Year	Design	Organization	λ	Die Size	λ^2 area	cycle	$\frac{\text{ALU bit ops}}{\lambda^2 \text{s}}$
1990	LIFE-1 [LS90]	2×32	0.75μ	78mm^2	139M	20 ns	23
1993	VIPER [GNAB93]	4×32	0.6μ	12.9mm$\times$9.1mm	326M	40 ns	9.8

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{iALU}}{\frac{Area}{\lambda^2} \times P_{fraction} \times t_{cycle}}$$

Table 4.5: Survey of VLIW Capacity

In 10 operations (See Figure 4.3), the processor can perform the 32b XOR required for the parity calculation – 32 2-input gate evaluations or 11 4-input gate evaluations. Again, assuming a MIPS-X like processor and 1 CPI, we yield:

$$F_{d_{yield}} = \frac{\text{11 gate evaluations}}{68\text{M}\lambda^2 \times \text{10 cycles} \times \text{50ns}} = 0.32 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

4.2 VLIW Processors

Very Long Instruction Word (VLIW) machines are processors with multiple, parallel functional units which are exposed at the architectural level. A single, wide, instruction word controls the function of each functional unit on a cycle-by-cycle basis. Pedagogically, a VLIW processor looks like a processor with multiple, independent functional units. At this level, the VLIW processor does not look characteristically different from the modern superscalar processors included at the end of the processor table.

Table 4.5 summarizes the characteristics of two VLIW processors. With only two datapoints it is not possible to assess general trends. These examples seem to have about $2\times$ the peak capacity

Year	Design	$F_{density}$	$I_{density}$	$D_{desnsity}$
1990	LIFE-1 [LS90]	23	?	?
1993	VIPER [GNAB93]	9.8	0	1.0×10^{-4}

Table 4.6: VLIW Capacity Summary

of processors. To the extent this may be characteristic of VLIW designs, it may arise from the fact that the separate functional units share instruction control and management circuitry more than in superscalar processors.

VLIW processors may fail to achieve their peak for the same reasons as processors. In addition, they may suffer from:

- **Scheduling granularity** – instructions must be statically scheduled in blocks at the VLIW width. When this packing does not match the needs of the application, functional units may go idle.
- **Mismatch in Functional Unit Mix** – some VLIWs have a mix of functional units which perform different operations. If the mix of operations in the application does not match the functional unit mix at a fine-grained level, functional units may go idle, lowering yielded capacity.
- **Data Transfer** – one way VLIWs may achieve a higher functional density than superscalar processors is to segregate the register files, reducing the interconnect required to deliver data from the register file to the functional units and back. Some cycles will be required to transfer data between register files as necessary to allow the various functional units to cooperate on a task.

4.3 Digital Signal Processors (DSPs)

Digital Signal Processors (DSPs) are essentially specialized microprocessors which:

- Integrate a hardwired Multiplier Accumulator (MAC) unit
- Includes specialized datapaths allowing
  - zero overhead loops
  - parallel increment of counters and pointers

The net effect of these additions is generally to increase the percentage of yielded capacity and particularly to increase the yielded capacity on tight loop multiply operations.

Table 4.7 reviews several DSP implementations. On non-multiply operations, the peak performance is generally lower than the processors. For the kinds of operations typical of DSPs, these processors will generally yield much closer to their peak capacity than processors.

Year	**Design**	**Organization**	**Die Size**	λ	λ^2 **area**	**cycle**	$\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$
1985	[WDW+85]	1×16	15.5mm²	1.0μ	16M	100 ns	10
1986	[vMWvW+86] [Gol87]	1×16	88.5mm²	1.0μ	90M	125 ns	1.4
1987	[KNK+87]	1×16	11.5mm×12.9mm	0.65μ	350M	50 ns	0.9
1987	[CBBF87]	1×32	6mm×8.5mm	0.5μ	200M	60 ns	1.3
1989	[PML+89]	1×16	71mm²	0.6μ	200M	100 ns	0.8
1992	[SKYH92]	1×16	9.5mm×10.5mm	0.6μ	275M	50 ns	1.2
1993	[USO+93]	1×16	9.3mm×9.1mm	0.4μ	530M	93 ns	0.33
1995	[NHK95]	1×32	8.5mm×8.5mm	0.25μ	1.2G	10 ns	2.8
1996	[KOT+96]	2×16	10mm×9.7mm	0.25μ	1.6G	25 ns	0.8

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{iALU}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Sizes above include MACs, which generally amount to 5-10% of the total DSP die area.

Table 4.7: Survey of DSP Capacity

Year	**Design**	$F_{density}$	$I_{density}$	$D_{desnsity}$
1985	[WDW+85]	10	0	1.5×10^{-5}
1986	[vMWvW+86]	1.4	0	4.9×10^{-5}
1987	[KNK+87]	0.9	2.9×10^{-6}	9.3×10^{-5}
1987	[CBBF87]	1.3	1.0×10^{-5}	4.1×10^{-5}
1989	[PML+89]	0.8	1.6×10^{-5}	4.1×10^{-5}
1992	[SKYH92]	1.2	1.4×10^{-5}	3.0×10^{-5}
1993	[USO+93]	0.33	1.5×10^{-5}	1.5×10^{-4}
1995	[NHK95]	2.8	8.9×10^{-7}	2.9×10^{-5}
1996	[KOT+96]	0.8	?	?

Table 4.8: DSP Capacity Summary

4.4 Memories

Most general-purpose devices use memories to store instructions and data. A memory can also be used directly to implement complex computational functions. For complicated functions, a memory lookup can often provide high, programmable computational capacity. For just a few examples see [Has87, HT95, RS92, Lev77].

Model We characterize a memory by:

- depth – either in terms of address bits (a) or total number of memory words (2^a)
- width – w, the number of bits read out for each a bits of address put into the memory

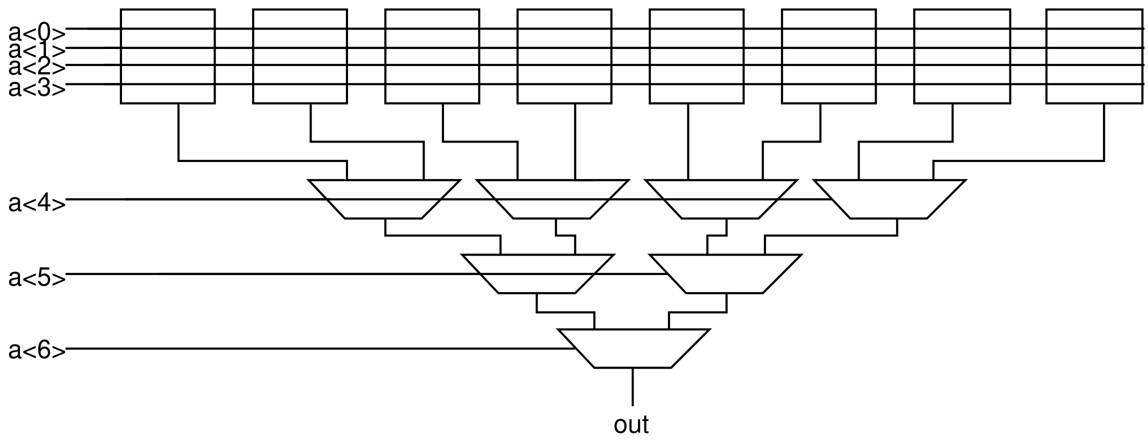


Figure 4.4: Gate Implementation of any Function Computed by 7-input Lookup Table

- t_{cycle} – the minimum time required between successive read operations

Capacity Provided The capacity provided by a memory is highly dependent on the inherent complexity of the logic function being computed. The lower bound is trivially zero since we could program the identify function into a memory.

We can use a counting argument to determine how complicated the functions can get. We start by observing that an a-input, one-output lookup table can implement 2^{2^a} different functions. We then consider how many gates it requires to implement any of these functions. Each gate can be any function of four inputs, so each gate can implement 2^{2^4} functions. A collection of n gates can thus implement at most $\left(2^{2^4}\right)^n$ functions (less due to overcounting). In order to implement any of the 2^{2^a} functions provided by the table, we need at least:

$$
\begin{aligned}
2^{2^a} &\leq \left(2^{2^4}\right)^n = 2^{\left(n \cdot 2^4\right)} \\
2^a &\leq n \cdot 2^4 \\
2^{(a-4)} &\leq n
\end{aligned}
\tag{4.1}
$$

Conversely, by construction, we can show that any function computed by the a input lookup table can be computed with $2^{(a-3)}-1$ gate evaluations. As suggested in Figure 4.4, we can use $2^{(a-4)}$ gates to select the correct functional value based on the low four bits of the address. We then build a binary mux reduction tree to select the final output based on the remaining address bits. This tree requires $2^{(a-4)}-1$ muxes. Together, the $2^{(a-3)}-1$ gates compute any function computable by the a-input lookup table.

An a input by one output table lookup can thus provide between $2^{(a-4)}+1$ and $2^{(a-3)}-1$ gate evaluations per cycle for the most complicated a input functions. Since the bounds are essentially a factor of two apart, we can approximate the peak as $2^{(a-4)}$ gate evaluations per cycle. If the table is

w bits wide, the table provides at most w times as many gate evaluations. Putting all this together, we get:

$$C_{memory\_peak} \approx \frac{w \cdot 2^{(a-4)}}{\text{area} \cdot t_{cycle}} \quad (4.2)$$

Tables 4.9, 4.10 and 4.11 reviews memory implementations, showing the peak functional density for each memory array. For the most complex functions, memories provide the highest capacity of any general-purpose architecture. For less complex operations, however, memories are inefficient, yielding very little of their potential capacity.

For example, an 8-bit add operation with carry output requires 16 gate evaluations. Performed in a $2^{16} \times 9$ memory, such as a 9-bit version of the 64K$\times$18 memory from [SMK+94], this provides only 2.4 gate-evaluations/λ^2s. The inefficiency of the memory-based adder increases with operand size since the number of gate evaluations in an n-bit add increases as $2n$ whereas the memory area increases as $2^{2n} \times (n+1)$.

For all the memories listed, the capacity is based on continuous cycles of random access. In particular, nibble, fast page, or synchronous access in DRAMs is not exploited. For example, [TNK+94] achieves 13,500 gate-evaluations/λ^2s on random access. In sequential access mode, the part can output 18 bits every 8ns. For large sequential access, this means an effective cycle time of 8ns instead of the 48ns quoted – a factor of six improvement in cycle time and capacity. Used in this mode, the peak performance is 81,000 gate-evaluations/λ^2s.

It is also worth noting that, unlike processors, the capacity of memories has increased over time. This is likely due to:

- increased specialization of the fabrication processes to memories – especially the introduction of three-dimensional structures in DRAMs, local interconnect, and high poly film resistors for SRAMs
- increased pipelining of memory access

Modern processors actually dedicate a significant portion of their area to memory. Table 4.12 summarizes the peak capacity the processor can extract by using table lookups in its D-cache. The area used in calculating this capacity is the entire processor for the processors listed in Table 4.2. This peak capacity can be thought of as the peak capacity one could extract from each load operation when using the on-chip D-cache for table lookup operations.

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{gate-evals}}{\lambda^2 s}$
1984	[BDN84]	16K×1	16.3mm^2	1.0μ	16M	35 ns	1800
1984	[SMI+84]	32K×8	6.7mm×8.9mm	0.6μ	164M	46 ns	2200
1984	[YKK+84]	64K×1	4.7mm×6.6mm	0.75μ	55M	25 ns	3000
1984	[SCLB84]	4k×16	32.6mm^2	0.75μ	58M	30 ns	2400
1984	[MSM+84]	8K×8	6.0mm×6.8mm	0.6μ	113M	28 ns	1300
1984	[OKH+84]	2K×8	2.7mm×3.5mm	0.75μ	17M	16 ns	3800
1984	[CH84]	4K×4	11mm^2	1.05μ	10M	18 ns	5700
1984	[MMS+84]	64K×1	3.2mm×6.0mm	0.65μ	45M	20 ns	4600
1985	[YTN+85]	32K×8	5.0mm×9.2mm	0.65μ	110M	45 ns	3400
1985	[SAI+85]	32K×8	49.6mm^2	0.65μ	120M	45 ns	3100
1985	[SGS+85]	8K×8	31.8mm^2	0.75μ	57M	35 ns	2100
1985	[KEK+85]	32K×8	40.7mm^2	0.6μ	110M	55 ns	2600
1986	[CC86]	8K×8	45mm^2	0.75μ	80M	35 ns	1500
1986	[KIK+86]	256k×1	4.5mm×10.6mm	0.5μ	190M	25 ns	3500
1986	[FRV+86]	64K×1	3.4mm×8.9mm	0.75μ	54M	13 ns	5900
1987	[WBS+87]	32K×8	6.8mm×9mm	0.6μ	170M	21 ns	4600
1987	[KTO+87]	128K×8	8mm×13.7mm	0.5μ	440M	35 ns	4300
1987	[WHS+87]	128K×8	5.5mm×14.8mm	0.4μ	510M	34 ns	3800
1987	[MOT+87]	128K×8	6.9mm×15.4mm	0.4μ	660M	25 ns	4000
1987	[GHS+87]	32K×8	9.5mm×7.8mm	0.65μ	175M	40 ns	2300
1988	[SKI+88]	128K×8	7.6mm×12.4mm	0.4μ	590M	44 ns	2500
1988	[WBEK+88]	1M×1	10.6mm×8.5mm	0.35μ	730M	29 ns	3100
1988	[CDH+88]	128K×8	12.2mm×7.7mm	0.35μ	770M	25 ns	3400
1988	[STT+88]	256K×4	7.5mm×12mm	0.35μ	730M	18 ns	5000
1988	[SHU+88]	256K×4	6.2mm×15.2mm	0.4μ	580M	15 ns	7500
1988	[KWA+88]	1M×1	5.5mm×15.7mm	0.35μ	710M	14 ns	6600
1988	[ONN+88]	32K×8	4.4mm×9.5mm	0.4μ	260M	7.5 ns	8400
1989	[VPP+89]	256K×1	3.9mm×9.5mm	0.35μ	300M	14 ns	3900
1989	[MMK+89]	512K×8	7.5mm×17.4mm	0.25μ	2.1G	25 ns	5000
1989	[SIY+89]	1M×1	5.3mm×10.3mm	0.25μ	870M	9 ns	8300
1990	[FPH+90]	256K×1	11.6mm×3.7mm	0.5μ	170M	8 ns	12000
1990	[ASO+90]	4M×1	7.7mm×18.6mm	0.28μ	1.9G	15 ns	9200
1990	[HKM+90]	4M×1	8.4mm×18.0mm	0.3μ	1.7G	20 ns	7800
1990	[SIS+90]	512K×8	7.2mm×16.9mm	0.25μ	1.9G	23 ns	5900
1990	[OHK+90]	512K×8	7.8mm×17.4mm	0.25μ	2.2G	23 ns	5200
1991	[CCS+91]	32K×16	11.1mm×10.1mm	0.4μ	700M	2 ns	23500
1992	[SSN+92]	32K×8	6mm×9mm	0.4μ	340M	200 ns	240
1992	[GOK+92]	2M×8	18.3mm×12.5mm	0.2μ	5.7G	12 ns	15300
1992	[MKS+92]	4M×4	10.4mm×21.5mm	0.2μ	5.6G	15 ns	12500
1992	[SIU+92]	256K×4	4mm×7.4mm	0.15μ	1.3G	7 ns	7200
1993	[SKS+93]	4M×4	9.7mm×21.9mm	0.18μ	6.9G	9 ns	16800
1993	[SUT+93]	4M×4	10.4mm×10.6mm	0.13μ	7.1G	20 ns	7400
1994	[IKM+94]	4M×4	10.3mm×20.9mm	0.2μ	5.4G	30 ns	6500

Table 4.9: Survey of Peak Memory Logic Capacity (SRAM)

Extract from: Reconfigurable Architecture for General Purpose Computing

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{gate-evals}}{\lambda^2 \text{s}}$
1984	[MKS+84]	256K×1	6.3mm×6.3mm	0.75μ	70M	140 ns	1700
1984	[BCH+84]	256K×1	50mm^2	1.0μ	50M	150 ns	2200
1984	[MKM+84]	256K×1	30.2mm^2	0.8μ	47M	116 ns	3000
1984	[KFO84]	256K×1	46.8mm^2	0.6 μ	130M	100 ns	1250
1984	[SNT+84]	128K×8	9.4mm×8.1mm	0.5μ	300M	120 ns	1800
1985	[KCE+85]	1M×1	5.5mm×10.5mm	0.5μ	230M	160 ns	1800
1985	[KFM+85]	1M×1	5mm×13mm	0.6μ	180M	260 ns	1400
1985	[SFO+85]	1M×1	5mm×12.5mm	0.6μ	170M	190 ns	2000
1985	[TJ85]	256K×4	6.0mm×11.4mm	0.6μ	190M	190 ns	1800
1986	[FSO+86]	1M×1	4.4mm×12.3mm	0.6μ	150M	190 ns	2300
1986	[TTS+86]	4M×1	6.2mm×16.0mm	0.4μ	620M	300 ns	1400
1986	[FOW+86]	4M×1	7.8mm×17.5mm	0.5μ	550M	200 ns	2200
1986	[KAI+86]	64K×4	3.1mm×6.9mm	0.6μ	59M	200 ns	1400
1986	[HOW+86]	1M×1	4.8mm×13.2mm	0.6μ	180M	260 ns	1400
1987	[MNA+87]	4M×1	4.9mm×14.9mm	0.4μ	450M	220 ns	2600
1987	[KSE+87]	4M×1	6.4mm×17.4mm	0.4μ	690M	230 ns	1600
1987	[OFW+87]	4M×1	6.9mm×16.1mm	0.45μ	550M	220 ns	2200
1987	[MYM+87]	256K×4	4.7mm×13.8mm	0.5μ	260M	220 ns	1100
1988	[YKMI88]	256K×16	7.5mm×12.7mm	0.4μ	600M	200 ns	2200
1988	[LCwH+88]	128K×4	8.1mm×9.6mm	0.5μ	310M	36 ns	2900
1988	[ANH+88]	16M	8.2mm×17.3mm	0.3μ	1.6G	180 ns	3700
1988	[IYK+88]	16M×1	5.4mm×17.4mm	0.25μ	1.5G	120 ns	5800
1989	[WOI+89]	4M×4	17.5mm×12mm	0.35μ	1.7G	190 ns	3200
1989	[FOS+89]	16M×1	7.9mm×17.4mm	0.3μ	1.5G	150 ns	4600
1989	[CTK+89]	16M×1	8.0mm×16mm	0.28μ	1.7G	150 ns	4100
1989	[AFM+89]	16M×1	7.7mm×17.5mm	0.25μ	2.2G	120 ns	4100
1989	[CKC+89]	16M×1	8.5mm×18.4mm	0.3μ	1.7G	190 ns	3200
1989	[LBK+89]	256K×4	6.8mm×12.3mm	0.5μ	335M	56 ns	3500
1990	[TTK+90]	16M×1	8.2mm×15.9mm	0.28μ	1.7G	150 ns	4100
1990	[KDK+90]	4M×1	5.6mm×15.2mm	0.35μ	700M	120 ns	3100
1990	[KSB+90]	16M×1	7.8mm×18.1mm	0.25μ	2.3G	150 ns	3100
1991	[NTT+91]	16M×4	9.7mm×20.3mm	0.15μ	8.8G	180 ns	2700
1991	[MMM+91]	64M×1	12.5mm×18.7mm	0.2μ	5.8G	120 ns	6000
1991	[TTU+91]	64M×1	19.9mm×11.3mm	0.2μ	5.6G	120 ns	6200
1991	[OTW+91]	64M×1	9.2mm×19.1mm	0.2μ	4.4G	90 ns	11000
1991	[YNH+91]	4M×16	234mm^2	0.2μ	5.6G	95 ns	7500
1991	[NNO+91]	4M×1	4.8mm×11.1mm	0.3μ	592M	60 ns	7400
1992	[HAH+92]	16M×1	8mm×16.6mm	0.3μ	1.5M	120 ns	6000
1992	[KDK+92]	512K×8	13.2mm×6.4mm	0.4μ	525M	60 ns	8300
1993	[STN+93]	16M×16	13.6mm×24.5mm	0.13μ	21G	60 ns	13100
1993	[KHK+93]	64M×4	14.4mm×33.2mm	0.13μ	30.6G	100 ns	5500
1994	[TNK+94]	1M×18	17.1mm×6.6mm	0.25μ	1.8G	48 ns	13500
1994	[AOT+94]	32M×8	13.3×22.8mm	0.13μ	19.5G	90 ns	9600
1994	[TTT+94]	32M×8	13.2mm×25.9mm	0.13μ	21.9G	56 ns	13700
1995	[SMK+94]	32K×9	1.7mm×5.0mm	0.4μ	53M	50 ns	7000
1995	[SMK+94]	64K×18	2.1mm×4.9mm	0.25μ	165M	40 ns	11200

Table 4.10: Survey of Peak Memory Logic Capacity (DRAM)

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

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{gate-evals}}{\lambda^2 s}$
Pseudostatic							
1984	[KSY+84]	32K×9	55mm^2	1μ	55M	125 ns	2700
1991	[SKK+91]	512K×8	6.5mm×14.2mm	0.4μ	580M	116 ns	3900
Virtually Static							
1986	[NSS+86]	128K×8	6mm×13.8mm	0.5μ	330M	150 ns	1300

Table 4.11: Survey of Peak Memory Logic Capacity (Hybrid)

Year	Design	Ref.	D-Cache $\frac{\text{gate-evaluations}}{\lambda^2 s}$
1984	RISC II	[SKPS84]	0
1984	MIPS	[RPJ+84]	0
1987	MIPS-X	[HHC+87]	0
1987	PA-RISC	[YFJ+87]	0
1990	PA-RISC	[TLB+90]	0
1990	SPARC	[MMN+90]	39
1992	SuperSparc	[ANAB+92]	250
1992	Alpha	[DWA+92]	610
1994	PA-RISC	[RDB+94]	0
1994	MIPS	[SYN+94]	150
1995	PowerPC	[BBB+95]	510
1995	UltraSparc	[CDd+95]	330
1995	SPARC V9	[SPA+95]	0
1995	Alpha	[BAB+95]	580
1996	MIPS	[KDS+96]	170
1996	PA-RISC	[LLNK96]	0
1996	ARM	[MWA+96]	1000
1996	Alpha	[GBB+96]	550

Table 4.12: Survey of Processor On-Chip Memory Capacity

4.5 Field-Programmable Gate Arrays (FPGAs)

Field-Programmable Gate Arrays (FPGAs) are composed of a collection of programmable gates embedded in a programmable interconnect. Programmable gates are often implemented using small lookup tables. The small lookup tables with programmable interconnect allow one to take advantage of the structure inherent in many computations to reduce the amount of memory and space required to implement a function versus the full memory arrays of the previous section. Ultimately, this allows FPGA space required for an application to scale with the complexity of the application rather than scaling exponentially in the manner of pure memories.

Model For pedagogical purposes, we consider an FPGA composed of:

- n, four-input lookup tables (4-LUTs) for gates with an optional flip-flop on the output of each LUT which can be used for pipelining or data storage
- "adequate" programmable interconnect to wire up functions using the 4-LUTs
- a minimum operating cycle time, t_{cycle}, which accounts for the time to travel through one LUT and its associated interconnect.

Year	**Design**	**Organization**	**Size**	λ	λ^2 **area**	**cycle**	$\frac{\text{gate-evals}}{\lambda^2\text{s}}$
1986	Xilinx 2K [CDF+86]	1 CLB (4-LUT)	$693\mu \times 715\mu$	1μ	500K	20 ns	100
1988	Xilinx 3K [Xil89, HDJ+88]	64 CLBs (2×4-LUT/CLB)	5mm×6mm (XC3020 die)	0.6μ	83M	13 ns	120
1991	UTFPGA [CSA+91]	3 4-LUTs	$900\mu\times800\mu$	0.6μ	2M	7 ns	210
1992	Xilinx 4K [Xil94b]	49 CLBs (2×4-LUT/CLB)	4.8mm×4.6mm (XC4005 Quadrant)	0.6μ	61M	7 ns	230
1994	LEGO [Seo94]	4 4-LUTs	$1240\mu\times1184\mu$	0.6μ	4M	4.1 ns	240
1995	DPGA [TEC+95]	16 4-LUTs	$1500\mu\times1750\mu$	0.5μ	10.5M	7 ns†	210
1995	Xilinx 5K [Xil91]	49 CLBs (4×4-LUTS/CLB)	3mm×3.3mm (XC5206 Quadrant)	0.3μ	110M	6 ns	290
1995	Altera Flex 8K [Alt95]	1008 LEs (4-LUT/LE)	8mm×10.5mm (81188A die)	0.3μ	930M	7.5 ns	144
1995	ORCA 2C [ATT95]	256 PLCs (4×4-LUT/PLC)	10mm×9.8mm (ATT2C10 die)	0.3μ	$1.1\text{G}\lambda^2$	7 ns	134

† No context switch – 10 ns cycle for context switch

$$\text{gate-evals}/\lambda^2\text{s} = \frac{N_{4LUT}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Table 4.13: Survey of FPGA Capacity

Year	**Design**	$F_{density}$	$I_{density}$	$D_{desnsity}$
1986	Xilinx 2K [CDF+86]	100	2.0×10^{-6}	2.0×10^{-6}
1988	Xilinx 3K [Xil89, HDJ+88]	120	1.5×10^{-6}	1.5×10^{-6}
1991	UTFPGA [CSA+91]	210	1.5×10^{-6}	1.5×10^{-6}
1992	Xilinx 4K [Xil94b]	230	1.6×10^{-6}	1.6×10^{-6}
1994	LEGO [Seo94]	240	9.8×10^{-7}	9.8×10^{-7}
1995	Xilinx 5K	290	1.8×10^{-6}	1.8×10^{-6}
1995	Altera 8K [Alt95]	144	9.3×10^{-7}	9.3×10^{-7}
1995	ORCA 2C [ATT95]	134	1.1×10^{-6}	1.1×10^{-6}

Table 4.14: FPGA Capacity Summary

Capacity Provided Running at full capacity and minimum operating cycle, the FPGA provides $\frac{n}{t_{cycle}}$ gate evaluations per cycle. Modern FPGAs can hold on the order of 2000 4-LUTs and run at cycle times on the order of 5-10ns. Table 4.13 computes the normalized capacity provided by a few representative FPGAs. From these numbers we see that an FPGAs provide a peak capacity on the order of 200-300 $\frac{\text{gate-evaluations}}{\lambda^2\text{s}}$.

FPGA capacity has not change dramatically over time, but the sample size is small. There is a slight upward trend which is probably representative of the relative youth of the architecture.

This peak, too, is not achievable for every application. Some effects which may prevent an application from achieving this peak include:

- **Limited interconnect** – When the network connectivity is inadequate, all of a device's capacity cannot be used. This may require either that cells buffer and transmit data or simply that cells go unused in the area. Conventional FPGA interconnect routes most applications with over 80% utilization.
- **Pipeline efficiency limits** – In heavily pipelined systems, capacity can be required to pass data across pipeline stages to all of its points of consumption. This transit capacity consumes device capacity without contributing to the evaluation capacity required for the application.
- **Limited ability to pipeline operations** – Some tasks have cyclic dependencies where a result is required before the next round of computation can begin. Unless several, orthogonal tasks are interleaved on the FPGA, the cyclic path length limits the rate at which resource can be reused and, in turn, prevents the application from fully utilizing the FPGAs capacity.
- **Limited I/O Bandwidth** – The I/O cycle time on most FPGAs is higher than the logic cycle time shown in Table 4.13. Data transfer to and from the FPGA may limit the capacity which can actually be applied to a problem.
- **Limited need for this functionality** – If a piece of functionality implemented in an FPGA is not required at the rate and frequency achievable on the FPGA, the FPGA can be employed far below its available capacity.

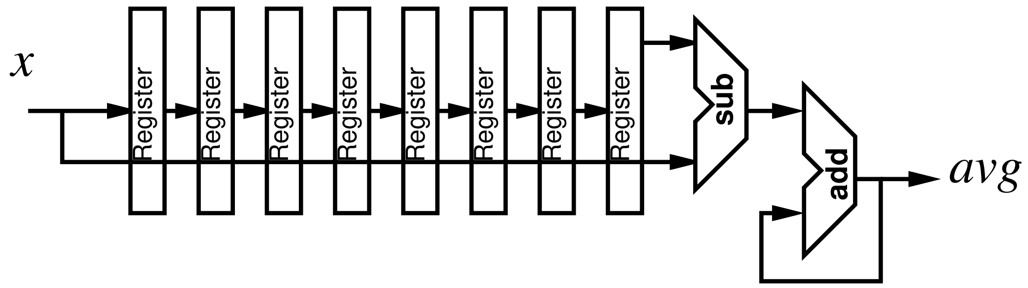


Figure 4.5: Windowed Average – Pipelined FPGA Implementation

- **Need for additional functionality** – When the FPGA cannot hold the required functional diversity for a task and must be reprogrammed in order to complete the task, the device goes partially or entirely unused during the reprogramming cycle.

As one example of pipelining, i/o, and functionality limitations, DEC's Programmable Active Memories ran from 15-33MHz for several application [BRV92]. At these rates, the peak functional density extracted from the XC3090's employed was 13-26 $\frac{\text{gate-evaluations}}{\lambda^2\text{s}}$, only about 10-20% of the potential functional density.

Example: Average Calculation Consider, again, our windowed average calculation:

$$avg_i = \left(\frac{1}{8}\right) \cdot (x_{i-3} + x_{i-2} + x_{i-1} + x_i + x_{i+1} + x_{i+2} + x_{i+3} + x_{i+4})$$

Figure 4.5 shows a pipelined datapath to compute this windowed average. A 16-bit add on an XC4000 part is operates in 21ns. Thus a cycle time of 42ns should be achievable if the x_i's are 28 bits each – if they are 12-bit entities the 21ns cycle would be feasible. With the 32-bit datapath, the impementation requires:

- 8 pipeline registers of 16 CLBs each (128 CLBs)
- 1 adder of 17 CLBs
- 1 subtractor of 17 CLBs

$$F_{d_{yield}} = \frac{128 \text{ gate evaluations}}{1.25\text{M}\lambda^2 \times 162 \times 42\text{ns}} = 15 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

The cycle time can easily be cut in half by pipelining the two halves of each 32b operation, effectively doubling yielded functional density.

Example: Parity Calculation Consider also the FPGA implementation of the 32-bit parity calculation.

$$p = d_{31} \otimes d_{30} \otimes \cdots \otimes d_0$$

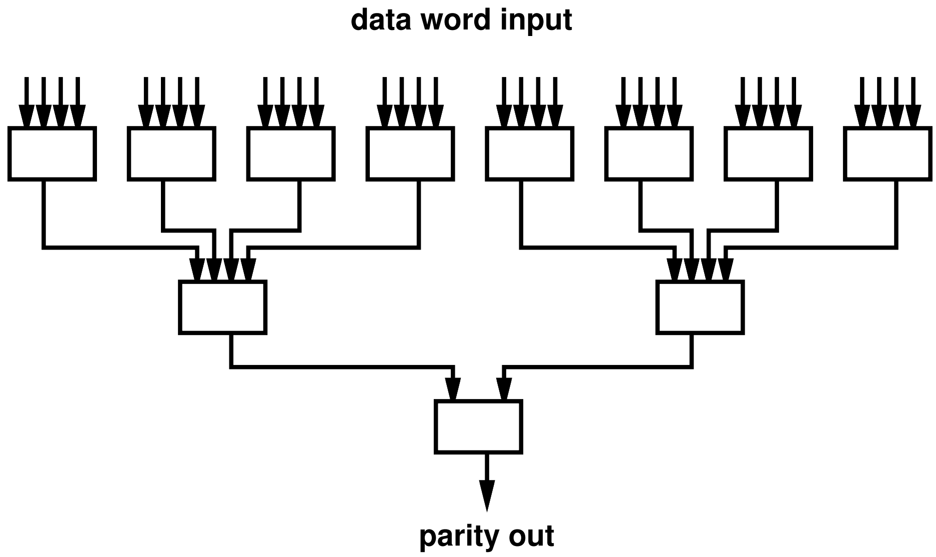


Figure 4.6: 32-bit Parity – 4-LUT Implementation

The FPGA can build the 11 gate parity reduction (See Figure 4.6). The path is three gates long. At $\approx$7ns/gate, the unpipelined version operates in roughly 21ns.

$$F_{d_{yield}} = \frac{\text{11gate evaluations}}{1.25\text{M}\lambda^2 \times 11 \times 21\text{ns}} = 38 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

Pipelining the XOR-reduction, we can reduce the cycle time and increase yield. If we pipeline at the gate level and assume that we can only cycle the part at a 10ns cycle due to clocking limitations, we yield:

$$F_{yield_d} = \frac{\text{11 gate evaluations}}{1.25\text{M}\lambda^2 \times 11 \times 10\text{ns}} = 80 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

4.6 Vector and SIMD Processors

Single-Instruction, Multiple-Data (SIMD) machines are composed of a number of processing elements performing identical operations on different data items. Vector processors perform identical operations on a linear ensemble of data items. At a pedagogical level vector processors are essentially SIMD processors, though in practice the two architectures have traditionally been optimized for different usage scenarios.

Model For pedagogical purposes, we consider a SIMD/Vector array composed of:

- n processing elements (or vector units) all of which perform the same operation on each cycle
- w-bits wide processing element

Year	**Design**	**Organization**	**Size**	λ	λ^2 **area**	**cycle**	$\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$
1987	DEC MP [Gro87]	32×4	10.4mm×9.4 mm	1μ	98M	100 ns	13
1990	MP1 [Nic90]	32×4	11.6mm×9.5mm	0.8μ	170M	70 ns	11
1990	SLAP [FHR94]	4×16	7.9mm×9.2mm	1μ	73M	100 ns	8.8
1990	BLITZEN [HBD94]	128×1	11mm×11.7mm	0.5μ	514M	50 ns	5
1993	MP2 [KT93]	32×32	14mm×14mm	0.5μ	780M	80 ns	16
1994	IMAP [YKF+94]	64×8	15.5mm×15.6mm	0.28μ	3G	25 ns	6.6
1995	MIT Abacus [BSV+95]	1000 PEs (2 3-LUTs/PE)	6.5mm×7.3mm	0.5μ	190M	8 ns	660
1995	MGAP-2 [GOI95]	2×2	0.4mm^2	0.4μ	2.5M	10 ns	160
1996	Sony [KHN+96]	4320 PEs	15.1mm×15mm	0.2μ	5.7G	20 ns	38
1996	PIP [AKY+96]	128×8	18.8mm×16.7mm	0.19μ	8.7G	33 ns	3.6

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{PE}}{\frac{Area}{\lambda^2} \times P_{fraction} \times t_{cycle}}$$

Table 4.15: Survey of SIMD Processor Capacity

Year	**Design**	$F_{density}$	$I_{density}$	$D_{desnsity}$
1987	DEC MP [Gro87]	13	0	3.4×10^{-4}
1990	MP1 [Nic90]	11	0	2.4×10^{-4}
1990	SLAP [FHR94]	8.8	0	?
1990	BLITZEN [HBD94]	5	0	2.5×10^{-4}
1993	MP2 [KT93]	5.4	0	5.2×10^{-5}
1994	IMAP [YKF+94]	6.6	0	6.7×10^{-4}
1995	MIT Abacus [BSV+95]	660	0	3.0×10^{-4}
1995	MGAP-2 [GOI95]	160	0	2.6×10^{-5}
1996	Sony [KHN+96]	38	7.4×10^{-7}	2.0×10^{-4}
1996	PIP [AKY+96]	3.6	0	1.9×10^{-3}

Table 4.16: SIMD Processor Capacity Summary

- instruction control and distribution logic
- a minimum operating cycle time, t_{cycle} – the rate at which new instructions can be initiated in the array

Capacity Provided The SIMD/Vector array provides provides a peak of $\frac{n\times w}{t_{cycle}}$ ALU bit operations per cycle or $\frac{2n\times w}{t_{cycle}}$ gate-evaluations per cycle. Abacus, a modern, fine-grained SIMD array, supports 1000 1-bit PEs and can operate at 125MHz. Abacus thus provides 660 $\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$. Table 4.15 computes the normalized capacity provided by several SIMD arrays of varying granularity, and Table 4.17 shows the composition of a modern vector microprocessor.

SIMD/Vector arrays only achieve their peak capacity when every PE/VU is computing a useful

Year	**Design**	**Organization**	**Size**	λ	λ^2 **area**	**cycle**	$\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$
1995	[ABI+95]	1×32+16×32	16.75mm×16.75mm	0.5μ	1.1G	22 ns	22

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_{sd} \times N_{scalar} W_{vd} \times N_{vector\ units}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Table 4.17: Example Vector Processor Capacity

Year	**Design**	$F_{density}$	$I_{density}$	$D_{desnsity}$
1995	[ABI+95]	22	2.3×10^{-7}	1.6×10^{-5}

Table 4.18: Vector Processor Capacity Summary

logic operation on every cycle. Limitations to achieving this peak include:

- **Limited, local interconnect** – PEs are typically connected only to a few neighbors. Every communication operation occupies a PE without providing any gate-evaluation capacity. On SIMD arrays, when data is moved into the array, around in the array, or out of the array, PEs can be occupied for several cycles without performing any logical operations.

- **Inhomogeneous operation** – All PEs are only usefully employed when the same operation is required on every data bit. When this is not the case, many PEs sit idle or perform no useful work. PEs are often masked out of operation in order to perform computations selectively on data bits.

Flynn [Fly72] summarizes some of the limitations associated with SIMD processing.

Example: Average Calculation Returning to our windowed average calculation:

$$avg_i = \left(\frac{1}{8}\right) \cdot (x_{i-3} + x_{i-2} + x_{i-1} + x_i + x_{i+1} + x_{i+2} + x_{i+3} + x_{i+4})$$

Here, we assume the data is resident in PE memory on the array. It could have been loaded via a background load operation during a previous operation if it started off chip. Groups of 32 PEs are assigned to each word. To perform the average we shift the target data across the array the 8 times and accumulate at each group of 32 PEs as shown in Figure 4.7. The average takes a total of 30 cycles on 32 PEs to perform what we determined earlier to be 128 gate evaluations:

$$F_{d_{yield}} = \frac{128 \text{ gate evaluations}}{0.19\text{M}\lambda^2 \times 32 \times 30 \text{ cycles} \times 8\text{ns}} = 89 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

Operation	**cycles**
initialize result with local value	1 (assumed)
shift	1
accumulate	3
shift	1
accumulate	3
shift	1
accumulate	3
shift	1
accumulate	3
shift	1
accumulate	3
shift	1
accumulate	3
shift	1
accumulate	3
store result	1 (assumed)

Figure 4.7: Abacus (SIMD) Implementation of Windowed Average

Example: Parity Calculation Consider also the Abacus implementation of the 32-bit parity calculation.

$$p = d_{31} \otimes d_{30} \otimes \cdots \otimes d_0$$

The SIMD array can perform a series 31 bitwise shift and **xor** operations to effect an XOR-scan. The **xor** can be folded into the shift such that scan operation only takes 31 cycles for the shift-XOR plus one to configure the operation. At the end of the scan operation, the partiy result is in the high (or low) processor of each 32-bit word.

$$F_{d_{yield}} = \frac{11 \text{ gate evaluations}}{0.19\text{M}\lambda^2 \times 32 \times 32 \text{ cycles} \times 8\text{ns}} = 14.5 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

The data memory can be preloaded with a sequence of bypass operations to allow faster accumulation. The scan can then be performed in $\log_2(32) = 5$ operations, where each operation is 3 cycles long.

$$F_{d_{yield}} = \frac{11 \text{ gate evaluations}}{0.19\text{M}\lambda^2 \times 32 \times 15 \text{ cycles} \times 8\text{ns}} = 31 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

4.7 Multimedia Processors

Multimedia processors are a recent hybrid of microprocessors, DSP, and Vector/SIMD processors. Aimed at processing video, graphics, and sound, these processors support efficient operation on data of various grain sizes by segmenting up their wide-word ALUs to provide SIMD parallel

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{ALU bit ops}}{\lambda^2 \text{s}}$
1995	[Sla95]	128 bits	290mm^2	0.25μ	4.6G	3.3 ns	8
1995	[Sla95]	128 bits	100mm^2	0.25$\mu^\dagger$	1.6G	1 ns	80
1996	[Eps95, TNH$^+$96]	4×72	12.8mm×14mm	0.25μ	2.9G	16 ns	6.3

† Experimental BiCMOS process

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_{sd} \times N_{scalar} W_{vd} \times N_{vector\ units}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Table 4.19: Multimedia Processor Capacity

Year	Design	$F_{density}$	$I_{density}$	$D_{desnsity}$
1995	[Sla95]	8	1.8×10^{-6}	5.6×10^{-5}
1995	[Sla95]	80	5.1×10^{-6}	1.6×10^{-4}
1996	[Eps95, TNH$^+$96]	6.3	2.2-8.9$\times10^{-8}$	0.96-1.2$\times10^{-5}$

Table 4.20: Summary of Multimedia Processor Capacity

operation on the bytes within the word. This segmentation combats the increasing inefficiency associated with processing small data values on wide-word processors.

From Table 4.19, we see the CMOS multimedia processor have the same peak functional density as processors. The major difference is that the segmentation allows these processor to operate on 16-bit and byte-wide data without discarding a factor of 4-8× in performance. Of course, this is true only as long as these finer-grained operations can be performed efficiently in a SIMD manner.

The BiCMOS multimedia processor promised by MicroUnity would have a significantly higher performance density by exploiting a novel process. The comparison between their architecture in CMOS and BiCMOS makes it clear that this functional density advantage comes primarily from the process and not from the architecture.

4.8 Multiple Context FPGAs

Like FPGAs, multicontext FPGAs are composed of a collection of programmable gates embedded in a programmable interconnect. Unlike FPGAs, multicontext devices store several configurations for the logic and the interconnect on the chip. The additional area for the extra contexts decreases functional density, but it increases functional diversity by allowing each LUT element to perform several different functions.

Table 4.21 summarizes the capacities of some experimental, multiple-context FPGAs. Like FPGAs, these devices may suffer from limited interconnect or application pipelining limits. The additional context memory makes them less susceptible to functionality limits than traditional components. Chapter 10 details the usage of multicontext devices including their relative capacity yield compared to single context devices.

Year	Design	Composition	Per Cycle	Size	λ	λ^2 area	cycle	$\frac{\text{gate-evals}}{\lambda^2 \text{s}}$
1995	VEGA [JL95]	2048 4-LUT	1	144mm^2 (1 PE)	0.6μ	400M	10 ns	0.25
1995	DPGA [TEC$^+$95]	64 4-LUTs	16	1500$\mu$$\times$1750$\mu$ (subarray)	0.5μ	10.5M	10 ns	150
1996	TSFPGA [CD96]	64 4-LUTs	2-8	1.1mm$\times$1.2mm (subarray)	0.25μ	21M	5 ns	19-76

$$\text{gate-evals}/\lambda^2\text{s} = \frac{N_{4LUT}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Table 4.21: Survey of Multi-Context FPGA Capacity

Year	Design	$F_{density}$	$I_{density}$	$D_{desnsity}$
1995	VEGA [JL95]	0.25	5.1×10^{-6}	5.1×10^{-6}
1995	DPGA [TEC$^+$95]	150	6.1×10^{-6}	1.5×10^{-6}
1996	TSFPGA	19-76	3.0×10^{-6}	$3\text{-}12\times10^{-6}$

Table 4.22: Multi-Context FPGA Capacity Summary

Year	Reference	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{ALU bit ops}}{\lambda^2 \text{s}}$
1983	[LRSS84]	1$\times$16	6mm$\times$6mm	2.0μ	9M	$\approx$4$\times$140 ns	3
1991	[D$^+$92]	1$\times$32	150 mm^2	0.5μ	600M	62.5 ns	0.9
1991	[FKS91]	2$\times$32	18.85mm$\times$9.85mm	0.5μ	740M	20 ns	4.3
1992	[Sei92]	1$\times$32	9.25mm$\times$10.0mm	0.6μ	257M	33 ns	3.8

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{PE}}{\frac{Area}{\lambda^2} \times P_{fraction} \times t_{cycle}} \quad (4.3)$$

Table 4.23: Survey of MIMD Processor Capacity

4.9 MIMD Processors

Contemporary MIMD processors have largely been built from collections of microprocessors. As such, the functional density of these multiprocessors is certainly no larger than that of the microprocessors employed for the compute nodes. Since these machines typically require additional components for routing between processor and to connect processors into the routing network, the average functional density is actually much lower.

Table 4.23 samples a few processors which were designed explicitly for multiprocessor implementation. These processor integrate the basic network interface and, in some cases, a portion of the routing network, onto the device. While the sample size is too small to draw any strong

Year	Design	Organization	Size	λ	λ^2 area	cycle	$\frac{\text{ALU bit ops}}{\lambda^2\text{s}}$
1992	[CR92]	8×16	6.8mm×6.7mm (core)	0.6μ	126M	40ns	25
1995	[YR95]	48×16	11.5mm×11.2mm (core)	0.5μ	515M	20ns	75
1996	[MD96]	1×8	1.5mm×1.2mm (PE)	0.25μ	29M	10ns	28

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{PE}}{\frac{Area}{\lambda^2} \times t_{cycle}}$$

Table 4.24: Survey of Reconfigurable ALU Capacity

Year	Design	$F_{density}$	$I_{density}$	$D_{desnsity}$
1992	[CR92]	25	5.1×10^{-7}	1.2×10^{-5}
1995	[YR95]	75	7.5×10^{-7}	8.9×10^{-6}
1996	[MD96]	28	$0.14\text{-}1.1\times10^{-6}$	7.1×10^{-5}

Table 4.25: Survey of Reconfigurable ALU Capacity

conclusions, the highest capacity implementations show only about half the functional density of the microprocessors we reviewed in Section 4.1.

4.10 Reconfigurable ALUs

Reconfigurable ALUs are composed of a collection of coarse-grain ALUs embedded in a programmable interconnect. Their word orientation and limitation to ALU operations distinguishes them from FPGAs.

Model For pedagogical purposes, a reconfigurable ALU contains:

- n, w-bit ALUs
- "adequate" programmable interconnect to wire up functions of the ALUs
- a minimum operating cycle time, t_{cycle}, which accounts for the time to operate in one ALU or traverse the interconnect between ALUs.
- optionally, a small instruction store associated with each ALU

Capacity Provided Running at full capacity and minimum operating cycle, the reconfigurable ALU provides $\frac{n\times w}{t_{cycle}}$ ALU bit operations per cycle. Experimental reconfigurable ALUs achieve roughly 50 ALU bit operations/λ^2s.

Like a processor D-cache, the memory on MATRIX can be used as a large lookup table. Using the MATRIX 256×8 memory for function lookup, MATRIX can achieve up to 440 4-LUT gate-evaluations/λ^2s.

Like processor, reconfigurable ALUs may suffer lower yield due to:

- **Mismatched grain-size and limited ALU control** – When fine-grain operations are required, the word-wide interconnect limits which bits may interact with each other. ALU operations are word-wide SIMD making sub-word operations awkward and inefficient.

Unlike processors, the reconfigurable interconnect allows these architectures to avoid much of the data movement overhead necessary on processors. Like FPGAs, pipelining, interconnect, and functionality limits may prevent full utilization.

Example: Average Calculation Returning to our windowed average calculation:

$$avg_i = \left(\frac{1}{8}\right) \cdot (x_{i-3} + x_{i-2} + x_{i-1} + x_i + x_{i+1} + x_{i+2} + x_{i+3} + x_{i+4})$$

Figure 4.8 shows a pipelined datapath to compute this windowed average on MATRIX. In this scheme, 4 BFUs (See Figure 1.4 and Chapter 13) are used to serve as an 8 value delay register, and 4 are used to perform the addition and subtration. Two cycles are required for each result so that a single datapath can be used for the add and subtract and so that the single memory can provide one read and one write cycle. The implementation yields:

$$F_{d_{yield}} = \frac{128 \text{ gate evaluations}}{28.8\text{M}\lambda^2 \times 9 \times 2 \text{ cycles} \times 10\text{ns}} = 25 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

Example: Parity Calculation Consider also the MATRIX implementation of the 32-bit parity calculation.

$$p = d_{31} \otimes d_{30} \otimes \cdots \otimes d_0$$

The most straightforward implemenation, uses the memory as an 8-LUT to calculate the parity of 8 bit data chunks. A total of 5 such chunks will perform the entire calculation (See Figure 4.9). Assuming pipelined operation of the first four and final reductions:

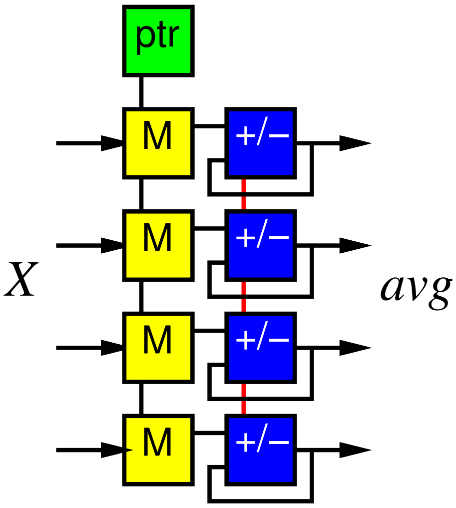
$$F_{yield_d} = \frac{11 \text{ gate evaluations}}{28.8\text{M}\lambda^2 \times 5 \times 10\text{ns}} = 7.6 \left(\frac{\text{gate evaluations}}{\lambda^2\text{s}}\right)$$

4.11 Summary

Table 4.26 summarizes the observed computational densities for the general-purpose architecture classes reviewed in this section.

Memories provide the highest programmable capacity of any of the devices reviewed. However, they only yield this capacity on the most complex functions – those whose complexity is, in fact, exponential in the number of input bits. The capacity they provide is not *robust* in the face of less complex tasks.

Reconfigurable devices provide the highest general-purpose capacity which can be deployed to application needs. Unlike memories capacity consumption scales along with problem complexity. Their peak performance is $10\times$ all non-reconfigurable architectures, with the exception of large,



ptr	**memory**	**calculate**
	read <ptr>	avg←avg+new
increment ptr modulo 8	write <ptr>←new	avg←avg-old

Figure 4.8: Windowed Average – MATRIX Implementation

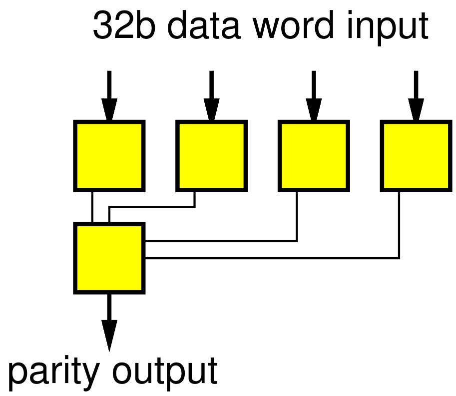


Figure 4.9: 32-bit Parity – MATRIX Implementation

Architecture	$\frac{\text{gate-evals}}{\lambda^2 \text{s}}$	**Limitations**
Memory	1500-15000	most complicated functions only
SIMD (1000's of PEs)	60-1200	highly homogenous computations only
FPGA	100-300	regular, highly pipelined, computations only
RALUs	50-150	semi-regular, word-wide operations
Few context DPGAs	30-150	semi-regular operations
Vector/VLIW	20-50	coarse-grain, semi-regular operations
SIMD (100's of PEs)	10-30	homogenous computations
Processors/Multimedia	4-20	word-wide operations
DSPs	2-20	word-wide operations
Highly multicontext FPGAs	0.25	

Table 4.26: General-Purpose Computational Capacity Summary

well engineered SIMD arrays. Fine-grained devices, such as FPGAs, are robust to grain-size variation, as well. Reconfigurable architectures are not, however, robust to tasks with functional diversity larger than the aggregate device capacity. Multicontext devices, such as the DPGA, sacrifice a portion of the peak FPGA capacity density to partially mitigate this problem – providing support for much higher on chip functional diversity.

Large SIMD or vector arrays have high peak performance because they ammortize a single stream of instruction control, bandwidth, and memory among a large number of active computing elements. They handle high diversity with the ability to issue a new instruction on each cycle. However, they require very large granularity operations in order to efficiently use the computational resources in the array.

Processors are robust to high functional diversity, but achieve this robustness at a large cost in available capacity – $10\times$ below reconfigurable devices. They also give up fine-grain control of operations, creating a potential for another $10\times$ loss in performance when irregular, fine-grained operations are required. Vector and VLIW structures provide slightly higher capacity density for very stylized usage patterns, but are less robust to tasks which deviate from their stylized control paradigm.

Here we see distinctions in granularity, operation diversity, and yieldable capacity. The key issues we used to classify architectures was the way the devices store and distribute instructions to processing elements. Characterizing instructions and interconnect issues with a focus on *RP*-space is the goal of Part III.

In this segment we review hardwired, programmable, and configurable multiply implementations. The custom multiplier implementations show us the functional density achievable by custom hardware on its intended task for comparison with the general-purpose structures reviewed in Chapter 4.

We use the multiply operation for this comparison because it is relatively simple and important to many computing tasks including signal processing. Because of its importance and regularity, it has received much attention over the years including many, high quality, custom implementations. Multiply is probably one of the first computational operators to be implemented in most new VLSI processes. *Considering the amount of attention given to custom multiply implementations, the comparison between custom multiplies and configurable implementations represents an upper bound on the performance disparity between custom and configurable implementations.* Few functions, if any, should show a larger disparity, and most show a significantly smaller disparity. Multiply is also interesting since it is the first piece of custom logic added to "general-purpose" processors.

In this section we use a domain specific metric for functional capacity, the multiply bit operation (MPY bit op). To allow us to compare multiplies of various sizes, we assume each $n \times m$ multiply requires $n \cdot m \cdot$ MPY bit ops. As such, we metric multiply functional density in MPY Bit Ops$/\lambda^2$s and compute it as shown in Equation 5.1.

$$D_{mpy} = \frac{m \times n}{\frac{Area}{\lambda^2} \times t_{cycle}} \tag{5.1}$$

An $n \times n$ multiply can be done in less than $O(n^2)$ operations (see for example [Knu81]), but, for the multiplies reviewed here, all of the circuits and algorithms do scale as $O(n^2)$.

5.1 Custom Multipliers

Table 5.1 summarizes the performance of numerous custom multipliers according to Equation 5.1. Implementations range from sub 1000 to almost 9000 MPY Bit Ops$/\lambda^2$s with 2000-4000 MPY Bit Ops$/\lambda^2$s representing the range of typical, high-performance, custom multipliers. Like processors there is no clear trend for improvement with time or decreasing feature size. The latest designs, if anything, show a tendency to emphasize latency over throughput resulting in lower functional density.

5.2 Semicustom Multipliers

Table 5.2 shows a few, sample, semicustom multiplier implementations. At 330 and 560 MPY Bit Ops$/\lambda^2$s, the gate array and standard cell implementations provide a factor of 5-10 less functional density than the custom implementations.

Extract from: Reconfigurable Architecture for General Purpose Computing

Year	**Design**	**Organization**	**Size**	λ	λ^2 **area**	**cycle**	$\frac{\text{MPY bit ops}}{\lambda^2 s}$
1984	[LGC84]	8×8	1.25mm^2	1.5μ	0.56M	120 ns	960
		16×16	5mm^2	1.5μ	2.2M	120 ns	960
1984	[UKY84]	24×24	3.8mm×3.8mm	1.0μ	14.4M	71 ns	560
1985	[GGA+85]	32×32	5.3mm×5.7mm	1.0μ	30M	56 ns	600
1985	[HFML85]	16×16	1.7 mm×1.7mm	0.75μ	5.1M	40 ns	1250
1986	[NSLKE86]	8×8	1.5mm×0.4mm	0.5μ	2.4M	3 ns	8900
1987	[LGS87]	8×8	0.61mm×0.58mm	0.5μ	1.4M	9.5 ns	4800
1988	[KKHY88]	32×32	3.2mm×5.2mm	1.0μ	17M	59 ns	1000
1988	[SJ88]	4×4	1.37mm^2	1.0μ	1.4M	16 ns	730
1989	[SH89]	64×64	3.8mm×6.5mm	0.8μ	39M	47 ns	2300
1989	[SLM+89]	16×16	1.55mm×1.44mm	0.25μ	36M	6.75 ns	1100
1990	[ADD90]	32×32	9880 mil^2	0.5μ	25.5M	35 ns	1150
		24×16	3819 mil^2	0.5μ	9.9M	28 ns	1400
		16×16	2888 mil^2	0.5μ	7.5M	22 ns	1600
1990	[YYN+90]	16×16	1.3mm×3.1mm	0.25μ	64M	3.8 ns	1000
1990	[SA90]	56×56	3.4mm×6.5mm	0.5μ	88M	30 ns	1200
1991	[MNH+91]	54×54	3.62mm×3.45mm	0.25μ	200M	10 ns	1500
1992	[FHT+92]	24×24	3.42mm×4.5mm	0.6μ	43M	30 ns	450
1992	[GSNS92]	54×54	3.36mm×3.85mm	0.4μ	81M	13 ns	2800
1993	[LS93]	12×12	2.5mm×3.7mm	0.5μ	37M	5 ns	780
1993	[SV93]	8×8	1.5mm×1.4mm	0.8μ	3.3M	4.3 ns	4500
1994	[KHANW94]	11×11	1.53mm^2	1.0μ	1.5M	22 ns	3600
		11×16	0.9mm^2	0.6μ	2.5M	19 ns	3700
1995	[OSS+95]	54×54	3.77mm×3.41mm	0.125μ	823M	4.4 ns	810
1995	[IIF+95]	16×16	0.77mm×0.72mm	0.125μ	35M	10 ns	730
1996	[HKKM96]	54×54	17mm^2	0.15μ	760M	2.5 ns	1500
1996	[LE96]	4×4	0.224mm^2	0.5μ	0.90M	17 ns	1100
1996	[MNS+96]	54×54	3.1mm×3.1mm	0.25μ	150M	8.8 ns	2200
1996	[MYO+96]	32×32	2.35mm^2	0.2μ	59M	18 ns	980

Table 5.1: Survey of Multiplier Capacity

Year	**Design**	**Organization**	**Size**	λ	λ^2 **area**	**cycle**	$\frac{\text{MPY bit ops}}{\lambda^2 s}$
Gate Array							
1987	[BMNW87]	16×16	14.4mm^2	0.75μ	26M	30 ns	330
Standard Cell							
1993	[FA93]	16×16	3mm^2	0.63μ	7.7M	60 ns	560
Layout Generator							
1993	[FA93]	16×16	1mm^2	0.63μ	2.6M	40 ns	2500

Table 5.2: Sample Semi-Custom Multiplier Capacity

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

Architecture	Reference	Multiply Op	area and time	$\frac{\text{MPY bit ops}}{\lambda^2 s}$
Processor				
(basic ALU ops)	[SKPS84]	8×8	41 instructions	0.3
		16×16	81 instructions	0.7
(w/mstep)	[Cho89]	8×8	10 instructions	2
		16×16	18 instructions	4
		32×32	34 instructions	9
(w/ booth step)	[RPJ+84]	16×16	9 instructions	4
(w/ multiplier)	[BBB+95]	64×64	2 per cycle	250
DSP (16×16 MAC)	[WDW+85]	16×16	1 cycle	165
DSP (16×16 MAC)	[Gol87]	16×16	1 cycle	23
DSP (16×16 MAC)	[PML+89]	16×16	1 cycle	13
DSP (2×16×16 MAC)	[USO+93]	16×16	0.5 cycles	10
DSP (32×32 MAC)	[NHK95]	32×32	1 cycles	89
Memory	[SMK+94]	8×8	1 64K×18 block	10
	[SKS+93]	11×11	6 ICs	0.3
SIMD	[BSV+95]	8×8	8 PEs, 66 cycles	80
		16×16	16 PEs, 126 cycles	84
		32×32	32 PEs, 235 cycles	90
(ALU only)	[YKF+94]	8×8	1 PE, 40 cycles	1.3
(w/ lookup)		8×8	1 PE, 11 cycles	4.8
Vector				
(w/ 16×16 mpy)	[ABI+95]	16×16	8 per cycle	82
FPGA	[ATT94]	8×8	27 PLCs, 19ns	30
	[Alt96]	8×8	164 LEs, 49ns	8.6
	[LE94]	8×8	66 CLBs, 102ns	7.6
		16×16	102 CLBs, 152ns	13
		32×32	174 CLBs, 254ns	18.5
		200×200	930 CLBs, 1320ns	26
	[ID95]	16×16	316 CLBs, 26ns	25
	[ID95]	16×16	88 CLBs, 120ns	19
PADDI2	[YR95]	16×8	4 PEs, 50MHz	150
MATRIX		8×8	1 BFU, 20 ns	110
		16×16	6 BFU, 20 ns	74

Table 5.3: Survey of Programmable Multiply Capacity

5.3 General-Purpose Multiply Implementations

For comparison, Table 5.3 summarizes the capacity density of several configurable and programmable implementations. Processors without specialized multiply support show a factor of 10,000× lower performance density than hardwired multipliers. Processors, with multiply or booth step operations have only a factor of 1,000× lower performance density. FPGAs are a factor of 100-300× less dense than custom hardware. Processors, DSPs, and reconfigurable ALUs with integrated multipliers are only a factor of 10-20× lower in performance density. Figure 5.1 shows these basic relationships.

//R1,R2 hold inputs		
	ADD R0,R0,R3	
//repeated for number of bits in R2 input		
	AND R1,#1,R4	*//mask low bit*
	JUMP lequ,ZBITn	*//skip add if zero*
	SLL R1,#1,R1	*//delayed branch slot*
	ADD R3,R1,R3	*//add in scaled term*
ZBITn:	SRA R2,#1,R2	*//scale for next add*
//result in R3		

Table 5.4: Multiply Using Standard ALU Operations

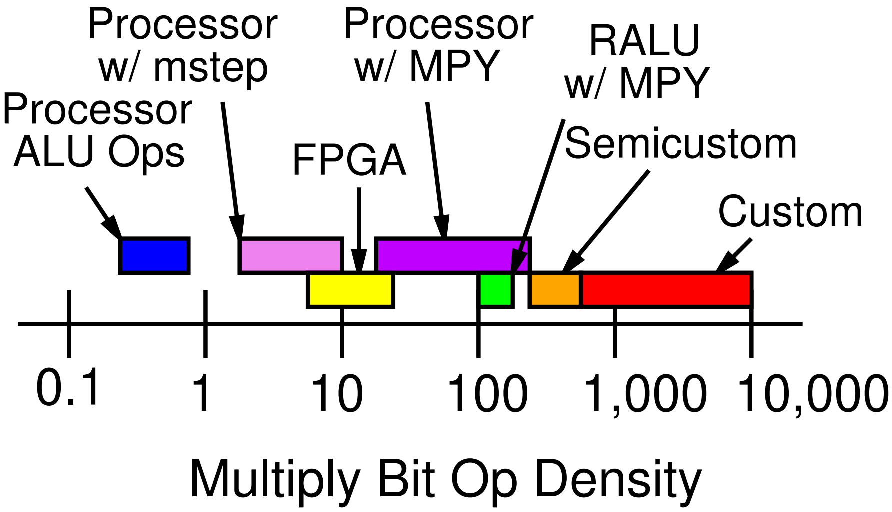


Figure 5.1: Comparison of Programmable and Custom Multiply Functional Densities

5.4 Hardwired Functional Units in "General-Purpose Devices"

One thing we note from Table 5.3 is that processors with integrated multipliers provide roughly 10% of the performance density of a custom multiplier. This comes about simply by dedicating $\approx$10% of the processor real-estate to hold a custom multiplier. Because of the importance of the multiply function in many applications and the 100-1,000$\times$ performance density differential achievable by setting aside this 10%, many processors and all DSPs augment the general-purpose core with a hardwired multiplier. Custom multiply and floating-point logic are the two main piece of custom logic which have been regularly integrated onto conventional "general-purpose" computing devices for this reason.

Structure	**Reference**	64×64	54×54	32×32	16×16	8×8	4×4
Custom 64×64	[SH89]	2300	1600	560	140	35	9
Custom 54×54	[GSNS92]		2800	970	240	60	15
Processor							
(w/multiplier)	[BBB+95]	250	180	63	16	4	1
(w/mstep)	[Cho89]			9	4	2	0.8
(ALU Ops)	[SKPS84]				0.7	0.3	0.2
FPGA							
	[LE94]	23	21	19	13	8	3.5

Table 5.5: Yielded Multiply Capacity as a Function of Granularity

Architecture	**Reference**	**Multiply Op**	**area and time**	$\frac{\text{MPY bit ops}}{\lambda^2 s}$
Processor	[Cho89]	8×8	8 instructions	2
		16×16	16 instructions	5
Memory	[SMK+94]	16×16	2 64K×18 block	19
	[SKS+93]	22×22	11 ICs	0.7
FPGA	[Cha93]	8×8	22 CLBs, 25 ns	93
		16×16	84 CLBs, 40 ns	61

Table 5.6: Survey of Specialized Programmable Multiply Capacity

5.5 Multiplication Granularity

A custom multiplier is often called upon to perform multiplies for a variety of data sizes. When multiplying operands smaller than the native multiply size, the custom multiplier yields lower multiply functional density than indicated in Table 5.1. Table 5.5 compares the yielded capacity of the various custom and programmable multipliers reviewed above.

5.6 Specialized Multiplication

In many applications, one of the operands in the multiply is a constant – or changing slowly. In these case, the operation complexity is slightly reduced, in general, and may be greatly reduce in particular circumstances. Hardwired, 2-operand, multipliers cannot take advantage of this reduced complexity whereas programmable and configurable devices can. Table 5.6 summarizes the multiply capacity provided on specialized multiplies. For comparison with the previous tables, the multiply capacity density is calculated as if it is performing a full $n \times m$ multiply. It might be more accurate to say the complexity of the problem decreased rather than the density of multiply bit ops increased, but the ratio of the performance density numbers is the same whichever way we view it. Note that the densities shown in Table 5.6 apply for *any* constant operand. Particular operands may admit to much tighter implementations.

5.7 Summary

In general, reconfigurable devices achieve 100-300$\times$ lower capacity density than their custom multiply counterparts. At the same time, they achieve 10-30$\times$ better performance than a processor building a multiply out of ALU operations. For this particular operation, most processors include a specialized multiply-step operation, which brings them closer to parity with the reconfigurable devices, or integrate a custom multiplier, which gives them a 10$\times$ advantage over the reconfigurable devices. Reconfigurable devices which also include custom multiply support achieve about the same multiply density as processor with integrated, custom, multipliers. When large, custom multiplier arrays are used on small data, the gap between the custom devices and the reconfigurable devices narrows. Similarly, when a multiply operand is constant or slowly changing, reconfigurable devices may exploit the reduction in operation complexity to narrow the density gap.

We have already noted that conventional FPGAs are poor at handling a functional diversity which is larger than the aggregate functional capacity provided by a single device (Section 4.5). Handling larger diversity may require reloading the FPGA programming, a slow process for conventional FPGAs. During the reload time, the device goes largely unused. Alternately, a more generic processing unit can be built on top of the FPGA and microsequenced like a processor. In the most extreme case of spatial limitations, we might end up building a processor-like design on top of the FPGA. Table 6.1 summarizes the capacity density provided by several processors which have been built on top of FPGAs.

From Table 6.1, we see that such processors, when optimized for the FPGA, have a peak capacity of about 2 ALU bit operations/λ^2s, or about one fourth the capacity of a custom processor. The architecture for R16 and jr16 are moderately straight RISC processor architectures, and are likely to yield about the same fraction of this capacity as most other RISC processors.

At a 4$\times$ penalty from custom processors, for high diversity operations, one would certainly be better off using, or building, a custom processor. As the commonality in the computational task increases and the area available to the FPGA increases, the FPGA can build more application specialized structures, realizing higher capacity density. This suggests there is a continuum between the most highly diverse functional operations, where FPGAs are 4$\times$ less dense than processors, to the most regular operations, where FPGAs provide 10-100$\times$ more performance density.

It is also interesting to note that the performance density penalty for handling these highly diverse operations on an FPGA is much less than the performance density penalty associated with implementing a multiplication on the FPGA.

With only a 4$\times$ performance density penalty, an FPGA processor is roughly equivalent to a

Year	**Design**	**Organization**	**Design Size**	λ^2 **area**	**cycle**	ALU bit ops/λ^2s
1991	Fliptronics R16 [Fre94]	1$\times$16	150 XC4K CLBs	190M	50 ns	1.7
1994	nP [WHG94]	1$\times$8	40 XC3K CLBs	52M	3$\times$30 ns	1.7
1994	MacDLX [Dur94]	1$\times$32	$\approx$1000 XC4K CLBs	1.2G	500 ns	0.05
1994	jr16 [Gra94]	1$\times$16	$\approx$200 XC4K CLBs	250M	25 ns	2.6
1996	j32 [Gra96]	1$\times$32	$\approx$ 250 XC4K CLBs	310M	63 ns	1.6
1996	Hokie [GHH+96]	1$\times$16	$\approx$ 140 XC4K CLBs	175M	63 ns	1.5

$$\text{ALU Bit Ops}/\lambda^2\text{s} = \frac{W_d \times N_{iALU}}{\frac{Area}{\lambda^2} \times P_{fraction} \times t_{cycle}}$$

Table 6.1: Survey of FPGA-Implemented Processor Capacity

$4\times$ smaller processor. From table 4.2, we have seen aggregate processor capacity increase from 15Mλ^2 in 1984 to 5Gλ^2 in 1995, or about 70% per year. The $4\times$ capacity density thus puts a processor implemented on an FPGA implemented in a modern processes roughly equivalent to a 2.5-3 year old processor. As such, FPGA processors – which can ride the FPGA technology to track technology advances – may be an attractive option for running legacy assembly code.