WHITE PAPER

# Everything You Wanted to Know About SOC Memory*

# * But Were Afraid to Ask

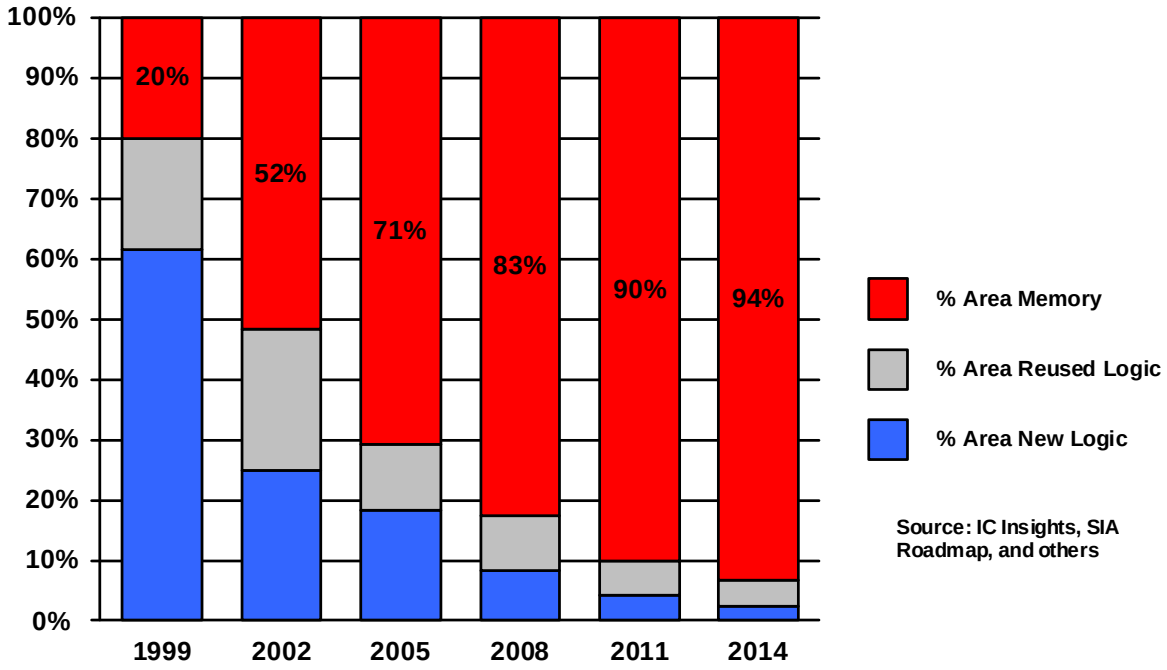
*If you are a member of an SOC design team, or if you manage one, then memory is critically important to you. On today's multicore SOC designs, more on-chip silicon is devoted to memory than to anything else on the chip and yet memory is often added as an afterthought. Don't let that happen to your team.*

This white paper discusses the many alternatives for on-chip and off-chip memory usage that SOC designers must understand to develop successful multicore SOCs. It discusses the essentials of SOC memory organizations for multicore designs, on-chip SRAM and DRAM, local memories and caches, on-chip non-volatile memories, and memory controllers for off-chip memory. It covers the difference between 6T and 4T SRAM designs, the system design ramifications of NAND and NOR Flash ROM, and how DDR2 and DDR3 SDRAMS compare (the differences might surprise you).

## The State of SOC Memory

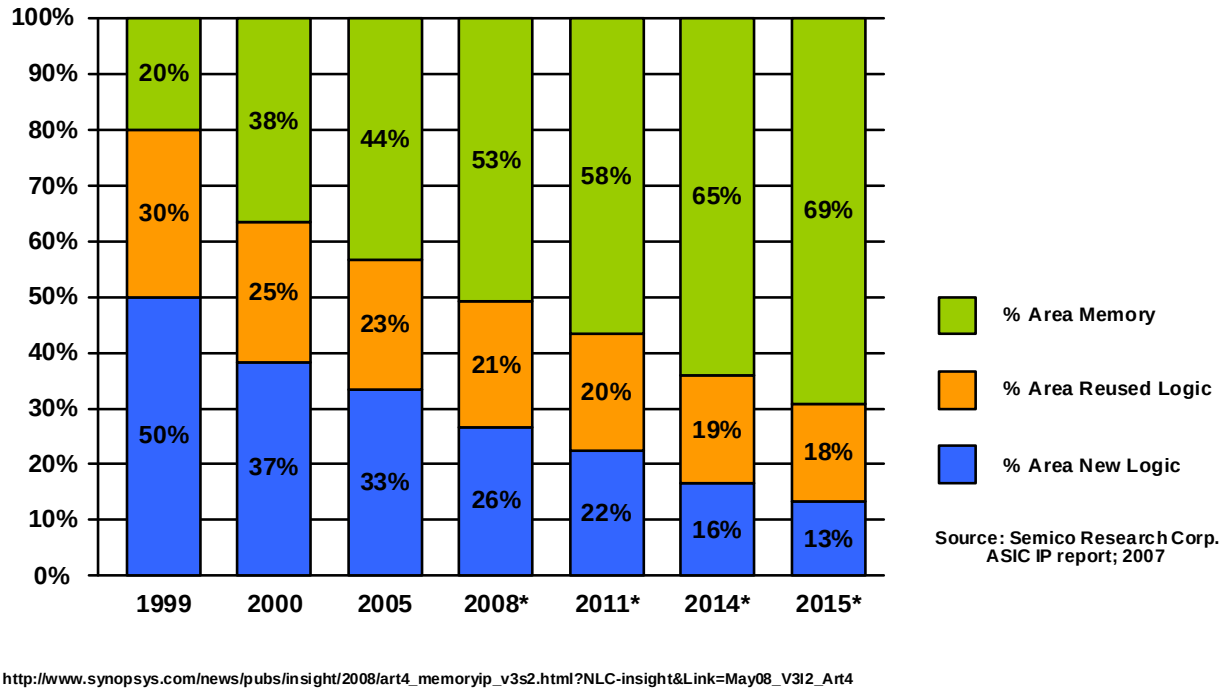
Consider what's happening to the SOC's on-chip memory. Figure 1 shows a 2004 graph predicting that the average SOC would devote more than 80% of its silicon to memory by the year 2008. This percentage was predicted to grow until logic made up less than 10% of an average SOC. Now these figures are just averages and, of course, future predictions are just a guess. This particular guess was synthesized from data provided by IC Insights, the SIA roadmap, and other sources.

Figure 2 shows a more recent and more conservative prediction of memory use in SOCs made by Semico in 2007. The analysts at Semico predicted that only a little more than half of an SOC designed in 2008 would consist of memory and they don't even agree with the prediction shown in Figure 1 with respect to past SOC memory usage.




Doris Keitel-Schulz, Infineon, MPSOC 2004

**Figure 1: Memory Use on ASICs, Prediction #1 from 2004**




http://www.synopsys.com/news/pubs/insight/2008/art4_memoryip_v3s2.html?NLC-insight&Link=May08_V3I2_Art4

**Figure 2: Memory Use on ASICs, Prediction #2 from 2007**

Your SOC will no doubt be different than either of the two predictions shown—but that really isn't important. Whether the total amount of area devoted to memory on your SOC is 50% or 80%, it's still a big part of your chip. There's an increasing amount of data to process in today's applications and that data requires more on-chip memory. How much more is determined during the design of the SOC.

## SOC Evolution

Before discussing where we are today with respect to on-chip SOC memory, let's set the stage by discussing how we got here. Let's briefly review the evolution of the SOC.

Figure 3 depicts the evolution of the SOC. The 1980s saw the real arrival of ASICs—actually gate arrays, which were custom chips that vacuumed up all of the glue logic on a board-level design. Back then, there weren't enough gates on an ASIC to implement a processor, so there were no SOCs. SOCs—by definition—are ASICs that incorporate one or more processors.

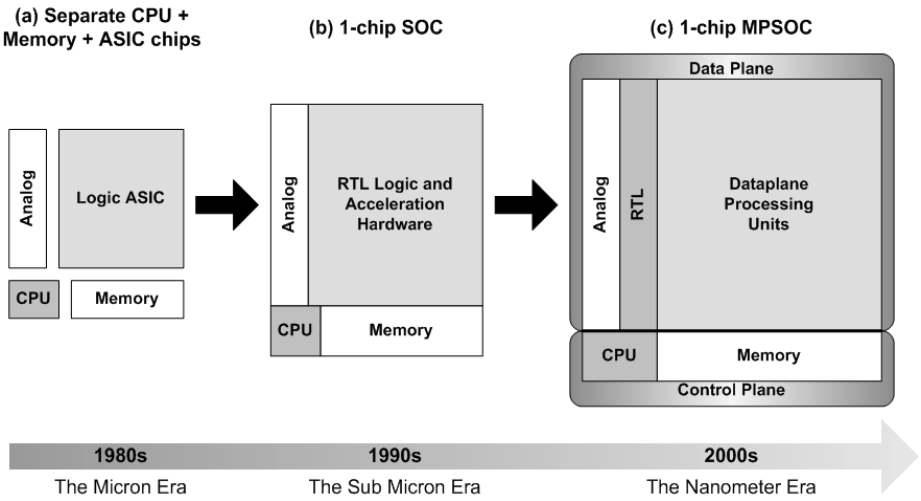



**Figure 3: ASIC and SOC Evolution**

By the mid 1990s, IC manufacturing technology had advanced enough to allow the inclusion of a processor on the standard-cell ASIC along with memory and glue logic. This was the start of the SOC era. System-level design needed no advances at that point because SOC-based systems essentially looked just like earlier single-processor systems implemented at the board level. Architecturally, there was little or no difference.

Thanks to Moore's Law, we're now in the MPSOC era. We can easily put several processors on an SOC, creating an MPSOC. In fact, we put enough processors on a chip now to require some sort of differentiation among the processor types. In other words, one or two on-chip processors take the traditional CPU role on these chips and the other processors perform other sorts of tasks. Designers of networking equipment such as switches and routers coined the term "control plane" to describe the part of the system were the CPUs do their thing. The other part of the chip, the part handling high-speed data, is called the dataplane. For networking applications, the dataplane handles packet traffic. In other chips, the data might be audio or video or some other form of high-speed, streaming data.

### Drilling into the Dataplane

If you drill down into an SOC's dataplane, as illustrated in Figure 4, you see a variety of processing, all taking place simultaneously. Some of this processing includes audio and video encoding/decoding and pre- or post-processing, baseband DSP, security protocol processing. There are of course many other processing types. You might find an RTL block or an actual processor performing the processing in the dataplane. Processors used in the dataplane are often called dataplane processors (DPUs). RTL and specialized DPUs are far more efficient at dataplane processing than are general-purpose processor cores and DSPs. Increasingly, SOC designers are using more DPUs and less RTL.

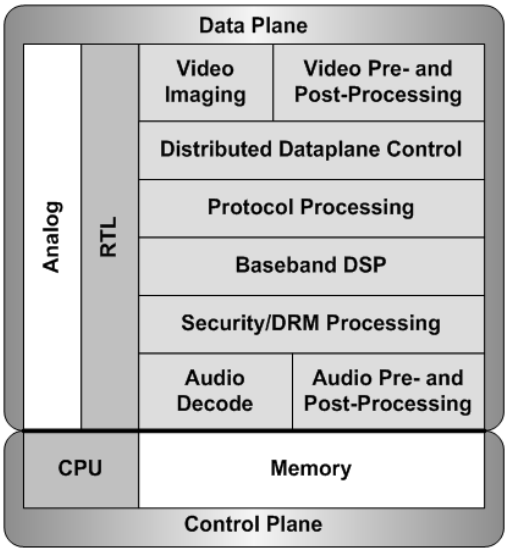



**Figure 4: SOC Dataplane Tasks**

The reason's pretty simple. Dataplane algorithms are generally developed in C on fast, inexpensive PCs by algorithm designers. Once the algorithms are "perfected," which means they're good enough to do the job, they need to be implemented for the design. If you use a dataplane processor, you can go straight from C-based algorithm to implementation with no intermediate translation, and there's no chance for error. If you translate from C to RTL to create task-specific hardware, usually by hand, there's a chance for error in the translation and there's the need to verify the resulting custom hardware, so design risk increases and so does the time required to create the hardware implementation.

Each of the DPUs needs memory. Actually, whether a dataplane function is implemented with a DPU or with custom RTL, the function itself generally requires memory. For example, a video decoder can be implemented with one or more DPUs or as an RTL decoder block. Either way, the function needs memory to hold multiple video frames for processing and the amount of memory needed for each dataplane function depends more on the task being implemented than on whether it's a firmware-driven DPU or an RTL datapath controlled by a hardwired state machine.

### SDRAM: The cheapest RAM bits you can buy

Even though the amount of on-chip memory continues to grow thanks to lithography shrinks, the amount of memory available for processing is often insufficient for a number of reasons. Some systems need a lot of memory and on-chip memory is not the least expensive memory available. The cheapest memory, in terms of cost per bit, comes packaged in a standard SDRAM (synchronous DRAM) chip.

The current cost leader in the SDRAM category is DDR3 memory (DDR4 memory specifications are expected to be released in 2011 with production beginning in 2012).

Whether your design uses DDR2 or DDR3 SDRAM, the SOC needs a special memory controller to talk to off-chip DDR memory. You can design your own DDR memory controller block or you can purchase one as an IP block.

Even one memory controller may not be enough. It's likely you'll need off-chip, nonvolatile storage as well to hold code and data for on-chip use. The most cost effective way to do this today is to use off-chip Flash. You'll need a Flash memory controller to run the off-chip Flash and that memory controller will do different things and talk over a different interface than the SDRAM memory controller.

But before discussing memory controllers, let's look at one of the dataplane processing subsystems built from a dataplane processor and memory. Figure 5 shows an oversimplified diagram of such a subsystem. There are only three components: a processor, a block of memory, and an interconnect bus. How do we know it looks like this?

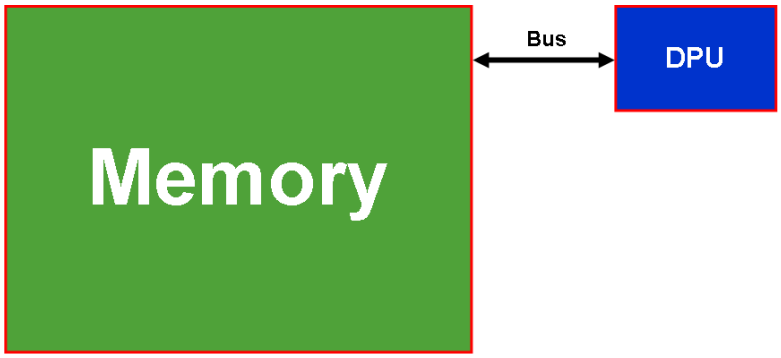



**Figure 5: Simple Processor/Memory Model**

We know because of the chip shown in Figure 6, which is a photo of Intel's 4004, the world's first commercial microprocessor. Intel introduced the 4004 to the world in November 1971. Note that the Intel 4004 is a 16-pin device. Why? Because Intel was a memory company back in 1970, when the 4004 was developed, and it had all the equipment it needed to package chips in 16-pin DIPs. So 16 pins was the most economical package for Intel and the 4004 microprocessor was therefore designed to fit in that package. As a result, the Intel 4004 microprocessor has a 4-bit multiplexed bus. The processor puts the address on the bus in three consecutive nibbles and the data goes out or comes in using a fourth nibble.

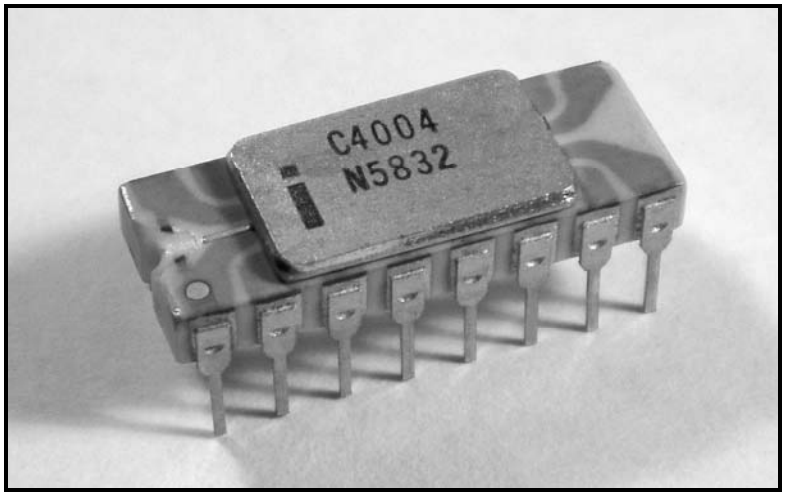



**Figure 6: Intel 4004 Microprocessor**
**(Photo courtesy of Stephen A. Emery Jr., www.ChipScapes.com)**

Because of the requirement to fit the 4004 into a 16-pin package, we got bus-based and bus-centric systems as a result. The 4004 used a bus-based system where many memories and peripheral devices hang off of one bus.

Simple as it is, this sort of interconnect scheme was revolutionary back in 1971. Prior to this date, most electronic systems employed point-to-point interconnect routing and systems with buses were relatively rare. That's because buses require multiplexing circuits, address decoders, and latches—all electronic components that were relatively more expensive than the wires they replaced in non-bused systems.

However, when all you have is a 16-pin package, pins are expensive and multiplexed busing is the only choice. Now even though the MCS-4 system shown in Figure 7 is over four decades old, most of today's microprocessor-based system designs, even on-chip designs, mimic this architecture. Busing is still very much with us even though DIP packages and pin limitations no longer restrict our design choices when developing SOCs.

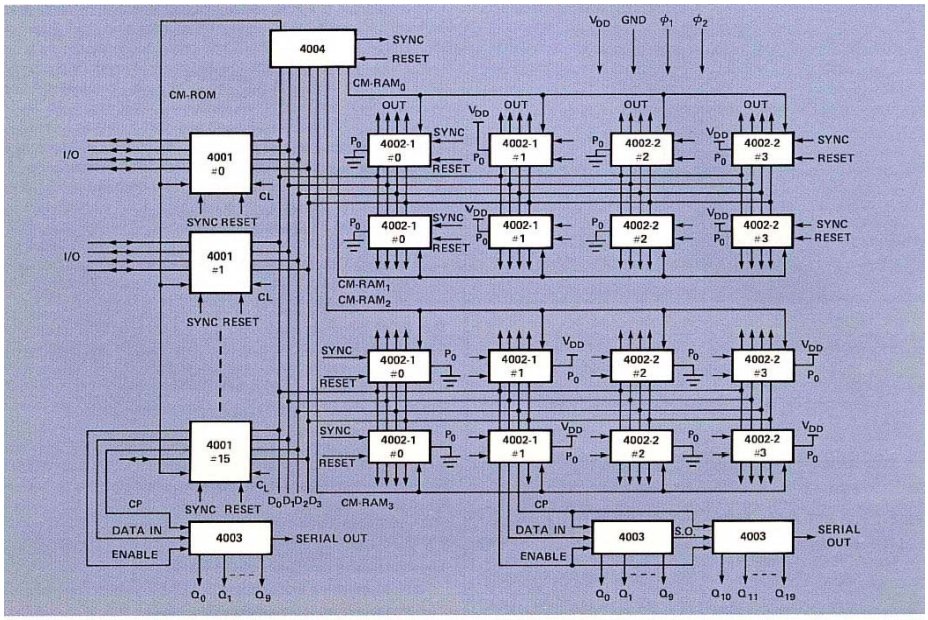



Figure 1. MCS-4 System Interconnection

**Figure 7: Intel 4004-based MCS-4 System**

The Intel MCS-4 system employs its one bus to talk to both memory and peripherals. Microprocessor vendors followed the precedent set by the Intel 4004 and Intel's follow-on 8008 and 8080 microprocessors for many years. A Von Neumann computer system like the Intel 4004 uses the same interconnect and the same memory space to access instruction and data memory. However, this is not the only possible architecture for a processor-based system. There's an alternative architecture that can improve processor performance. It's called the Harvard architecture.

## Computing goes to Harvard

Three decades before Intel developed the 4004 microprocessor, during the 1930s, Howard Aiken conceived of a programmable computing machine while he was a graduate physics student at Harvard University. His idea was to build a machine to solve differential equations using numeric methods. Aiken later shopped his ideas for a programmable computing engine, first to the Monroe Calculating Machine Company and then to IBM.

IBM was interested and it both funded and developed the Automatic Sequence Controlled Calculator, the ASCC, later renamed the Harvard Mark I. The Harvard Mark I was an electromechanical marvel built from 78 adding machines and electromechanical calculators. It consisted of some 765,000 individual switches, relays, rotating shafts, and clutches. Although not electronic, the Harvard Mark I was a programmable calculating machine.

The Mark I's instruction memory was a 24-channel, sequentially-read, punched paper tape, which meant there were no conditional instructions. For data storage, the machine used 60 banks of rotary switches for constants and 72 electromechanical counters for intermediate data values. Each storage counter stored a 23-digit decimal value using 24 electromechanical counting wheels. A full-capacity multiplication took 5.7 seconds and a divide took 15.3 seconds.

One of the legacies the Harvard Mark I left to us is the use of separate data and instruction storage. This non-Von-Neumann computer architecture with separate data and instruction memory (and separate data and instruction address spaces) is now called the Harvard architecture. Its advantage is that there is separate interconnect for the instruction and data memories and that separate interconnect doubles the information bandwidth into and out of the processor. This improved bandwidth was not an especially important feature in 1943 when this computer built from relays, clutches, and other mechanical parts became operational, but it has become very important in the 21st century, when processors run at clock frequencies of hundreds of megahertz or Gigahertz.

Harvard architectures in microprocessors aren't new. Just three years after Intel introduced the groundbreaking 4004 microprocessor, Texas Instruments introduced the 4-bit TMS1000, the first single-chip microcontroller and the first single-chip processor with a Harvard architecture. It had separate address spaces for the instruction and data memories. The TMS1000 combined processor, instruction

memory, and data memory together on one chip, so the package's pin count did not dictate the use of one external bus to talk to instruction and data memory.

A year later, Fairchild Semiconductor introduced the first commercial microprocessor with a Harvard architecture, the Fairchild F8. However, this processor was not a single-chip device. It split the instruction and data-processing functions into two chips and so it made good sense to employ a Harvard architecture because pin count was not the limitation it might have been if the processor was housed in one package.

The Fairchild F8 had a 64-byte register file, which was huge for the time, and systems based on this processor often didn't require any external data bus at all. Similarly, Intel's 8048 microcontroller—introduced in 1976—put the processor, instruction memory, and data memory all on one chip, so again, the package's pin count did not force the use of one external bus for instruction and data memory and the 8048 employs a Harvard architecture.

So does the quirky, bipolar Signetics 8x300, introduced in 1978, which had a 16-bit instruction word but operated on 8-bit data. A Harvard architecture with separate instruction and data spaces was ideal for this machine. It used a 13-bit address space for instructions and a 5-bit address space for data. The 8x300 output the instruction address on a separate address bus and multiplexed the data-space address and the data itself on an 8-bit "Interface Vector" bus.

Finally, TI's first DSP, the TMS 32010—introduced in 1982—employed a Harvard architecture and a hardware 16x16-bit multiplier to significantly boost the processor's DSP abilities far beyond those of general-purpose processors. The TMS 32010 and its successors set a precedent and most DSPs now employ Harvard architectures for performance. So as you can see, microprocessor and microcontroller designers have been using Harvard processor architectures to boost performance since the earliest days of microprocessor design.

### Make the memory faster: Where did caches come from?

Another way of improving processor performance is to improve memory speed. In the early days of microprocessor-system based design, semiconductor memories were as fast as or faster than the processors they serviced. However, with increasing processor clock rates and the rise of DRAM in the late 1970s, processor speeds began to outpace main-memory access times.

This problem had arisen more than a decade earlier with mainframes and magnetic core memory. IBM developed a high-speed memory buffer for the IBM 360 Model 85 and Lyle R. Johnson, then editor of the IBM System Journal, pleaded for a better, catchier name than "high-speed buffer" for the new form of memory. The term "Cache memory" was born. High-speed memory caches are small semiconductor memories that "remember" recent accesses to main memory so that the processor can get faster access to the data or instructions stored in the cached locations during subsequent accesses. In addition, memory caches usually load adjacent memory

locations during the caching operation, using a characteristic called “locality” to improve memory subsystem performance.

The key point to remember here is that cache memory is faster, and usually more expensive, than main memory. Cache memory is generally implemented with SRAM and bulk main memory is usually implemented with DRAM, especially if it’s off chip. The Zilog Z80000 was one of the first if not the first microprocessor to have an on-chip cache memory. It had a 256-byte (not kilobyte or megabyte) unified, on-chip cache.

### Improving the cache concept

Harvard architectures may boost performance by doubling a processor’s bus bandwidth but they complicate a programmer’s life by making it difficult to move information between instruction and data spaces. However, it’s possible to merge the performance-enhancing benefits of caches and Harvard architectures to improve processor performance. In 1985, MIPS did exactly that by introducing the first commercial RISC processor chip, the R2000, which had a unified instruction and data space but boosted performance by using separate off-chip instruction and data caches. However, the MIPS R2000 had only one external address and data bus, to save on pin count, so it had to double the transfer rate on these buses to realize the performance benefits.

The MIPS R3000 was a redesign of the R2000 with enhanced clock speed. The R3000 was the first really successful version of the MIPS processor. It came out at 25 MHz and eventually reached 40 MHz. Because it used external cache memories and a multiplexed memory bus, system designers found it a challenge to design processor boards with R3000 processor chips.

And so we reach the present day with respect to processor evolution. The best performance with the least timing hassle is realized with separate instruction and data caches, each with its own high-speed private bus to the processor. In addition, a third main bus provides the processor with access to main memory. This main bus must support several protocols for interrupts, DMA, multiple bus masters, and so forth. As a result, it can take a few cycles for reads and writes to propagate through a processor’s main bus. Couple that with the slower main memory and you see why fast instruction and data caches can improve performance. A good processor can keep all three of these buses active simultaneously, to maximize performance while minimizing the need for fast main memories and fast global on-chip buses.

> Note: Cache memories usually consist of two memory arrays. A data array holds the instructions or data depending on whether it’s the instruction cache or the data cache. The tag arrays hold indexes or directories that tell the processor’s cache controller just what’s being held in the cache.

### Memory for DPUs

For dataplane processors, the designer often knows a lot more about memory behavior than in the general case for control-plane processors. As a result, it's possible to manually cache instructions in local instruction and data memories for maximum DPU performance by bringing the required instructions into the local memory with firmware-driven block reads or DMA from main memory. While caches speed program execution in general, placing data and code in local memories using policies informed by simulations of code execution can improve performance even more because there's never a "cache miss," so a program need never wait for a cache-line fill to occur.

Caches are quite appropriate for many systems, but the following items should be considered, especially for DPU applications:

- Cache memory's access latency is not fixed or constant and may vary each time an application runs, based on the state of the cache. Some applications—particularly dataplane applications—are sensitive to this variability in access latency.
- Caches do not work well for data sets that do not exhibit much locality or for data that's not organized into large blocks. Caches help accelerate system performance by transferring data in blocks the size of a cache line from the main system memory into the relatively small local cache. Performance suffers if the data or instructions loaded into the cache are not used more than once or if multiple accesses are not made to different words in a cache line.
- Based on physical design constraints such as area, power, or speed, the cache size might be sub-optimal, which results in extensive cache thrashing where cache lines are constantly being evicted. The result will be inefficient operation and the cache is more hindrance than help in such cases.

Many dataplane applications don't require cache memory at all. Local instruction and data memories are sufficient and are less costly in terms of silicon area than cache memories because they do not require tag arrays. In addition, local memories have a fixed access time, unlike cache, which have one access time when instructions or data are in the cache and another when there's a cache miss.

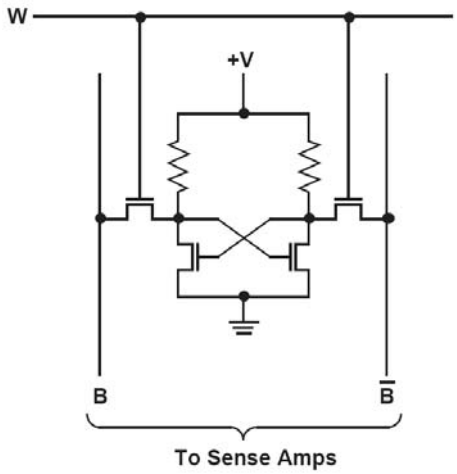
## Types of On-Chip Memory

With this architectural discussion as background, let's turn our attention to the types of on-chip memory you can use to build an SOC. There are three broad categories of on-chip memory:

1. The first type is static RAM or SRAM. Essentially this is just a big array of latches. SRAM is very common in SOC design because it's fast and because it's built from the same transistors used to build all of the logic on the SOC, so no process changes are required.
2. However most SRAM bit cells require at least four transistors and some require as many as ten, so on-chip dynamic RAM or DRAM is becoming increasingly popular. DRAM stores bits as capacitive charge, so each DRAM bit cell requires only one transistor and a capacitor. DRAM's advantage is density, or put in another way, DRAM's advantage is lower cost per storage bit. However, DRAM tends to be slower than SRAM; it has some peculiar requirements that affect system design such as destructive reads, the need for periodic refresh, and special timing requirements. In addition, the capacitors in the DRAM bit cells require specialized IC processing, which increases die cost.
3. Finally, every SOC needs memory that remembers code and data even when the power is off. The cheapest and the least flexible non-volatile memory is ROM. However, you can't change ROM contents after the SOC is fabricated, so we have many, many alternatives for non-volatile memory including EPROM and Flash memory and then a host of new specialty memories, discussed later.

## The SRAM

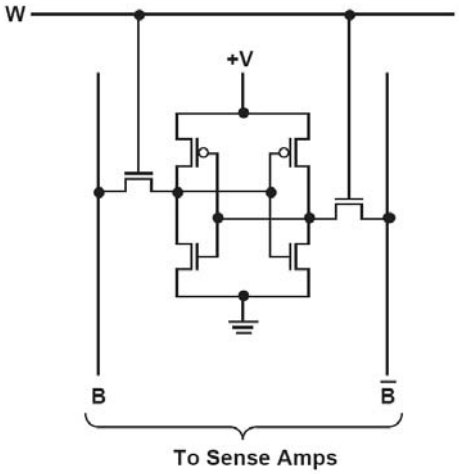
Figure 8 shows a four-transistor SRAM cell. Two N-channel transistors are cross coupled with a couple of pull-up resistors to form a flip flop and two more transistors are write-enable transistors that allow the memory-array control circuits to set the state of the flip flop. Note that current is always flowing through one of the two resistors, so memory designers try to make these resistors as large as possible while still meeting noise-sensitivity goals. Even so, 4T SRAMs tend to have higher standby supply currents because there's always current flowing in one leg of each flip-flop in the memory array.




Source: ICE, "Memory 1997"

**Figure 8: 4-Transistor (4T) SRAM Cell**

Figure 9 shows the six-transistor version of the SRAM cell. The two pull-up resistors are replaced with two P-channel transistors to create a full CMOS latch. In general, 6T SRAM cells are faster than 4T SRAM cells, but they require slightly more silicon area. Note that there are also 8- and 10-transistor SRAM cells for systems that need soft-error resistance or immunity.




Source: ICE , "Memory 1997"

**Figure 9: 6-Transistor (6T) SRAM Cell**

Whether it's a 4T or 6T SRAM cell, the SRAM cell is really a couple of cross-coupled inverters. The two pass transistors connected to the word and bit lines must provide enough current to overpower the outputs of the inverters during a write, so they're generally larger than the transistors used to build the latch.

### One-Transistor SRAM

It would be nice to have the performance of SRAM but without the need for four or six transistors per bit cell. The result would be a fast, low-cost RAM. That's exactly the thinking behind the MoSys 1T SRAM. At it's heart, the Mosys 1T SRAM is really a dynamic RAM. Information is stored on a capacitor, not in a latch.

However, MoSys surrounds the array of DRAM cells with logic that masks the dynamic-RAM nature of the DRAM cell array including an SRAM cache, automatic refresh circuits, error-correcting logic, and a timing pipeline that gives the array an SRAM-like interface. The result is a memory array that mimics most of the characteristics of a standard SRAM memory while reportedly consuming half of the power and half of the area of a conventional on-chip SRAM.

The characteristics of one-transistor SRAM make it attractive enough to draw at least three offerings to the market. In addition to the MoSys 1T SRAM, Novelics has developed the coolRAM 1T—also offered through Synopsys' DesignWare program—and Innovative Silicon offers it's version of 1-transistor SRAM called Z-RAM.

The bit cells in Innovative Silicon's Z-RAM have no capacitors at all. Instead, the technology exploits trapped charge liberated by impact ionization and held by the floating-body effect within the channel of the bit cell's SOI (or silicon-on-insulator) MOS transistor.

The trapped charge alters the transistor's threshold voltage and this charge effect can be used to differentiate a zero from a one. Technical details of these one-transistor SRAMs are available under NDA, so we'll not be discussing them in any more detail here.

### On-Chip or Embedded DRAM

Instead, let's turn our gaze on dynamic RAM IP. We've already been discussing dynamic RAM, sort of, by discussing one-transistor SRAMs, which are DRAMs in disguise. However, if you want to cram the maximum number of bits in the minimum amount of area, you need on-chip or embedded DRAM.

We've already covered the DRAM's storage mechanism. It consists of a capacitor to store the bit and a transistor to read and write the bit.

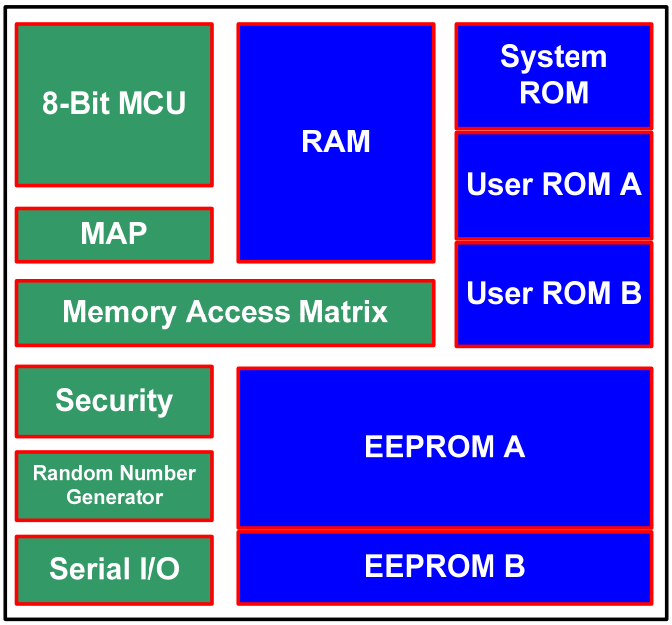
In general, adding on-chip DRAM is going to complicate IC fabrication and will increase die cost. For example, some embedded DRAM or eDRAM cells use trench capacitors where the capacitor is built into a deep trench in the silicon substrate. The trench is formed by anisotropic etching. Deep etching is not a normal part of the logic CMOS process, so it increases fabrication costs relative to pure CMOS logic wafers.

NEC Electronics (now Renesas Electronics) uses a stacked capacitor for its eDRAM. To keep the capacitor size reasonable, as shown on the left, NEC uses a special zirconium dielectric. Again, this substance is not part of the normal processing flow for CMOS logic wafers and the special steps needed to add the dielectric increase the manufacturing cost of the die. These increased manufacturing costs somewhat offset the cost reductions realized from the relatively smaller size of the DRAM cell compared to an SRAM cell.

Most ASIC vendors now offer embedded DRAM that looks more like one-transistor SRAM in terms of access time, hidden RAS/CAS and other DRAM-like timing, and absence of an external RAS precharge requirement. So from a system-design perspective, embedded DRAMs look more like static RAMs than in earlier days. However, you still need to refresh embedded DRAM. With the extra logic and SRAM caches needed to make embedded DRAMs look like SRAMs, these enhanced embedded DRAM arrays are larger than pure embedded DRAM arrays offered in earlier generations. Even so, embedded DRAM is well worth considering if your SOC requires large amounts of memory, as in tens or hundreds of megabits, concentrated in one or just a few memory arrays.

### On-Chip, Non-Volatile Memory

SRAMs and DRAMs are volatile. They lose their contents when system power shuts off. All embedded systems need some form of non-volatile memory, if only to hold boot code to load operating software from a disk or over a communications link. Most embedded systems need considerably more non-volatile memory to hold firmware images, constants, serial numbers, operating configuration and preferences, and other sorts of data. SOC designers have many choices when it comes to non-volatile memory including ROM, EEPROM, Flash EEPROM, and what might be called “specialty” non-volatile memory. We’d need many pages to write about these myriad varieties, and this white paper is long enough, but let’s take a quick look at their salient features.




Source: SGS-Thomson/ICE, "*Memory 1997*"

**Figure 10: Simple SOC**

Figure 10 shows a relatively simple SOC. It's a Smartcard chip, designed for very high volumes. On the chip, you'll find just one 8-bit processor. You'll also find no less than six memory arrays. The percentage of the chip used for memory at about 60% of the die, based on this simple diagram. Even here, with this simple system, memory is important. Note that there is one RAM array, there are three ROM arrays, and there are two EEPROM arrays. Non-volatile memory, which occupies nearly half of this die, is clearly a very important part of this system design and this design is more than 10 years old!

ROM is the simplest of all the non-volatile memories. One transistor for readout, a ground, and you're done. The ROM array transistors are completely compatible with CMOS logic processes. On-chip ROM arrays are small and fast. Their big disadvantage: once you fabricate the chip, you can't change the contents of the ROM. So information stored in ROM has to be very, very stable. And these days, what is that stable?

That's why there's EEPROM—electrically erasable PROM. These days, most EEPROM is flash EEPROM. The difference between EEPROM and flash EEPROM is that EEPROM memory cells are individually erasable while flash EEPROM arrays are designed for block erasure, which simplifies the circuit design and reduces the size of the memory array.

Both types of EEPROM work by storing charge within the gate of the FET that is used for the EEPROM memory cell. There are different ways of trapping the charge within the FET and there are different ways of forcing the charge into the gate, but in any case, the operative mechanism is storing charge in the gate, which alters the state of the FET.

In general, the transistors used to create EEPROM memory cells are not at all like the transistors used to build logic, so adding EEPROM to an SOC means altering the fabrication flow, which increases the cost. Nevertheless, on-chip EEPROM is so useful that many SOC designers are willing to accept the extra cost, thus avoiding an external EEPROM chip. However, a large number of SOC designers prefer to use an external EEPROM. There's no right answer, just different solutions in the design space.

You would think that ROM and EEPROM would just about cover the needs for non-volatile memory, but they don't. ROM is not alterable after chip fabrication and EEPROM cells require special IC fabrication techniques, which increase wafer and die costs.

A few vendors have targeted the available niche for non-volatile memory that can be built with conventional CMOS logic processing. Kilopass and SiDense are two such vendors. Kilopass offers memory IP called XPM and SiDense offers Logic NVM IP. Both technologies offer the SOC designer a way to incorporate OTP or one-time programmable (not erasable) non-volatile memory using very small transistors to create truly compact memory arrays.

Both Kilopass and SiDense program their devices by damaging the gate oxide insulation beneath the memory cell transistor's gate, thus creating an extra leaky transistor that's notably different electrically from a non-programmed transistor. The programming process is irreversible, hence one-time programmable. The resulting programmable device cannot be reverse engineered optically, so this sort of OTP memory is very good for holding secure code and data.

One-time programmable memory can be used for serial numbers; digital trimming, parametric, and preferences storage; device personalization, and to store last-minute code after IC fabrication.

Another NV memory vendor, NScore, offers a one-time programmable memory IP technology called PermSRAM based on hot-carrier electron injection. The storage mechanism here involves the high energy that electrons pick up as they drift in the transistor's gate from source to drain. Some electrons get enough energy to jump into the gate insulator or the insulating side-wall spacer. These electrons become trapped and they permanently increase the transistor's threshold voltage and drain current. Appropriate sense circuits can differentiate altered transistors from unaltered transistors, producing a memory device.

### Off-Chip Memory

Now that we've briefly touched on the several basic forms of on-chip memory, let's look at off-chip memory. When system designers need a large amount of RAM storage—billions of bits—then off-chip DDR (double-data-rate) SDRAM is likely to be the choice. That's because personal computers use DDR memory chips in very large volumes, which keeps the pricing for everyone relatively low. Even if an embedded design only requires a fraction of the capacity of a DDR memory chip or module, it may still be more economical to pay for the excess capacity because the system price will still be lower, thanks to the very large sales volume for DDR memory. Adding a DDR memory port to an SOC design, however, creates the need for an on-chip DDR memory controller.

Similarly, system-design considerations may make it more desirable to have non-volatile memory reside off chip. Again, this choice may result when a lot of non-volatile memory is needed or when the special manufacturing costs needed to add EEPROM to the SOC are prohibitively expensive. In such cases, the SOC design team will add a Flash memory controller.

### DDR SRAM and DDR Controllers

Communication with the DDR SRAM is like a communications protocol, with the transfer of command words that drive logic on the SDRAM chip itself. Although the signal names on the command word look like the old strobes from the non-synchronous DRAM days, the SDRAM interprets these signals as a group that forms a command word. The SDRAM itself handles the actual memory cell timing and the memory controller on the SOC is now responsible for optimizing the commands sent to the SDRAM.

Each SDRAM access requires a specific sequence of commands and there are delays that must be observed between each command in the sequence. The memory controller can hide many of those delays by overlapping multiple RAM-access requests. The quality of an SDRAM memory controller is therefore somewhat dependent on the ability to efficiently manage the SDRAM's operation to minimize latency and maximize bandwidth.

You should note that there isn't just one DDR protocol. DDR2 and DDR3 protocols differ, for example, so you need to be sure to pick an SDRAM controller that matches the SDRAM type or types that you intend to use in your system. There also are some programmable controllers that handle both multiple DDR memory protocols, but you should not assume that all DDR memory controllers handle all types.

A DDR memory controller's ability to queue, schedule, and optimize DDR memory access requests is especially important when the SOC contains multiple DPU cores issuing multiple simultaneous requests, as illustrated in Figure 11. The controller's scheduling ability alone may make the difference between 10% memory utilization and a utilization factor of 50% or more.

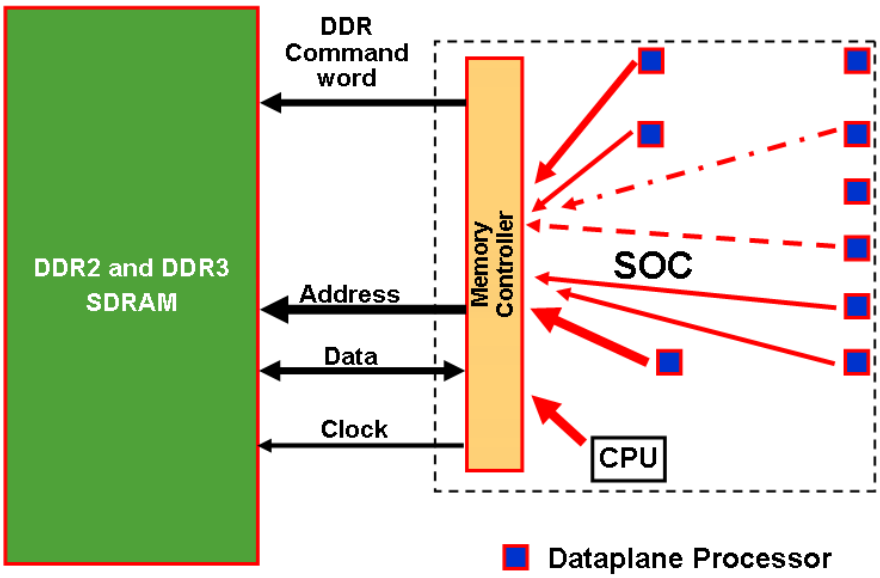



**Figure 11: Multiple Requesters for SDRAM Access**

Note that this illustration shows that some of the on-chip DPUs and the CPU place heavy access loads on the DDR memory (as shown by the heavy lines), some place lighter loads on the DDR memory, some place intermittent loads on the memory (shown by dashed and dotted lines), and some will operate entirely from local, on-chip memory and not access the external DDR memory at all.

With all of these requests issued from several processors, the performance of the DDR memory controller can have a huge effect on system efficiency and can also determine how fast the DDR memory needs to run. An efficient controller allows the system designer to specify slower—and therefore cheaper—DDR memory than an inefficient controller.

Here are some specific criteria, from one of Tensilica's partners, that might help you pick a DDR controller IP block:

- High data throughput efficiency
- Low latency
  - Low-latency path to issue high priority read commands
  - Programmable starvation avoidance to ensure bandwidth for low priority reads
- Modular architecture
  - Solution configured to meet customer requirements
  - Architecture that supports several protocols and multiprotocol combinations
- Quality IP with robust verification

Of course the decision will depend on your application, but in general you want high throughput efficiency. If the controller isn't efficient, you will not get the throughput you're paying for from those fast SDRAMs.

In addition, you'll probably want low latency for at least some of the processors accessing the SDRAM. If so, you'll need a DDR controller with a special low-latency path for high-priority SDRAM access commands. You may also want a controller that has a mechanism that prevents access starvation for low-priority commands. It's also likely you'll want a modular controller architecture to tailor the DDR controller to your specific design. And finally, you'll need to ask about how the controller IP was verified. One of the main reasons for buying a complex IP block, whether it's a processor core or a memory controller, is because the vendor has done the verification work for you. You'll want a good gut feeling that the verification of the block was at least as thorough as the verification you perform on your own custom IP blocks.

Note that the DDR control, address, and data signals operate off-chip at hundreds of Megahertz, so there's a physical component—called a PHY—in addition to the DDR controller's logic. The DDR PHY handles the analog aspects of interfacing to high-speed memory and the PHY is therefore process-specific while the controller logic is generally not process-specific.

You will need both the controller logic and the PHY if you want to add off-chip DDR memory to your SOC. If you're planning on varying the amount of DDR memory attached to your SOC, you'll need a flexible PHY to accommodate the varying loads and timings of different DDR configurations.

For reference, here are some companies that offer DDR memory controller IP blocks.:

- Denali
  - Databahn controller
- Synopsys DesignWare
  - DDRn Memory Interface
- ARM PrimeCell
  - Dynamic Memory Controller (DMC)

There are other vendors as well but there are no standards for DDR memory controllers, so there's a lot of variation from one offering to the next.

The choice between, say, DDR2 and DDR3 SDRAM is not obvious. Older memories offer lower cost per bit because they been in production for longer and the fab lines are amortized.

Newer memories are not necessarily faster. DDR2 memory, for example, can run as fast as DDR3 memory in some modes, but DDR3 uses less energy at the same throughput. That's because DDR3 SDRAM runs the chip's internal RAM array half as fast as DDR2 while operating 8 memory columns simultaneously as opposed to a DDR2 memory's 4 columns. Both types of DDR memory can have the same access throughput but operating the RAM array at lower clock rates gives DDR3 memory the edge in energy consumption.

One especially important item to note: Because DDR3 memory has an 8n prefetch, it will provide 8 words of information for each access. That means that embedded processors with 32-byte cache lines will have trouble with DDR3 SDRAM when connected with 64-byte interfaces — which are common — because one cache-line fill operation will cause 64 bytes to be read and delivered from the DDR3 memory. That means half of the retrieved information must be discarded, because the 64-byte SDRAM transaction exceeds the size of the processor cache's actual request. The excessively large SDRAM transaction degrades the effective throughput of the DDR3 SDRAM. However, this problem does not exist for processor cores that have, or can be configured with, 64-byte cache lines, like Tensilica's Xtensa processor family.

### Non-volatile, Off-Chip Memory and Flash Controllers

If your system requires a lot of non-volatile storage, you may well be planning to use external Flash EEPROM. Your SOC will connect to this external Flash memory through a dedicated interface. The type of interface depends on the type of Flash memory you select. There are two types of Flash memory: NAND and NOR Flash. Each type has different characteristics and the two types are complementary so it's possible you'd have both types in a system. The NAND Flash memory is the less expensive type of Flash memory because its cell size is smaller than the cell size of the NOR Flash. The NAND type of Flash groups memory transistors together and reads and programs the group simultaneously for greater layout efficiency.

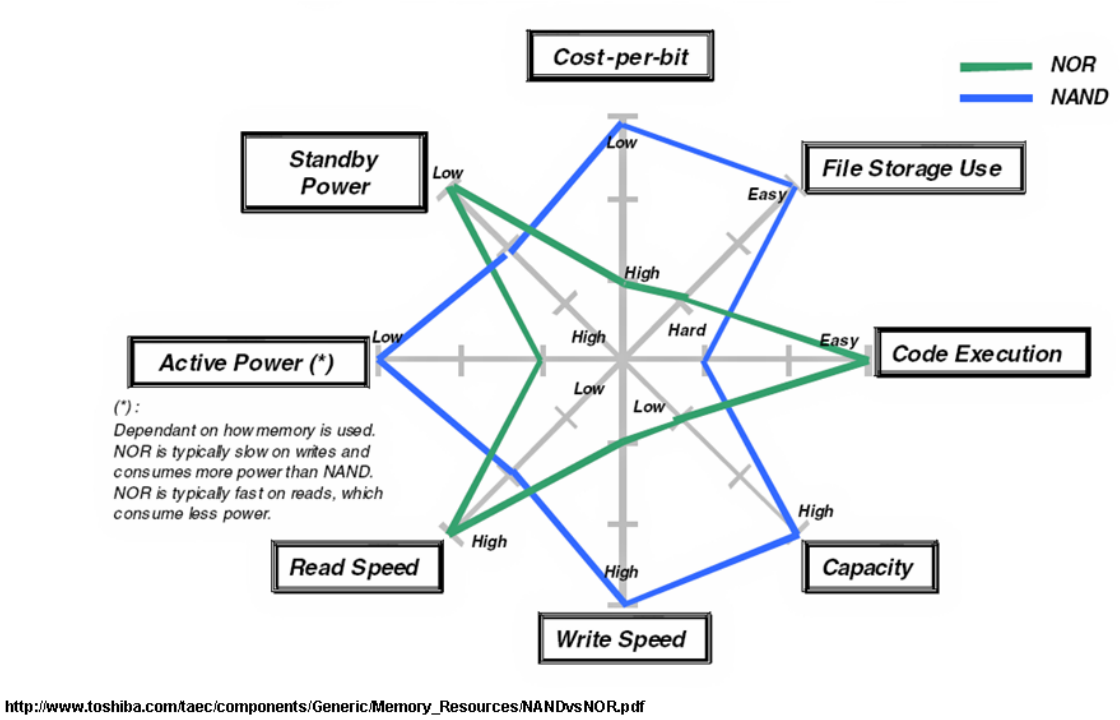
As a result, NAND Flash reads and writes are block-oriented and have a command-driven interface, like the hard disk drives that this type of memory is designed to replace. NOR Flash is truly random-access semiconductor memory and has an address/data interface that resembles RAM or ROM interfaces. Note that some types of NAND Flash are designed to allow booting directly from the Flash memory, but that's not universal.

Both types of Flash memory have limited write endurance, although "limited" actually means hundreds of thousands or millions of write cycles. To help reduce the problem of write endurance, you'll want to consider how your design will perform "wear leveling" to spread the write cycles around the Flash memory.

Figure 12 reproduces a diagram from Toshiba comparing NAND and NOR Flash memory characteristics. Note how complementary the two types of Flash are. NOR Flash offers fast, truly random-access reads and is therefore good for direct execution of code while the smaller NAND Flash cell size permits low cost per bit and high storage capacity.

You'll need different memory controllers to interface to external NAND and NOR Flash memory. For NOR Flash, a more conventional memory controller suitable for static RAM and ROM will suffice. This is because of the random-access nature of NOR Flash. However, you'll need firmware to deal with the peculiar requirements of NOR Flash — such as the relatively long write cycle relative to SRAM, error correction, the need to handle bad memory locations, and the need for wear leveling.

For NAND flash, you'll either need a Flash controller IP block, specifically designed to manage the block-oriented NAND Flash memory or you'll need an appropriate firmware stack to manage the memory with a processor.




http://www.toshiba.com/taec/components/Generic/Memory_Resources/NANDvsNOR.pdf

**Figure 12: Spider Diagram Comparison of NAND and NOR Flash**

### Special Applications for On-Chip Memory

Let's finish by discussing some special architectural uses for on-chip memory. There are three uses of particular note for SOC designers. The first is the use of dual- and multi-ported on-chip SRAM for processor-to-processor and block-to-block communications. The second use—for FIFO queues—can also be used for processor-to-processor and block-to-block communications. Finally, you can use high-speed lookup ROM and RAM to boost processing performance. All of these uses can be accomplished with conventional processor cores but special processor features can extract even more performance from these specialized memories.

Recall the basic bus-based microprocessor system architecture appearing in Figure 7. Things haven't changed much for single-processor systems, even on chip. Now let's consider the problem of introducing another processor into this system, making a multicore design. The easiest way to add another processor is to simply attach it to the existing bus as shown in Figure 13.

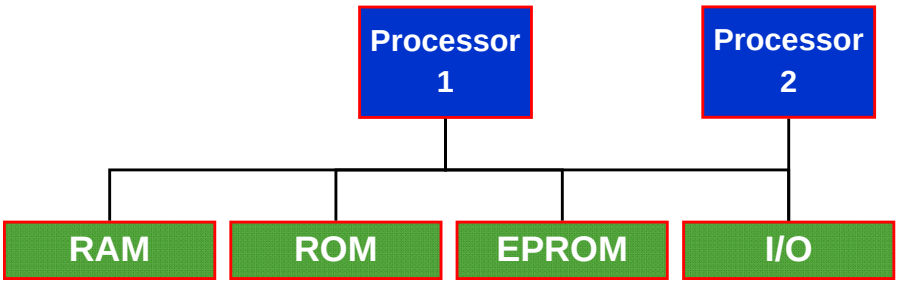



**Figure 13: Two Processors on One Bus**

The two processors communicate with each other using the shared RAM on the common bus. For two slow processors with little data to share, this architecture works fine. However, as processors get faster and as more data passes between processors, this architecture starts to saturate the bus. Each piece of data passed between processors requires one write and one read. That's two bus transactions plus bus arbitration required to move one piece of data. That's an inefficient way to pass data between processors.

One thing we can do to alleviate this overhead is add local RAM to each processor, as shown in Figure 14.

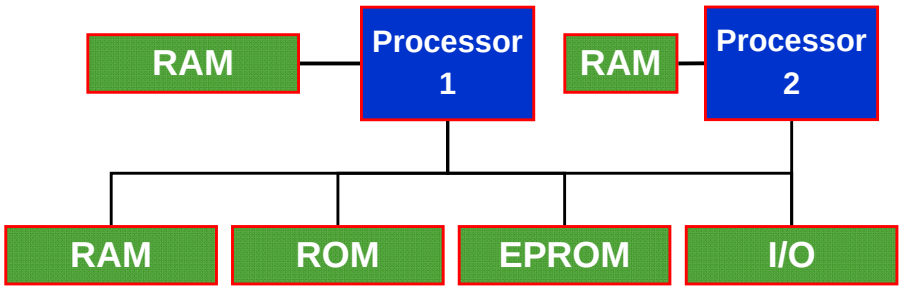



**Figure 14: Two Processors on One Bus with Local RAMs**

When Processor One wants to share data with Processor Two, it gets control of the shared bus and then writes directly into Processor Two's local RAM. This approach cuts the shared-bus traffic in half and boosts available shared-bus bandwidth. All of Tensilica's Xtensa processors support this sort of feature.

We can also remove all inter-processor traffic from the main bus by using a true dual-ported RAM, as shown in Figure 15. Here we see that Processors One and Two can pass data to each other using a dual-ported RAM that is not connected to the shared bus.

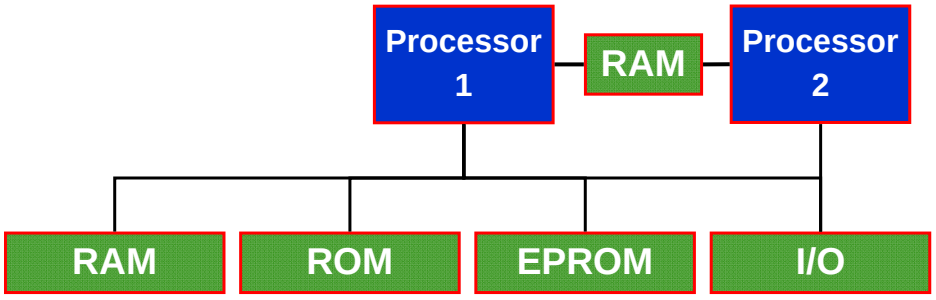



**Figure 15: Two Processors on One Bus with Shared Dual-Port RAM**

The dual-ported RAM acts as a mailbox between the processors. Beyond reducing or eliminating shared-bus traffic, this approach allows the shared RAM to be as wide or as narrow as needed to suit the needs of the application. The communications path through the dual-ported RAM is not restricted to the width of the shared bus. It can be wider or narrower depending on the needs of the application.

Each processor may also need its own local memory in addition to the shared memory, so it's very convenient if the processor core supports multiple local memories (such as found in Tensilica's Xtensa processors).

You can also use a FIFO memory to link two processors as shown in Figure 16. A FIFO is a unidirectional memory with no explicit address port. Processor 1 can send information to Processor 2 at any time by writing to the FIFO. Processor 2 reads from the FIFO as needed to retrieve the information. FIFO-based communication is very common on SOCs these days. Note that the FIFO traffic is separate from traffic on the shared processor bus, as was the case for dual-ported memories.

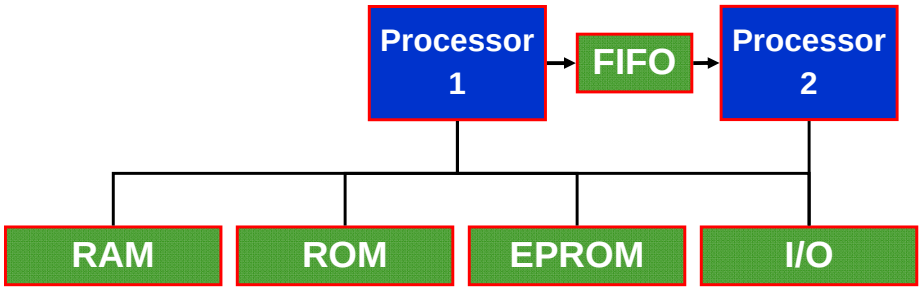



**Figure 16: Two Processors on One Bus with FIFO Communications**

It's possible to equip Processor 1 with multiple FIFO ports that lead to more than one processor as shown in Figure 17. It takes surprisingly little hardware to do this and the performance benefits are huge. This drawing shows a second FIFO port to Processor 3, which is not on the shared bus.

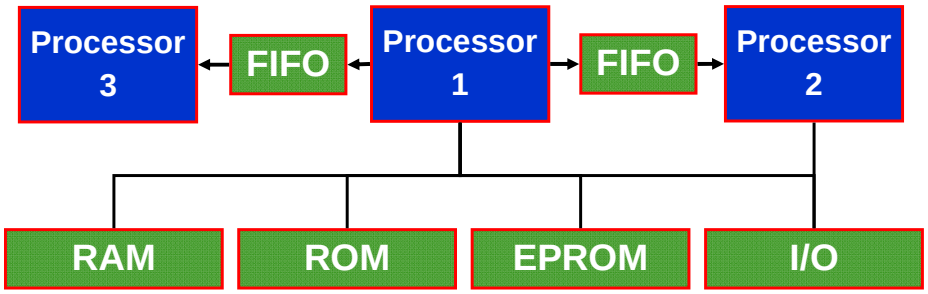



**Figure 17: Three Processors on One Bus with FIFO Communications**

And of course, it's possible to have bidirectional FIFO communications between two processors using two FIFOs as shown in Figure 18. Note that Tensilica's Xtensa processors can be configured with several FIFO Queues for systems where the inter-processor connection requirements are high. In fact, Tensilica processors can support hundreds of FIFO Queues, but it's unusual to use more than 8 or 16 FIFO Queues per processor in a system design.

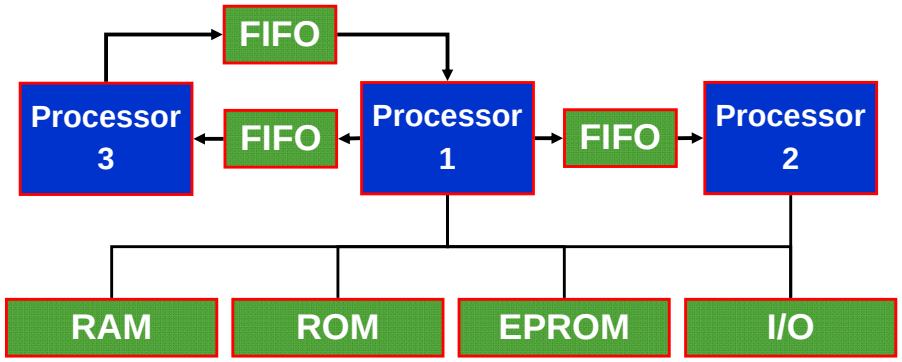



**Figure 18: Three Processors on One Bus with Bidirectional FIFO Communications**

There's one additional useful way to use on-chip memory. We can configure a local memory as a lookup table. Now of course, any processor can use any sort of memory, local memory or memory located somewhere on the bus, as lookup memory. An index into that memory retrieves the desired data. Processors do this all the time. Lookup tables are very, very common. But lookup tables located on memory attached to a bus take several cycles to access, due to the bus protocols. Any read of bus-based memory takes several cycles.

However, we can configure local memory as a direct, flow-through lookup table and Tensilica's Xtensa processors support this type of memory use very well. You can configure a lookup port and a lookup instruction that will output an index, essentially a memory address, and will accept data from the indexed location in memory on the next cycle. This design approach makes memory-based lookup tables, whether in ROM or RAM, very fast and can materially speed up many algorithms.

## Conclusion

This white paper has presented a lightning tour of SOC architectures and memories for SOC design. Memories of all types are crucial to the design of SOCs and they can be used in many ways, from the conventional bus-based and local memories seen in all SOC designs to multiported memories, FIFOs, and lookup tables. We've briefly looked at various static and dynamic RAMS, non-volatile memories, and memory controllers. SOC memory is a very broad topic and this white paper can only overview the topic. As always, check our Web site at www.tensilica.com for more technical information about using processors and memories in SOCs.