Chapter 2
Cache Architecture and Main Blocks

Embedded memory architecture is important as it is the first step in designing the memory subsystem and in deciding how the cache fits in the big picture. Since TCM is a simpler version of cache, in this book we will concentrate on cache design. Cache architecture is normally led by the micro architecture team with strong input from circuit design and process technology. Circuit design input provides area, access time, and power for a given cache size. Process technology team provides leakage estimation per memory cell type, expected yield, and soft error rate per cell type. The key decisions at the end of this effort is a spec outlining the cache hierarchy and size for each cache level, associativity, replacement policy, cache line, and cache blocks access (serial versus parallel) [6, 7, 26]. The process of reaching a decision is much like a negotiation process between the three main disciplines and as each one is an expert in his own domain the knowledge of the other domain is valuable in reaching optimum solution. For example, if the architecture experts understand some of the limitations on the circuit side like minimum voltage requirements, cell size versus performance versus leakage then he/she can propose innovative solutions on the architectural level to deal with retention, voltage island, etc. In the same way, circuit design knowledge of architecture and how the address and data are generated and consumed will help optimize the overall timing path.

2.1 Cache Main Blocks and Data Flow

The cache subsystem is shown in Fig. 2.1 and it consists of the data array which has the main data storage, tag array which determines the way-hit in the cache based on comparing the physical page number (ppn) with the tag value determined by the index bits. Additionally, each tag entry has state bits which hold important information regarding the validity of the tag. Depending on the cache architecture, the state bits can range from 1 to 3 or 4 bits for each cache line [15]. The number of state bits depends on the cache replacement algorithm and the cache write policy.

B. Mohammad, *Embedded Memory Design for Multi-Core and Systems on Chip*, Analog Circuits and Signal Processing 116, DOI 10.1007/978-1-4614-8881-1_2,

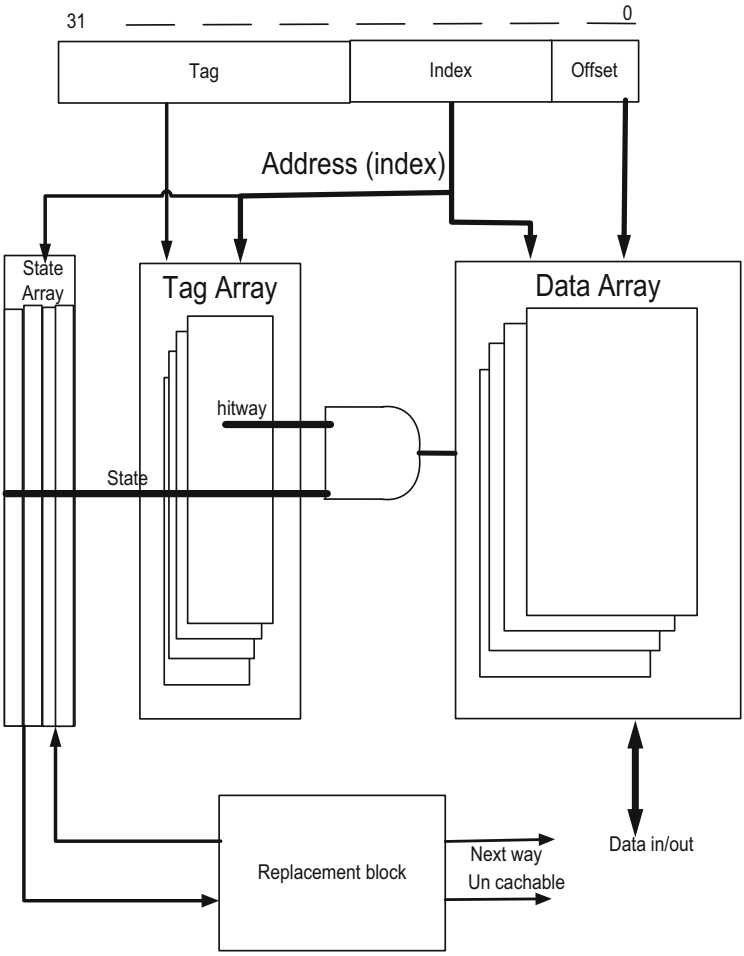

Fig. 2.1 Cache system main blocks and interface

Typical state bits are valid, reserved, and dirty. Both the hit way and the state array along with the index bits determined the entry in the data array.

The term cache hit is used to describe the case when the processor fetches data from the cache and locates it in one of its entry whereas the term cache miss is referred to the case when the processor looks up certain address in the cache but does not find it. Cache misses could be due to the data not being loaded from main memory (primary miss) or being invalidated due to other processor updating the data (secondary miss). The term hit rate is used to describe the percent of CPU cycles which are cache hit while the miss rate describes the remaining percentage of CPU cycles [6].

In addition to the cache blocks mentioned above recent embedded processors have introduced a store buffer which is a simple storage like a queue to hold write data either to caches or to main memory [15]. The main goal is to improve speed of the cache especially when store instruction is the critical path. The reason why store to memory is more critical is that unlike read (load instruction) one cannot speculate and write the data because the old stored data will be invalid, but read access can be speculated and if it is wrong there is no data integrity issue, it only waist energy.

Caches are organized such that every memory location has a specific entry or entries, depending upon the associativity of the cache. Any one of these possible entries is called a "way." Therefore, every way in the cache has its own state bits. Cache associativity is an efficient way to map a large size of main memory into much smaller cache sizes. There are three basic associativity types for caches which will be discussed in details in the next section.

2.2 Cache Associativity

1. Direct map or one way set associative cache: The entire cache here is mapped to one contiguous area in the main memory. In this case only one copy of the tag is needed and it indicates whether there is a hit or a miss on the cache. It requires less area overhead and is relatively fast access due to smaller load on the address bits and overall small tag. The downside is that its hit rate is low due to the 1:1 mapping between the block of the memory and the cache address.
2. Full associative cache: In this case any part of the main memory address can be mapped to any address in the cache. This requires a full associative tag array which is expensive in terms of area and timing. The advantage of this approach is that program has more flexibility than the direct map cache and potentially experiences less cache misses.
3. Set associative cache: It is between the direct map and the fully associative cache where the cache is divided into N-ways each has the same size. This organization requires N-comparators.

Table 2.1 lists the main metrics of cache for the three types of associativity. Most caches in modern microprocessors and SOCs are set associative where 4–8 way set associative is very common while higher associativity is mainly used for server chips and high performance processors.

Table 2.1 Comparison of cache type associativity in terms of hit ratio, speed, and area

Cache type	Hit ratio	Search speed (power)	Area overhead
Direct map	Good	Best	Best (low area overhead 1-comparator)
Fully associative	Best	Moderate	Worst (many comparators)
N-way associative	Very good	Good	Moderate (N-comparator)

2.3 Cache Memory Write Policy

Since cache memory holds important data and it is part of the main memory there has to be consistency between main memory data and the data in the cache. There are cases where the embedded memory has to write the data back to the main memory, and this occurs when the data in the cache has to be evicted to place more fresh data. There are two main ways the cache updates the main memory with the new data: write-through policy and write-back policy. Most processor use write-through for higher level caches like L2 and write-back for L1's.

2.3.1 Write-Through Policy

The processor writes to the main memory at the same time it updates the cache. It is the easiest to implement and does not require complex logic to arrange for bus transaction or coherency. The downside is that the processor has to wait for slow main memory access to finish the memory write. The other downside is that it could result in unnecessary write to the main memory as the write data could be updated shortly by the same processor, in addition to possible bus congestion due to many writes.

2.3.2 Write-Back Policy

Main memory gets written to only when "dirty" block gets updated in the cache. This requires a dirty state bit for each block set when cache block is written to. The advantage is that it eliminates unnecessary writes that is common for write-through. The main disadvantage is that it needs added hardware and protocol to eliminate the possible coherency or consistency issue especially for multi-processor [6, 15].

2.4 Replacement Algorithm

In addition to write policy cache has to select which block to replace when all blocks are used. There are three popular algorithms used to select the replacement block: random, least-recently used (LRU), and first-in-first-out (FIFO).

Random: Replace block gets selected at random. It is the simplest and it requires minimum logic to implement. However, it is the least effective and can result in replacing needed blocks.

LRU: Block that is least used can be replaced by a new block. It requires complex logic and storage area to keep track of each block usage, but it is efficient in selecting the correct block.

FIFO: In this method block gets pushed onto queue when accessed. The replaced block gets evicted using popping queue. It requires moderate logic to keep track of blocks and its efficiency is in between the LRU and Random.

In addition to replacement algorithm for single processor, multi-processor and SOC with shared caches have to deal with coherency which will be discussed in Chap. 3.

2.5 Cache Access Serial Versus Parallel

Another important decision to make for caches is how to access the different blocks in the cache. This decision has big impact on the access time and power. There are three main options for memory access:

1. Parallel access (TAG||State||Data): Access tag, state and data array are all in parallel. This is suitable for high performance processor and the result is potentially a single cycle access. The downside of this option is the fact that it has higher power due to two reasons:
   (a) Data array needs to fetch n-number of data sets where n refers to the number of way associativity. For example, if the cache is 4-way set associative then the data array fetches 4-sets of data in parallel with the tag lookup to select one of the four if state bit is valid. There is always wasted power for the three unselected ways; in addition, wasted area due to routing resources to route the four copies of data to the way select multiplexer.
   (b) When there is a miss on the tag due to tag miss or invalid cache line, the data array access power is still wasted.
2. Serial access (Tag||State → Data): In this design the tag and state arrays get accessed first and if there is a hit and is valid, the data array gets accessed. This is worse than the parallel access in terms of timing as in most cases it requires 2-cycles to access the tag first then data array. However, this is the best option for power as there will be no wasted energy on loading multiple copies of the data arrays from the different ways and if there is a miss then the data array is not accessed. The next section will discuss more about tag array design and its impact on timing and power.

There is also multiple level of parallelism where the data hit signal from the tag can be factored into the column multiplexer of the data array before the sense amplifier so this way the routing resources and power can be saved.

2.6 Cache Architecture Design Example

In this section, we will describe the tradeoffs and the process that is involved in designing a typical embedded memory. The first decision is to pick a memory size for each hierarchy, where it is always preferable to be a power of 2 sizes as it makes

it easy to organize and calculate. However, there is no reason not to have a memory size that is not a power of 2, for example 24 KB is used in some processor. The exact size is determined based on performance evaluation of popular application on the target architecture which sweep the memory size versus performance metrics such as million instruction per second (MIPS). The performance evaluation also assumes certain associativity and cache line so the recommendation for cache size, associativity, and cache line comes from architecture team based on performance evaluation. Popular memory architecture for embedded market and multicore is 32 KB L1, 8-way set associative and 256 KB L2 with 4-way set associative and cache line is 32 bytes for both caches. The L1 cache is normally run at the same frequency as the processor frequency (assume 1 GHz frequency is the target). The next step is to calculate the number of cache lines (sets) per way.

Number of sets per way is equal to (NS)

$$NS = \frac{cache\ size(Byte)}{associativity * cache\ line(Byte)}$$

$$L1\_NS = \frac{32 * 1024}{8 * 32} = 128$$

$$L2\_NS = \frac{256 * 1024}{4 * 32} = 2048$$

This number of sets per way will be the same in the tag array, state array, and PLRU, but the number of bits per set will be different for each array. It is clear from the above simple calculation that L2 has much more entries than L1 and hence it needs wider multiplexer to select the corresponding entry which requires longer access time. There are many options to design the data array which is referred to as banking. Next section will discuss in details banking options for data arrays.

2.6.1 Data Arrays Banking Options

Banking by way: In this option data arrays are divided into equal sections with each section containing all sets from the same way. In our example we will have eight sections for L1 one for each way that contains 128×256 bits of memory or 4 KB. The next level of details is how to organize this 4 KB into banks. The simplest way is to have one bank with 128 entries and 256 bits which means an array of 128 rows and 256 columns. Other options are to have multiple banks per way. For example, a bank of 64 rows by 128 columns gives 1 KB memory and then it needs four banks of the same instance. A general form of choosing the bank size is $BS = 2^n * 2^m$, where n is the number of rows and m is the number of columns.

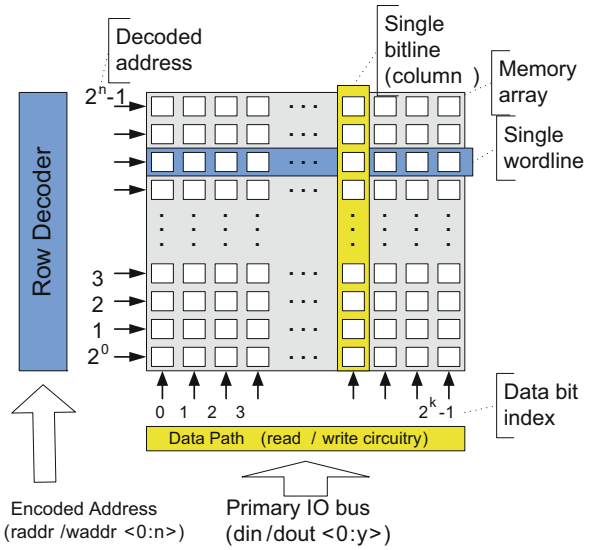

Fig. 2.2 Typical memory bank structure with main blocks

The m and n variables are decided based on the access time and power tradeoffs, for example smaller n will result in faster array access as the number of cells per bitline is small. Figure 2.2 shows the memory array organization and main sub-block:

1. *Memory array*: This is the actual memory storage component of the design containing 2^m columns (bitlines) by 2^n rows (wordlines) of memory cells. The block size is $2^m \times 2^n$ bits.
2. *Row decoder*: This is used to decode an **n** bit wide encoded address into an $\mathbf{2^n}$ bit wide decoded address which in turn is used to select the corresponding wordline to enable for read/write.
3. *Data path*: This contains read and write circuitry for each bitline. Typically this would be a bit line driver for write operations and a sense amp/output driver for read operations. Also a multiplexer for data in and out if there is a bit interleaving in the design.

Figure 2.3 shows two organization for the same 8×4 example, option a will result in slow bitline development due to eight loads and long wire while option b has shorter bitline and less load on it. In addition to better timing, organization b is good for soft error as adjacent cells in the same row do not belong to the same cache line/entry. The IO circuit benefit from a wider array as it has better aspect ratio than organization a. Most of SRAM cell used in majority of embedded memory has aspect ratio of 2 with shorter bitline and long wordline. The challenge with organization b is the need for column mux and the extra power wasted on the unselected bitline during read or write operation as all column share same wordline.

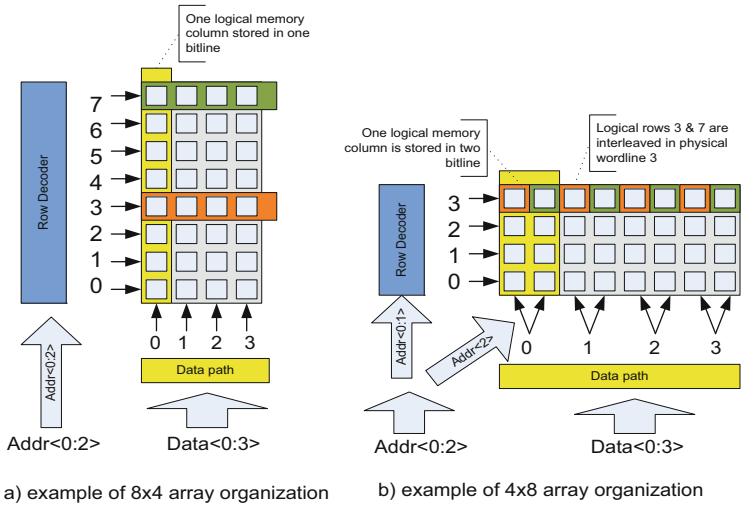

Fig. 2.3 Array organization: (**a**) 8×4 with long bitline wire, (**b**) 4×8 organizations

2.6.2 Tag Array Design for High Associatively Cache

The question of whether CAM tag- or SRAM tag-based cache designs are better is as old as cache design itself. However, with the introduction of the StrongARM™ processor [23], the issue has taken on greater significance in the embedded processor space. Ever since this design demonstrated the superiority of CAM-based designs, it has been a widely held belief that CAM-based caches inherently operate using lower power than SRAM-based ones. Both academic and industrial studies [25, 26] have described several of the reasons for choosing CAM tags over SRAM tags for high associative caches; however, these reasons do not include detailed quantitative arguments.

Moreover, with the introduction of multithreading and multicore SOC, the need for bigger cache size as well as better associative interaction are essential to achieving high memory performance. This combined with usage of the smaller geometry process technology—with wire cap becoming a significant contribution to power consumption—demonstrates the need for a detailed and through study of the two tag options.

We will present a detailed analysis of the same cache architecture implemented in both styles and show that, for usage patterns with moderate to high switching in address and data, CAM tag caches will consume the same or even more power than will SRAM tag caches.

In our comparison, we looked in detail at a recently completed DSP core Level 1 (henceforth L1) data cache and used data from a very similar cache from a

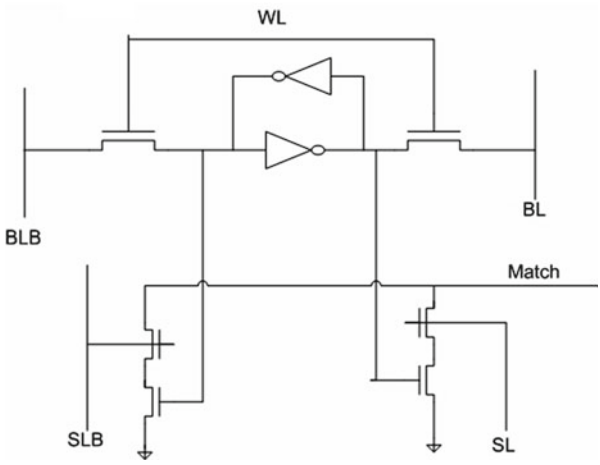

Fig. 2.4 CAM cell schematic example

high-performance ARM core designed for the same process technology; both devices were designed at Qualcomm. In both cases, the L1 data caches were 32 KB, 16 ways-set associative caches, each with a 32-byte cache line size and 64 entries per way.

The design of the DSP and ARM cores chosen for the L1 caches were both physically tagged and virtually indexed using 32-bit virtual addresses (VA) [15]. The 32-byte cache line size and a minimum page size of 4 KB effectively divide the addresses into the tag, index, and offset fields, as shown in Fig. 2.5.

The tag bits of the VA generated by the address generation unit (AGU) must be translated into a physical address (PA) through the translation lookaside buffer (TLB), before the cache access can be completed. The untranslated, index, and offset bits of the address are available much earlier than the PA tag bits. This timing difference is an important factor in the critical speed differences between the cache organizations.

We will discuss the comparisons between the two cache styles in terms of structural, timing, area, and power.

2.6.2.1 Structural Comparison

The main difference between a CAM-based tag and SRAM-based tag is that, in the CAM tag, each entry of the tag has its own comparator. The CAM cell (shown in Fig. 2.4) has both a 6T SRAM cell and a comparator with different topologies based on speed, power, and metric area [50]. The sl and slb are the search lines where the PPN address is compared to the stored value of the tag. The match line combines several CAM cells—typically eight of them—and it features a dynamic signal with pre-charge logic to precondition the node to logic 1. The match line also depends on the value of the tag, and either remains or is discharged to logic 0. In the

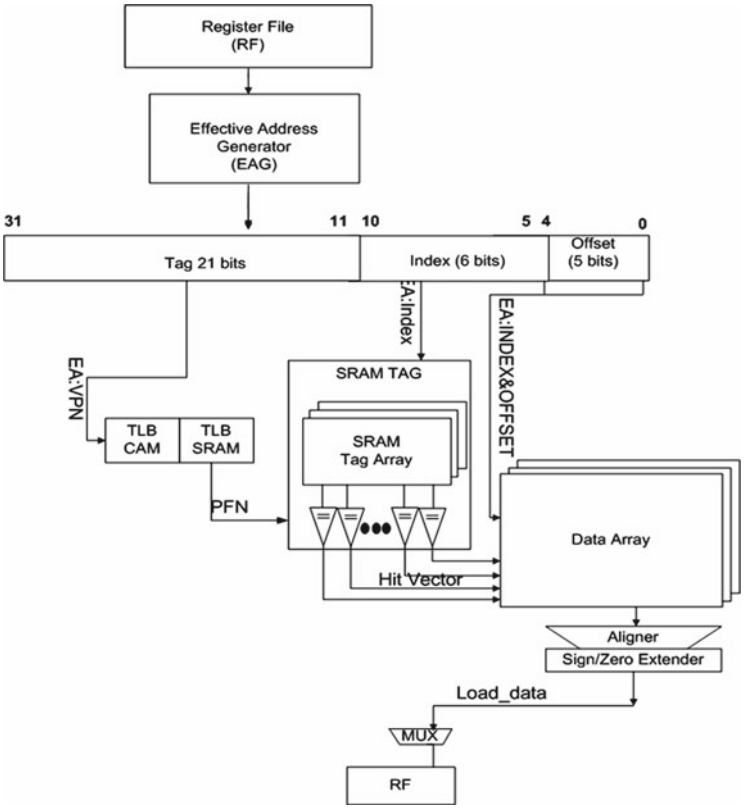

Fig. 2.5 SRAM-based tag cache operation and data flow

CAM-based tag, the higher address bits are distributed among the selected sets of tags to compare to the stored tag. The result is referred to as hit way.

In the SRAM-based tag, the number of comparators is equal to the number of ways. The tag data are stored in a typical small-signal array, which is accessed using the lower address bits to select the appropriate sets that need to be compared to the PA.

Figure 2.5 shows the data flow of the cache array using an SRAM-based tag.

Besides the data and tag arrays, this cache stores the cache-line state in a separate, multi-ported memory array referred to as the state array. The cache operation starts with the delivery of the 32-bit VA from the AGU. The VA must be translated into a PA using the TLB. The VA index bits are used to access the tag, data, and state arrays. The PA tag is compared to the tag values stored in the tag array entries, and after being qualified by the cache entry state, the hit results are used to select the corresponding data array entries. The state bit is used to identify the status of the cache line, that is, whether it is valid, invalid, or reserved. The replacement algorithm keeps track of each cache entry and updates the state array accordingly.

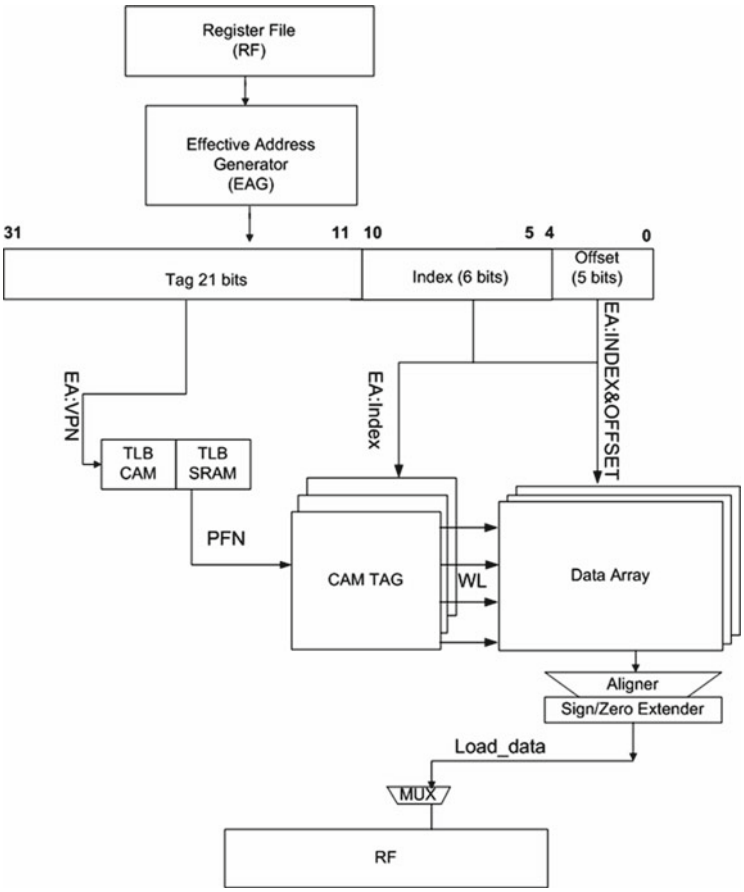

Fig. 2.6 CAM-based tag memory organization and data flow

A key architectural decision is whether the data arrays are accessed in series with the tag arrays or accessed via parallel tag arrays. For power considerations, a serial cache lookup is typically desired in an embedded processor. This implies that the TLB, tag, and state arrays are accessed first and that the data arrays are only accessed after the compared tag results are available. In such a design, the cache data array will not start until the exact set and way have been selected. Figure 2.6 shows the organization of the CAM-based cache. It is similar in many respects to the SRAM-based cache. For a CAM-based tag, the cache banking must be based on the index in order to store all the contents of a cache line, with its respective CAM entry. Additionally, all 16 cache lines for each set must be stored in the same bank to ensure that only a single set of CAM comparators is activated. Overall, these requirements allow for less flexibility in the organization of the CAM-based cache. Moreover, since the L1 in our case is pseudo-dual ported, keeping the entire cache line in one set of a bank is important for minimizing bank conflicts. Other banking

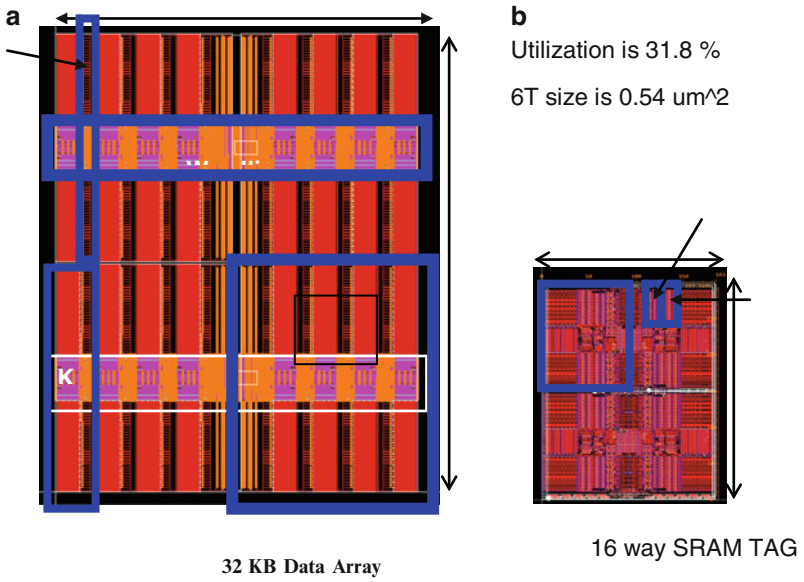

Fig. 2.7 SRAM-based tag 32 KB memory organization. (**a**) data array for sram-based (**b**) sram-based tag array

schemes could work functionally but would require either duplicate CAM entries or the activation of more than a single bank of CAM comparators.

2.6.2.2 Area and Floor Plan Comparison

The choice of CAM tag instead of SRAM tag array directly affects both banking options and the floor plan used. SRAM-based tag array caches are more flexible with regard to banking options, as the wordline selection occurs through the decoding of the index bits while factoring in the hit signals from the tag array. In our design, the wordline decoding occurs in three levels: first, the quad level, which is 8 KB and selected using EA[4:3]; second, sub-array selection is done using EA[6:5]; and third, the set of 16 ways is selected by EA[8:7]. Finally, the hit vector will select one of the 16 ways. Each sub-array has 64 IO (compared to 256 in the CAM-based tag), with 4:1 column muxes in sub-array selected by EA[10:9]. Figure 2.7 illustrates the data array area and hierarchy using an SRAM-based tag (Fig. 2.8).

Figure 2.6 shows how the CAM-based tag data array is organized; bits 8 and 5 of the EA are used to select sub-arrays while bits 10 and 9 are used to select sets of 16 ways.

The cache line for the SRAM tag is distributed to four double words (DW). Each quad contains one double word from all sets. This organization makes the fill and evicts bus routing much simpler than does the CAM-based tag, as each quad drives a DW. The state array is 64 entries, which match the number of entries per way, and has 48 columns, which are 3 bits per way. For the CAM-based tag, the state bits are added to the CAM array. The 6T SRAM cell for 65-nm area is 0.52 μm^2 and the

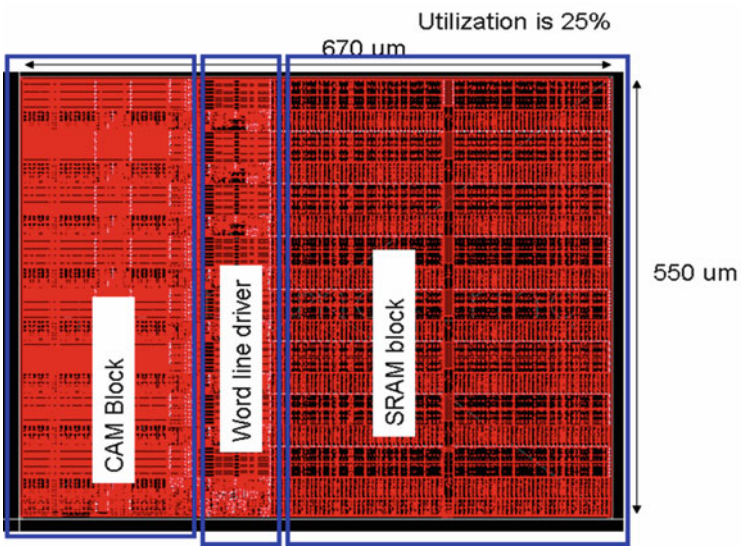

Fig. 2.8 CAM-based tag 16 KB memory organization

Table 2.2 Area of L1 32 KB 16 ways SRAM-based tag

SRAM-based tag area			
	X (μm)	Y (μm)	Area (mm^2)
32 KB data array	510	750	0.3825
SRAM tag	184	310	0.11408
State array	148	80	0.01184
Total area			0.508

Table 2.3 Area of L1 32 KB 16 ways CAM-based tag

CAM-based tag area			
	X (μm)	Y (μm)	Area (mm^2)
32 KB data array	542	700	0.3794
CAM tag	280	700	0.196
State array	150	160	0.024
Total area			0.599

CAM cell area, the conventional dynamic and CAM-based cell are 4 μm^2. Note that the CAM cell has an area eight times the size of that of the SRAM cell.

Tables 2.2 and 2.3 show the area for each tag implementation. The CAM-based design occupies 18 % more area than the SRAM-based tag design. This arises from the difference in area between the two tag designs.

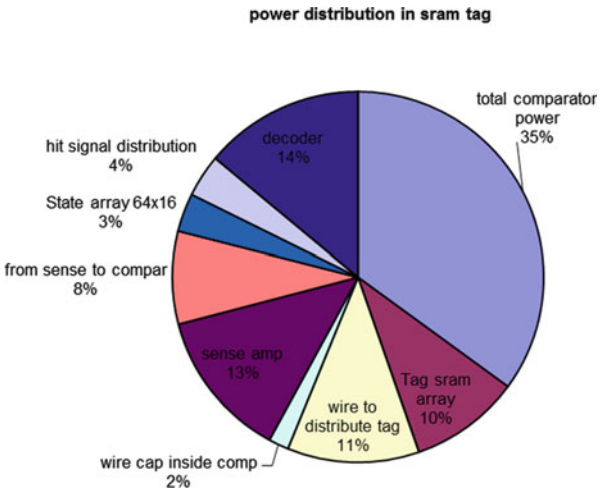

Fig. 2.9 Power distribution in L1 data cache tag (SRAM-based) for SA = 0.5

2.6.2.3 Timing Comparison

As is true for most caches, generating hit signals to determine which way of the 16 total ways needs to be accessed is the most critical path for the two cache designs. The CAM tag is distributed and tightly coupled with the data array sub-bank, making the timing path from the TLB to hit more critical. For SRAM tags, the tag array is compact and localized in a relatively small area; this makes the main speed path from TLB to hit signal less critical for SRAM-based tags. Intel xscale [25] with 32 ways-set associative implements a speculative CAM tag search parallel to the TLB. This results in a special read/write operation on the data array to enable the retaining of old data in case the TLB access misses. Further, it requires the addition of temporary storage of the previous data, which increases the cache size by 2 KB (which amounts to about a 6 % increase in the cache area). The output of the SRAM tag is 16 ways hit vectors that is one hit, and can be optimized in both routing and power. One more complication stemming from the physically distributive nature of the CAM-based tag is the combination of the hit/miss way, which is necessary for a replacement algorithm. Moreover, if the cache is a dual issue cache, such as pseudo-dual ported caches, the timing also becomes a challenge.

2.6.2.4 Power Comparison

Most of the previous work, which, for the most part, related to power comparison between CAM- and SRAM-based tags, overlooked the power associated with interconnect capacitance. Our analysis assumes 65-nm process technologies from a commercial foundry. In our comparison, we assume that the functions common to the two implementations—such as TLB, state array, and data array access, as well as the power associated with driving the load/store bus—are all equal. As is clear

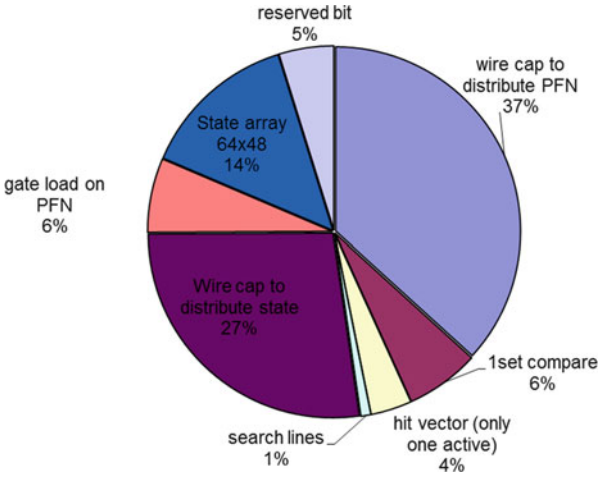

Fig. 2.10 Power distribution in L1 data cache tag (CAM-based tag) for SA = 0.5

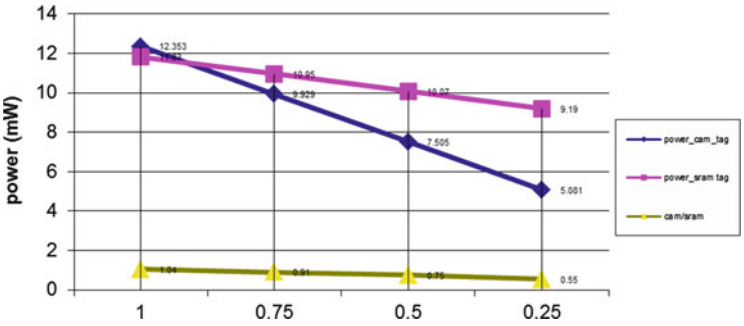

Fig. 2.11 Switching capacitance (energy-delay^2) of CAM tag and SRAM tag

from our earlier discussion, fill and evict operations consume more power in CAM-based tags, but these operations only seldom occur, so their effect on the total power consumption is small.

We now turn our focus to analyzing the power associated with the tag array and hit generation, which is the principal difference between the two designs. Figures 2.9 and 2.10 show the power distribution in the CAM-based tag and SRAM-based tag, using a switching factor (SF) of 0.5 for both cases. The switching factor is the percentage of the signal switching from cycle to cycle. For example, the PA bus is 22 bits, so an SF of 0.5 means that only 11 bits of the bus switch from low to high or from high to low between consecutive cache accesses. Figure 2.10 illustrates the power consumed by distributing the PA bus and state vector, which is mostly switching the wire capacitance. This process constitutes 63 % of the total active power consumed by the CAM tag of the L1 data cache. This makes the CAM-based tag more dependent on the data-switching factor. The magnitude of the SRAM-based

tag's dynamic power is mostly due to gate switching and accessing data from the SRAM block, which is implemented as a small-signal array. The biggest power contributor in the SRAM tag implementation is to the process of doing 16 comparisons of 22-bit (35 % of the total power).

Figure 2.11 shows a comparison of the two tag power implementations with a different SF. The graph shows that, for SF of 0.6, both tag implementations consume the same dynamic power. A smaller switching factor is more favorable to the CAM-based tag, with about 60 % less power consumed that the SRAM-based tag when SF=0.25. This key trend makes the decision between CAM versus SRAM tags dependent on the processor architecture and workload. For example, a shared cache for multicore- or fine-grained multithreading-based SOC will have a high SF on the PA bus due to its running of different programs. On the other hand, a single-issue general-purpose processor will have less activity on the PA bus, which makes the CAM-based tag more power-efficient.

2.6.2.5 Summary Tag Selection

Deciding on the tag array used in the memory subsystem has significant implications on power, area, and speed. In our analysis, we showed that CAM-based tags always are larger in area (constituting about 10–20 % of the total cache). Since memory subsystems constitute more than 50 % of the area in modern processors, this characteristic makes the CAM-based tag area overhead to the total processor area between 5 and 10 %. CAM-based tags have more timing challenges than SRAM tags due to the increase in area and the nature of the hit signal being physically distributed; recall that the hit signal is relatively localized in SRAM-based tags. Using CAM-based tags limits the banking options and affects the data array organization; column muxing and routing resources become commonplace. The advantage of CAM tags is that they are more power-efficient than SRAM-based tags, but only for processors with low switching activity factor on the physical address and state bits. This makes it architecture- and workload-dependent, and these characteristics need to be weighed before choosing one tag over the other.

With technology scaling, the impact of wire capacitance and leakage current on both area and speed becomes increasingly important. The SRAM arrays contain more than 90 % of the device and use 50 % of the chip area. Tag array itself consumes more than half the power of the memory subsystems. Hence, early planning and thorough understanding of all the factors that contribute to the power, area, and speed in SRAM memory access is also essential to making the right tag selection.

http://www.springer.com/978-1-4614-8880-4

Embedded Memory Design for Multi-Core and Systems on Chip

Mohammad, B.

2014, XIII, 95 p. 63 illus., 37 illus. in color., Hardcover
ISBN: 978-1-4614-8880-4