XBOX 360 SYSTEM ARCHITECTURE

THIS ARTICLE COVERS THE XBOX 360'S HIGH-LEVEL TECHNICAL REQUIREMENTS, A SHORT SYSTEM OVERVIEW, AND DETAILS OF THE CPU AND THE GPU. THE AUTHORS DESCRIBE THEIR ARCHITECTURAL TRADE-OFFS AND SUMMARIZE THE SYSTEM'S SOFTWARE PROGRAMMING SUPPORT.

Jeff Andrews
Nick Baker
Microsoft Corp.

•••••• Microsoft's Xbox 360 game console is the first of the latest generation of game consoles. Historically, game console architecture and design implementations have provided large discrete jumps in system performance, approximately at five-year intervals. Over the last several generations, game console systems have increasingly become graphics supercomputers in their own right, particularly at the launch of a given game console generation.

The Xbox 360, pictured in Figure 1, contains an aggressive hardware architecture and implementation targeted at game console workloads. The core silicon implements the product designers' goal of providing game developers a hardware platform to implement their next-generation game ambitions. The core chips include the standard conceptual blocks of CPU, graphics processing unit (GPU), memory, and I/O. Each of these components and their interconnections are customized to provide a user-friendly game console product.

Design principles

One of the Xbox 360's main design principles is the next-generation gaming principle—that is, a new game console must provide value to customers for five to seven years. Thus, as for any true next-generation game console hardware, the Xbox 360 delivers a huge discrete jump in hardware performance for gaming.

The Xbox 360 hardware design team had to translate the next-generation gaming principle into useful feature requirements and next-generation game workloads. For the game workloads, the designers' direction came from interaction with game developers, including game engine developers, middleware developers, tool developers, API and driver developers, and game performance experts, both inside and outside Microsoft.

One key next-generation game feature requirement was that the Xbox 360 system must implement a 720p (progressive scan) pervasive high-definition (HD), 16:9 aspect ratio screen in all Xbox 360 games. This feature's architectural implication was that the Xbox 360 required a huge, reliable fill rate.

Another design principle of the Xbox 360 architecture was that it must be flexible to suit the dynamic range of game engines and game developers. The Xbox 360 has a balanced hardware architecture for the software game pipeline, with homogeneous, reallocatable hardware resources that adapt to different game genres, different developer emphases, and even to varying workloads within a frame of a game. In contrast, heterogeneous hardware resources lock software game pipeline performance in each stage and are not reallocatable. Flexibility helps make the design "futureproof." The Xbox 360's three CPU cores, 48 unified shaders, and 512-Mbyte DRAM main memory will enable developers

Figure 1. Xbox 360 game console and wireless controller.

to create innovative games for the next five to seven years.

A third design principle was programmability; that is, the Xbox 360 architecture must be easy to program and develop software for. The silicon development team spent much time listening to software developers (we are hardware folks at a software company, after all). There was constant interaction and iteration with software developers at the very beginning of the project and all along the architecture and implementation phases.

This interaction had an interesting dynamic. The software developers weren't shy about their hardware likes and dislikes. Likewise, the hardware team wasn't shy about where next-generation hardware architecture and design were going as a result of changes in silicon processes, hardware architecture, and system design. What followed was further iteration on planned and potential workloads.

An important part of Xbox 360 programmability is that the hardware must present the simplest APIs and programming models to let game developers use hardware resources effectively. We extended programming models that developers liked. Because software developers liked the first Xbox, using it as a working model was natural for the teams. In listening to developers, we did not repackage or include hardware features that developers did not like, even though that may have simplified the hardware implementation. We considered the software tool chain from the very beginning of the project.

Another major design principle was that the Xbox 360 hardware be optimized for achievable performance. To that end, we designed a scalable architecture that provides the greatest usable performance per square millimeter while remaining within the console's system power envelope.

As we continued to work with game developers, we scaled chip implementations to result in balanced hardware for the software game pipeline. Examples of higher-level implementation scalability include the number of CPU cores, the number of GPU shaders, CPU L2 size, bus bandwidths, and main memory size. Other scalable items represented smaller optimizations in each chip.

Hardware designed for games

Figure 2 shows a top-level diagram of the Xbox 360 system's core silicon components. The three identical CPU cores share an 8-way set-associative, 1-Mbyte L2 cache and run at 3.2 GHz. Each core contains a complement of four-way single-instruction, multiple data (SIMD) vector units.[1] The CPU L2 cache, cores, and vector units are customized for Xbox 360 game and 3D graphics workloads.

The front-side bus (FSB) runs at 5.4 Gbit/pin/s, with 16 logical pins in each direction, giving a 10.8-Gbyte/s read and a 10.8-Gbyte/s write bandwidth. The bus design and the CPU L2 provide added support that allows the GPU to read directly from the CPU L2 cache.

As Figure 2 shows, the I/O chip supports abundant I/O components. The Xbox media audio (XMA) decoder, custom-designed by Microsoft, provides on-the-fly decoding of a large number of compressed audio streams in hardware. Other custom I/O features include

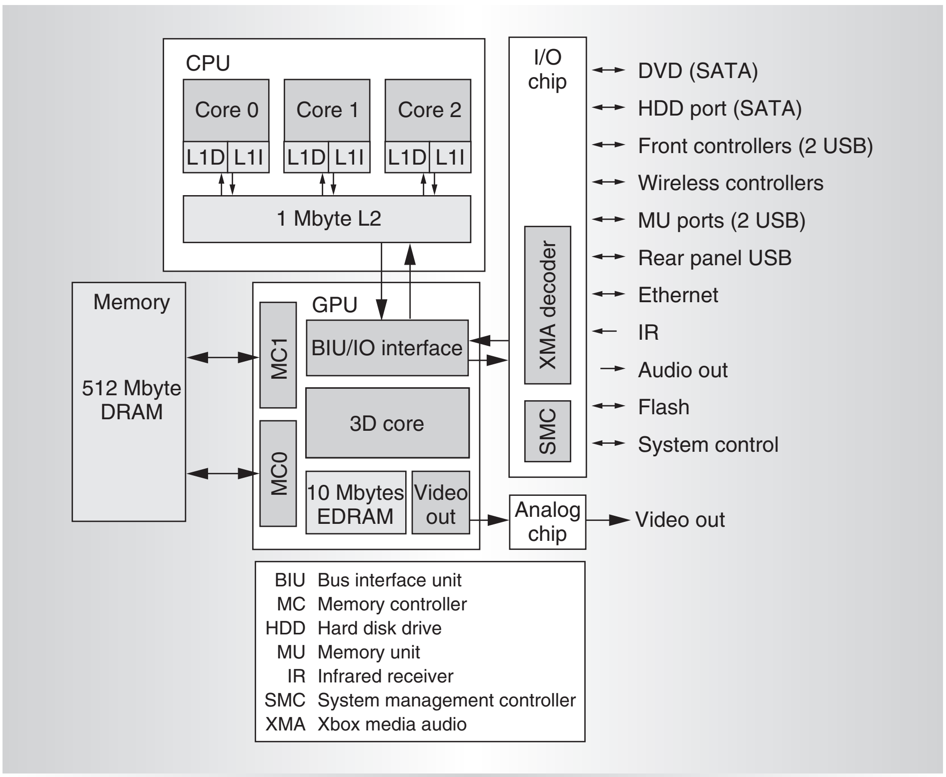


Figure 2. Xbox 360 system block diagram.

the NAND flash controller and the system management controller (SMC).

The GPU 3D core has 48 parallel, unified shaders. The GPU also includes 10 Mbytes of embedded DRAM (EDRAM), which runs at 256 Gbytes/s for reliable frame and z-buffer bandwidth. The GPU includes interfaces between the CPU, I/O chip, and the GPU internals.

The 512-Mbyte unified main memory controlled by the GPU is a 700-MHz graphics-double-data-rate-3 (GDDR3) memory, which operates at 1.4 Gbit/pin/s and provides a total main memory bandwidth of 22.4 Gbytes/s.

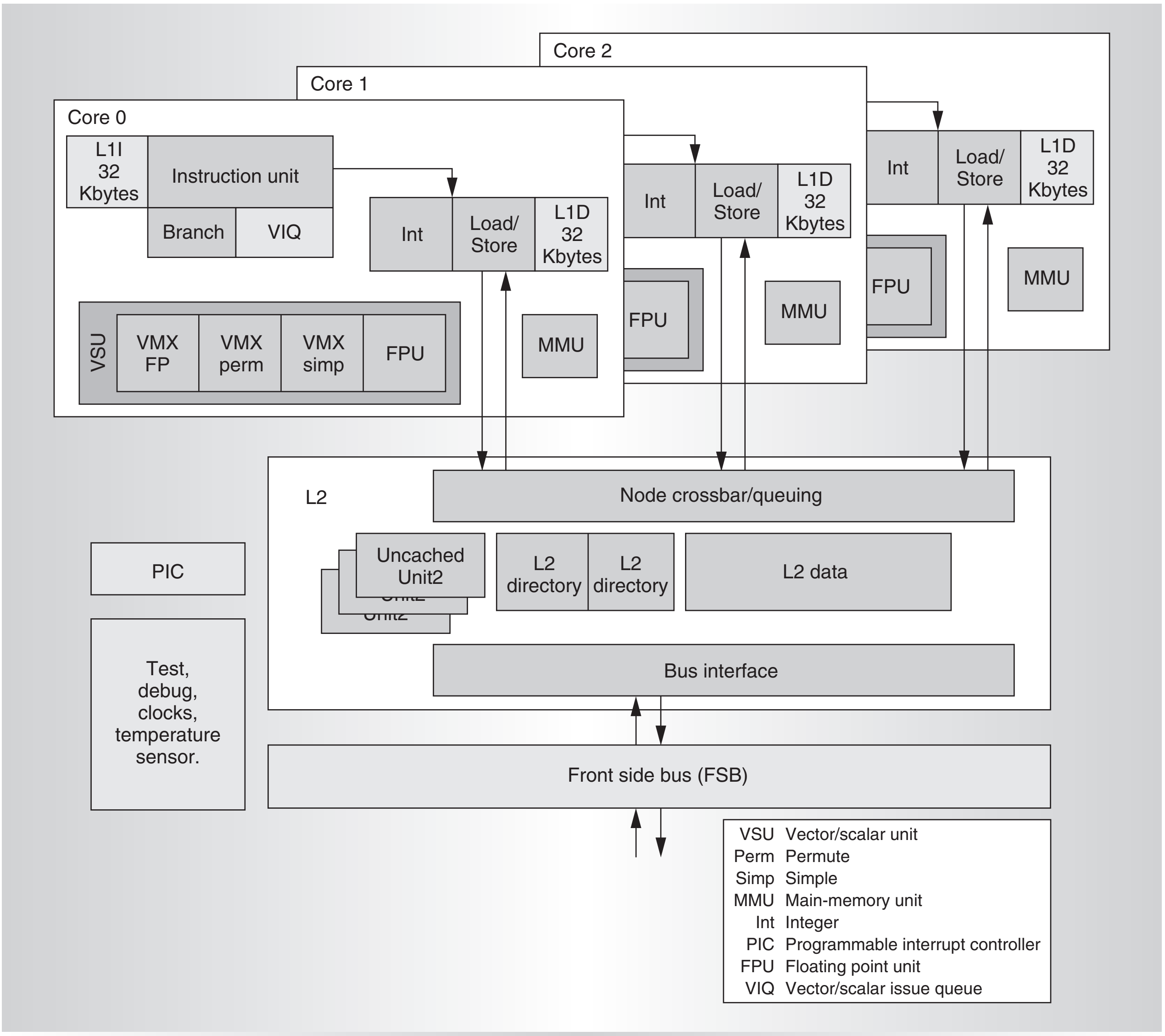
The DVD and HDD ports are serial ATA (SATA) interfaces. The analog chip drives the HD video out.

CPU chip

Figure 3 shows the CPU chip in greater detail. Microsoft's partner for the Xbox 360 CPU is IBM. The CPU implements the PowerPC instruction set architecture,[2-4] with the VMX SIMD vector instruction set (VMX128) customized for graphics workloads.

The shared L2 allows fine-grained, dynamic allocation of cache lines between the six threads. Commonly, game workloads significantly vary in working-set size. For example, scene management requires walking larger, random-miss-dominated data structures, similar to database searches. At the same time, audio, Xbox procedural synthesis (described later), and many other game processes that require smaller working sets can run concurrently. The shared L2 allows workloads needing larger working sets to allocate significantly more of the L2 than would be available if the system used private L2s (of the same total L2 size) instead.

The CPU core has two-per-cycle, in-order instruction issuance. A separate vector/scalar issue queue (VIQ) decouples instruction issuance between integer and vector instructions for nondependent work. There are two symmetric multithreading (SMT),[5] fine-grained hardware threads per core. The L1 caches include a two-way set-associative, 32-Kbyte L1 instruction cache and a four-way set-associative, 32-Kbyte L1 data cache. The write-through data cache does not allocate cache lines on writes.



VSU	Vector/scalar unit
Perm	Permute
Simp	Simple
MMU	Main-memory unit
Int	Integer
PIC	Programmable interrupt controller
FPU	Floating point unit
VIQ	Vector/scalar issue queue

Figure 3. Xbox 360 CPU block diagram.

The integer execution pipelines include branch, integer, and load/store units. In addition, each core contains an IEEE-754-compliant scalar floating-point unit (FPU), which includes single- and double-precision support at full hardware throughput of one operation per cycle for most operations. Each core also includes the four-way SIMD VMX128 units: floating-point (FP), permute, and simple. As the name implies, the VMX128 includes 128 registers, of 128 bits each, per hardware thread to maximize throughput.

The VMX128 implementation includes an added dot product instruction, common in graphics applications. The dot product implementation adds minimal latency to a multiply-add by simplifying the rounding of intermediate multiply results. The dot product instruction takes far less latency than discrete instructions.

Another addition we made to the VMX128 was direct 3D (D3D) compressed data formats,[6-8] the same formats supported by the GPU. This allows graphics data to be generated in the CPU and then compressed before being stored in the L2 or memory. Typical use of the compressed formats allows an approximate 50 percent savings in required bandwidth and memory footprint.

CPU data streaming

In the Xbox, we paid considerable attention to enabling data-streaming workloads, which are not typical PC or server workloads. We added features that allow a given CPU core to execute a high-bandwidth workload (both read and write, but particularly write), while avoiding thrashing its own cache and the shared L2.

First, some features shared among the CPU cores help data streaming. One of these is 128-byte cache line sizes in all the CPU L1 and L2 caches. Larger cache line sizes increase FSB and memory efficiency. The L2 includes a cache-set-locking functionality, common in embedded systems but not in PCs.

Specific features that improve streaming bandwidth for writes and reduce thrashing include the write-through L1 data caches. Also, there is no write allocation of L1 data cache lines when writes miss in the L1 data cache. This is important for write streaming because it keeps the L1 data cache from being thrashed by high bandwidth transient write-only data streams.

We significantly upgraded write gathering in the L2. The shared L2 has an uncached unit for each CPU core. Each uncached unit has four noncached write-gathering buffers that allow multiple streams to concurrently gather and dump their gathered payloads to the FSB yet maintain very high uncached write-streaming bandwidth.

The cacheable write streams are gathered by eight nonsequential gathering buffers per CPU core. This allows programming flexibility in the write patterns of cacheable very high bandwidth write streams into the L2. The write streams can randomly write within a window of a few cache lines without the writes backing up and causing stalls. The cacheable write-gathering buffers effectively act as a bandwidth compression scheme for writes. This is because the L2 data arrays see a much lower bandwidth than the raw bandwidth required by a program's store pattern, which would have low utilization of the L2 cache arrays. Data transformation workloads commonly don't generate the data in a way that allows sequential write behavior. If the write gathering buffers were not present, software would have to effectively gather write data in the register set before storing. This would put a large amount of pressure on the number of registers and increase latency (and thus throughput) of inner loops of computation kernels.

We applied similar customization to read streaming. For each CPU core, there are eight outstanding loads/prefetches. A custom prefetch instruction, extended data cache block touch (xDCBT), prefetches data, but delivers to the requesting CPU core's L1 data cache and never puts data in the L2 cache as regular prefetch instructions do. This modification seems minor, but it is very important because it allows higher bandwidth read streaming workloads to run on as many threads as desired without thrashing the L2 cache. Another option we considered for read streaming would be to lock a set of the L2 per thread for read streaming. In that case, if a user wanted to run four threads concurrently, half the L2 cache would be locked down, hurting workloads requiring a large L2 working-set size. Instead, read streaming occurs through the L1 data cache of the CPU core on which the given thread is operating, effectively giving private read streaming first in, first out (FIFO) area per thread.

A system feature planned early in the Xbox 360 project was to allow the GPU to directly read data produced by the CPU, with the data never going through the CPU cache's backing store of main memory. In a specific case of this data streaming, called Xbox procedural synthesis (XPS), the CPU is effectively a data decompressor, procedurally generating geometry on-the-fly for consumption by the GPU 3D core. For 3D games, XPS allows a far greater amount of differentiated geometry than simple traditional instancing allows, which is very important for filling large HD screen worlds with highly detailed geometry.

We added two features specifically to support XPS. The first was support in the GPU and the FSB for a 128-byte GPU read from the CPU. The other was to directly lower communication latency from the GPU back to the CPU by extending the GPU's tail pointer write-back feature.

Tail pointer write-back is a method of controlling communication from the GPU to the CPU by having the CPU poll on a cacheable location, which is updated when a GPU instruction writes an update to the pointer. The system coherency scheme then updates the polling read with the GPU's updated

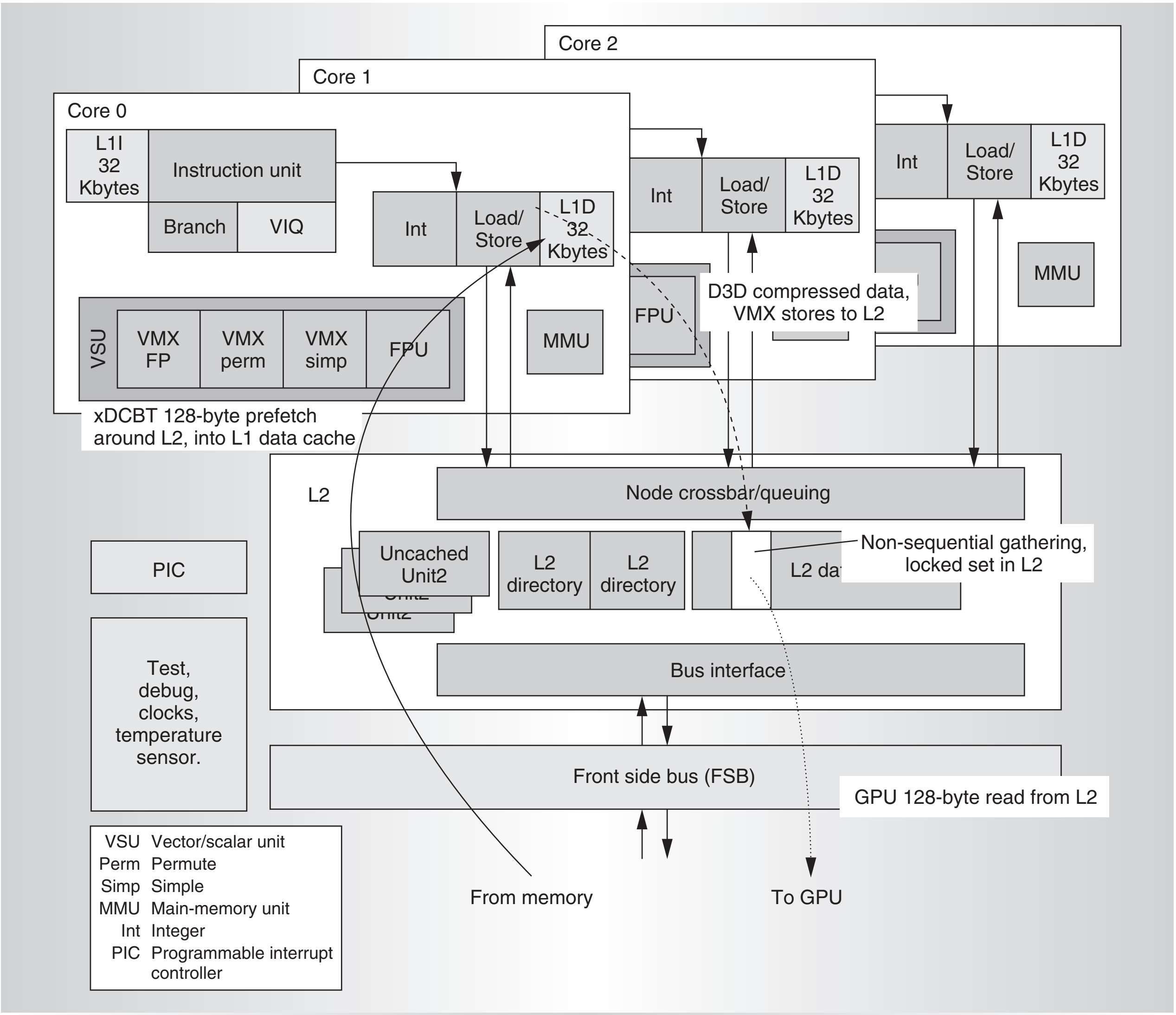


Figure 4. CPU cached data-streaming example.

pointer value. Tail write-backs reduce communication latency compared to using interrupts. We lowered GPU-to-CPU communication latency even further by implementing the tail pointer's backing-store target on the CPU die. This avoids the round-trip from CPU to memory when the GPU pointer update causes a probe and castout of the CPU cache data, requiring the CPU to refetch the data all the way from memory. Instead the refetch never leaves the CPU die. This lower latency translates into smaller streaming FIFOs in the L2's locked set.

A previously mentioned feature very important to XPS is the addition of D3D compressed formats that we implemented in both the CPU and the GPU. To get an idea of this feature's usefulness, consider this: Given a typical average of 2:1 compression and an XPS-targeted 9 Gbytes/s FSB bandwidth, the CPU cores can generate up to 18 Gbytes/s of effective geometry and other graphics data and ship it to the GPU 3D core. Main memory sees none of this data traffic (or footprint).

CPU cached data-streaming example

Figure 4 illustrates an example of the Xbox 360 using its data-streaming features for an XPS workload. Consider the XPS workload, acting as a decompression kernel running on one or more CPU SMT hardware threads. First, the XPS kernel must fetch new, unique data from memory to enable generation of the given piece of geometry. This likely includes world space coordinate data and specific data to make each geometry instance unique. The XPS kernel prefetches this read data during a previous geometry generation iteration to cover the fetch's memory latency. Because none of the per-instance read data is typically reused between threads, the XPS kernel fetches it using the xDCBT prefetch instruction around the L2, which puts it directly into the requesting CPU core's L1 data cache. Prefetching around the L2 separates the read data stream from the write data stream, avoiding L2 cache thrashing. Figure 4 shows this step as a solid-line arc from memory to Core 0's L1 data cache.

The XPS kernel then crunches the data, primarily using the VMX128 computation ability to generate far more geometry data than the amount read from memory. Before the data is written out, the XPS kernel compresses it, using the D3D compressed data formats, which offer simple trade-offs between number of bits, range, and precision. The XPS kernel stores these results as generated to the locked set in the L2, with only minimal attention to the write access pattern's randomness (for example, the kernel places write accesses within a few cache lines of each other for efficient gathering). Furthermore, because of the write-through and no-write-allocate nature of the L1 data caches, none of the write data will thrash the L1 data cache of the CPU core. The diagram shows this step as a dashed-line arc from load/store in Core 0 to the locked set in L2.

Once the CPU core has issued the stores, the store data sits in the gathering buffers waiting for more data until timed out or forced out by incoming write data demanding new 64-byte ranges. The XPS output data is written to software-managed FIFOs in the L2 data arrays in a locked set in the L2 (the unshaded box in Figure 4). There are multiple FIFOs in one locked set, so multiple threads can share one L2 set. This is possible within 128 Kbytes of one set because tail pointer write-back communication frees completed FIFO area with lowered latency. Using the locked set is important; otherwise, high-bandwidth write streams would thrash the L2 working set.

Next, when more data is available to the GPU, the CPU notifies the GPU that the GPU can advance within the FIFO, and the GPU performs 128-byte reads to the FSB. This step is shown in the diagram as the dotted-line arc starting in the L2 and going to the GPU. The GPU design incorporates special features allowing it to read from the FSB, in contrast with the normal GPU read from main memory. The GPU also has an added 128-byte fetch, which enables maximum FSB and L2 data array utilization.

The two final steps are not shown in the diagram. First, the GPU uses the corresponding D3D compressed data format support to expand the compressed D3D formats into single-precision floating-point formats native to the 3D core. Then, the GPU commands tail pointer write-backs to the CPU to indicate that the GPU has finished reading data. This tells the streaming FIFOs' CPU software control that the given FIFO space is now free to be written with new geometry or index data.

Figure 5 shows a photo of the CPU die, which contains 165 million transistors in an IBM second-generation 90-nm silicon-on-insulator (SOI) enhanced transistor process.

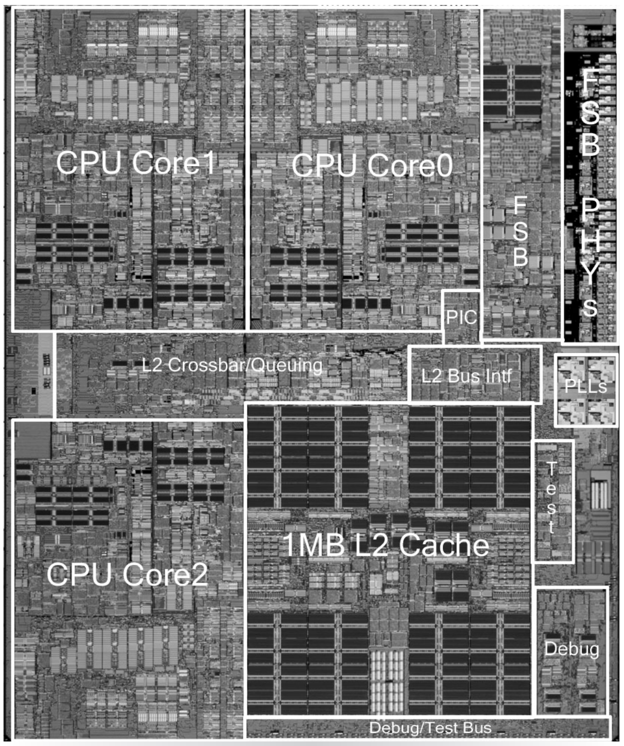


Figure 5. Xbox 360 CPU die photo (courtesy of IBM).

Graphics processing unit

The GPU is the latest-generation graphics processor from ATI. It runs at 500 MHz and consists of 48 parallel, combined vector and scalar shader ALUs. Unlike earlier graphics engines, the shaders are dynamically allocated, meaning that there are no distinct vertex or pixel shader engines—the hardware automatically adjusts to the load on a fine-grained basis. The hardware is fully compatible with D3D 9.0 and High-Level Shader Language (HLSL) 3.0,[9,10] with extensions.

The ALUs are 32-bit IEEE 754 floating-point ALUs, with relatively common graphics simplifications of rounding modes, denormalized numbers (flush to zero on reads), NaN handling, and exception handling. They are capable of vector (including dot product) and scalar operations with single-cycle throughput—that is, all operations issue every cycle. The superscalar instructions encode vector, scalar, texture load, and vertex fetch within one instruction. This allows peak processing of 96 shader calculations per cycle while fetching textures and vertices.

Feeding the shaders are 16 texture fetch engines, each capable of producing a filtered result in each cycle. In addition, there are 16 programmable vertex fetch engines with built-in tessellation that the system can use instead of CPU geometry generation. Finally, there are 16 interpolators in dedicated hardware.

The render back end can sustain eight pixels per cycle or 16 pixels per cycle for depth and stencil-only rendering (used in z-prepass or shadow buffers). The dedicated z or blend logic and the EDRAM guarantee that eight pixels per cycle can be maintained even with 4× antialiasing and transparency. The z-prepass is a technique that performs a first-pass rendering of a command list, with no rendering features applied except occlusion determination. The z-prepass initializes the z-buffer so that on a subsequent rendering pass with full texturing and shaders applied, discarded pixels won't spend shader and texturing resources on occluded pixels. With modern scene depth complexity, this technique significantly improves rendering performance, especially with complex shader programs.

As an example benchmark, the GPU can render each pixel with 4× antialiasing, a z-buffer, six shader operations, and two texture fetches and can sustain this at eight pixels per cycle. This blazing fill rate enables the Xbox 360 to deliver HD-resolution rendering simultaneously with many state-of-the-art effects that traditionally would be mutually exclusive because of fill rate limitations. For example, games can mix particle, high-dynamic-range (HDR) lighting, fur, depth-of-field, motion blur, and other complex effects.

For next-generation geometric detail, shading, and fill rate, the pipeline's front end can process one triangle or vertex per cycle. These are essentially full-featured vertices (rather than a single parameter), with the practical limitation of required memory bandwidth and storage. To overcome this limitation, several compressed formats are available for each data type. In addition, XPS can transiently generate data on the fly within the CPU and pass it efficiently to the GPU without a main memory pass.

The EDRAM removes the render target and z-buffer fill rate from the bandwidth equation. The EDRAM resides on a separate die from the main portion of GPU logic. The EDRAM die also contains dedicated alpha blend, z-test, and antialiasing logic. The interface to the EDRAM macro runs at 256 Gbytes/s: (8 pixels/cycle + 8 z-compares/cycle) × (read + write) × 32 bits/sample × 4 samples/pixel × 500 MHz.

The GPU supports several pixel depths; 32 bits per pixel (bpp) and 64 bpp are the most common, but there is support for up to 128 bpp for multiple-render-target (MRT) or floating-point output. MRT is a graphics technique of outputting more than one piece of data per sample to the effective frame buffer, interleaved efficiently to minimize the performance impact of having more data. The data is used later for a variety of advanced graphics effects. To optimize space, the GPU supports 32-bpp and 64-bpp HDR lighting formats. The EDRAM only supports rendering operations to the render target and z-buffer. For render-to-texture, the GPU must "flush" the appropriate buffer to main memory before using the buffer as a texture.

Unlike a fine-grained tiler architecture, the GPU can achieve common HD resolutions and bit depths within a couple of EDRAM tiles. This simplifies the problem substantially. Traditional tiling architectures typically include a

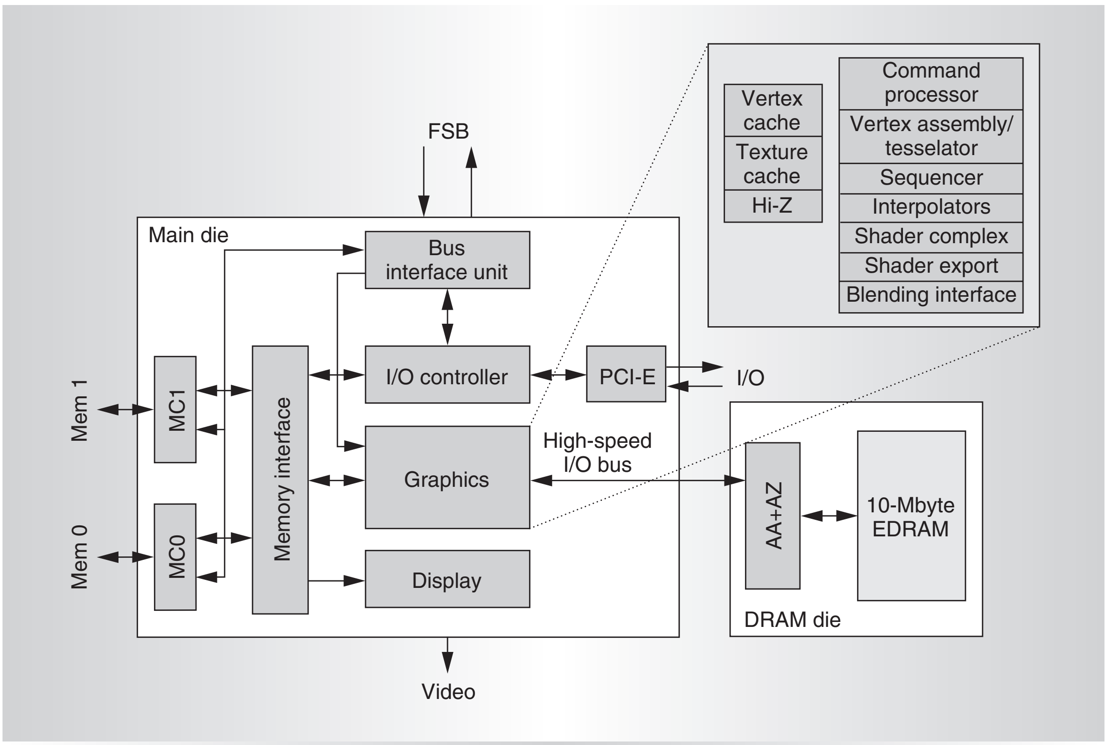


Figure 6. GPU block diagram.

whole process inserted in the traditional graphics pipeline for binning the geometry into a large number of bins. Handling the bins in a high-performance manner is complicated (for example, overflow cases, memory footprint, and bandwidth). Because the GPU's EDRAM usually requires only a couple of bins, bin handling is greatly simplified, allowing more-optimal hardware-software partitioning.

With a binning architecture, the full command list must be presented before rendering. The hardware uses a few tricks to speed this process up. Rendering increasingly relies on a z-prepass to prepare the z-buffer before executing complex pixel shader algorithms. We take advantage of this by collecting object extent information during this pass, as well as priming a full-resolution hierarchical z-buffer. We use the extent information to set flags to skip command list sections not needed within a tile. The full-resolution hi-z buffer retains its state between tiles.

In another interesting extension to normal D3D, the GPU supports a shader export feature that allows data to be output directly from the shader to a buffer in memory. This lets the GPU serve as a vector math engine if needed, as well as allowing multipass shaders. The latter can be useful for subdivision surfaces. In addition, the display pipeline includes an in-line scaler that resizes the frame buffer on the fly as it is output. This feature allows games to pick a rendering resolution to work with and then lets the display hardware make the best match to the display resolution.

As Figure 6 shows, the GPU consists of the following blocks:

- *Bus interface unit.* This interface to the FSB handles CPU-initiated transactions, as well as GPU-initiated transactions such as snoops and L2 cache reads.
- *I/O controller.* Handles all internal memory-mapped I/O accesses, as well as transactions to and from the I/O chip via the two-lane PCI-Express bus (PCI-E).
- *Memory controllers (MC0, MC1).* These 128-byte interleaved GDDR3 memory controllers contain aggressive address tiling for graphics and a fast path to minimize CPU latency.
- *Memory interface.* Memory crossbar and buffering for non-CPU initiators (such as graphics, I/O, and display).

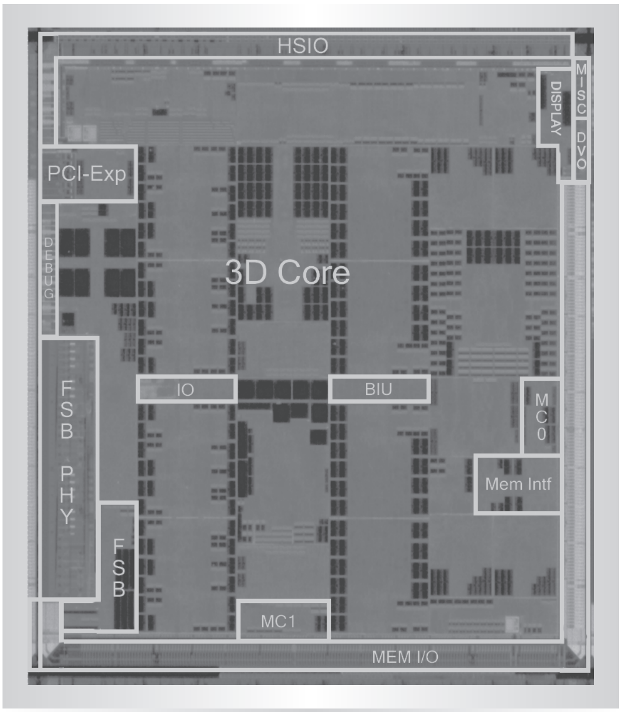


Figure 7. Xbox 360 GPU "parent" die (courtesy of Taiwan Semiconductor Manufacturing Co.).

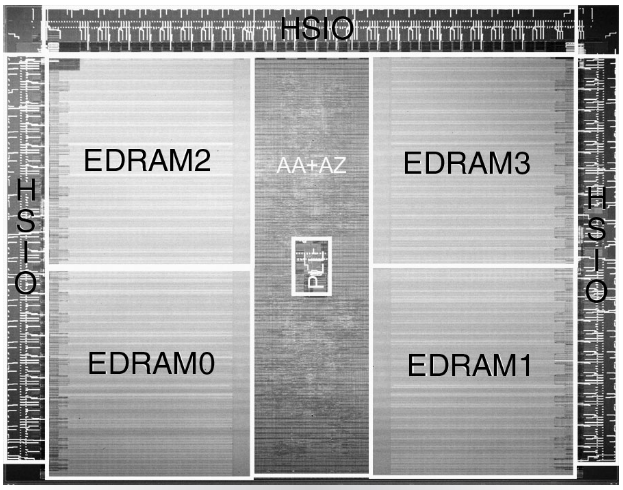


Figure 8. Xbox 360 GPU EDRAM ("daughter") die (courtesy of NEC Electronics).

- *Graphics.* This block, the largest on the chip, contains the rendering engine.
- *High-speed I/O bus.* This bus between the graphics core and the EDRAM die is a chip-to-chip bus (via substrate) operating at 1.8 GHz and 28.8 Gbytes/s. When multisample antialiasing is used, only pixel center data and coverage information is transferred and then expanded on the EDRAM die.
- *Antialiasing and Alpha/A (AA+AZ).* Handles pixel-to-sample expansion, as well as z-test and alpha blend.
- *Display.*

Figures 7 and 8 show photos of the GPU "parent" and EDRAM ("daughter") dies. The parent die contains 232 million transistors in a TSMC 90-nm GT. The EDRAM die contains 100 million transistors in an NEC 90-nm process.

Architectural choices

The major choices we made in designing the Xbox 360 architecture were to use chip multiprocessing (CMP), in-order issuance cores, and EDRAM.

Chip multiprocessing

Our reasons for using multiple CPU cores on one chip in Xbox 360 was relatively straightforward. The combination of power consumption and diminishing returns from instruction-level parallelism (ILP) is driving the industry in general to multicore. CMP is a natural twist on traditional symmetric multiprocessing (SMP), in which all the CPU cores are symmetric and have a common view of main memory but are on the same die versus separate chips. Modern process geometries afford hardware designers the flexibility of CMP, which was usually too costly in die area previously. Having multiple cores on one chip is more cost-effective. It enables shared L2 implementation and minimizes communication latency between cores, resulting in higher overall performance for the same die area and power consumption.

In addition, we wanted to optimize the architecture for the workload, optimize in-game utilization of silicon area, and keep the system easy to program. These goals made CMP a good choice for several reasons:

First, for the game workload, both integer and floating-point performance are important. The high-level game code is generally a database management problem, with plenty of object-oriented code and pointer manipulation. Such a workload needs a large L2 and high integer performance. The CMP shared L2 with its fine-grained, dynamic allocation means this workload can use a large working set in the L2 while running. In addition, several sections of the application lend themselves well to vector floating-point acceleration.

Second, to optimize silicon area, we can take advantage of two factors. To start with, we are presenting a stable platform for the product's lifetime. This means tools and programming expertise will mature significantly, so we can rely more on generating code than optimizing performance at runtime. Moreover, all Xbox 360 games (as opposed to Xbox games from Microsoft's first game console, which are emulated on Xbox 360) are compiled from scratch and optimized for the current microarchitecture. We don't have the problem of running legacy, but compatible, instruction set architecture executables that were compiled and optimized for a completely different microarchitecture. This problem has significant implications for CPU microarchitectures in PC and server markets.

Third, although we knew multicore was the way to go, the tools and programming expertise for multithread programming are certainly not mature, presenting a problem for our goal of keeping programming easy. For the types of workloads present in a game engine, we could justify at most six to eight threads in the system. The solution was to adapt the "more-but-simpler" philosophy to the CPU core topology. The key was keeping the number of hardware threads limited, thus increasing the chance that they would be used effectively. We decided the best approach was to tightly couple dedicated vector math engines to integer cores rather than making them autonomous. This keeps the number of threads low and allows vector math routines to be optimized and run on separate threads if necessary.

In-order issuance cores

The Xbox 360 CPU contains three two-issue, in-order instruction issuance cores. Each core has two SMT hardware threads, which support fine-grained instruction issuance. The cores allow out-of-order execution in the common cases of loads and vector/floating-point versus integer instructions. Loads, which are treated as prefetches, don't stall until a load dependency is present. Vector and floating-point operations have their own, decoupled vector/float issue queue (VIQ), which decouples vector/floating point versus integer issuance in many cases.

We had several reasons for choosing in-order issuance. First, the die area required by in-order-issuance cores is less than that of out-of-order-issuance cores. In-order cores simplify issue logic considerably. Although not directly a big area user, out-of-order issue logic can consume extra area because it requires additional pipeline stages to meet clock period timing. Further, common implementations of out-of-order issuance and completion use rename registers and completion queues, which take significant die area.

Second, in-order implementation is more power efficient than out-of-order implementation. Keeping power levels manageable was a major issue for the design team. All the additional die area required for out-of-order issuance consumes power. Out-of-order cores commonly increase performance because their issuance, tracking, and completion enable deeper speculative instruction execution. This deeper speculation means wasted power since whole execution strings are often thrown away. Xbox 360 execution does speculate but to a lesser degree.

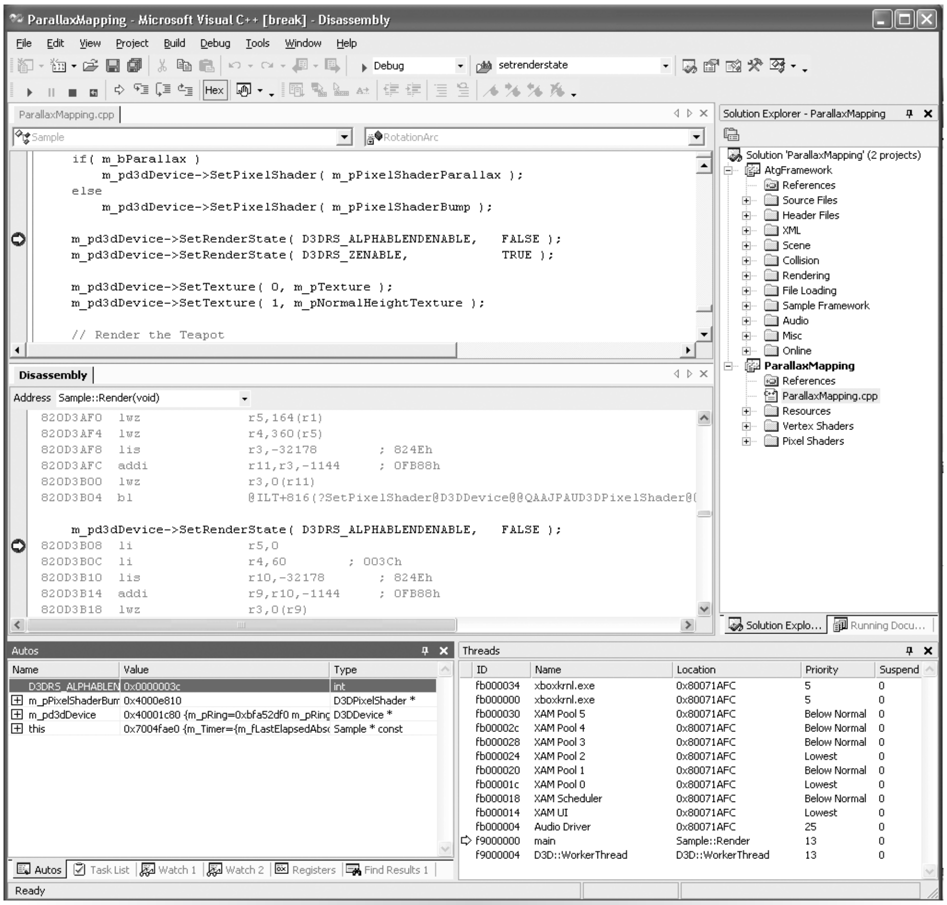
Third, the Xbox 360's two SMT hardware threads per core keep the execution pipelines more fully utilized than they are in traditional in-order designs. This helps keep the execution pipelines busy without out-of-order issuance.

Finally, in-order design is simpler, aiding design and implementation. Simplicity also makes performance more predictable, simplifying programming and tool optimizations.

EDRAM

HD, alpha blending, z-buffering, antialiasing, and HDR pixels take a heavy toll on memory bandwidth. Although more effects are being achieved in the shaders, postprocessing effects still require a large pixel-depth complexity. Also as texture filtering improves, texel fetches can consume large amounts of memory bandwidth, even with complex shaders.

One approach to solving this problem is to use a wide external memory interface. This limits the ability to use higher-density memory technology as it becomes available, as well as requiring compression. Unfortunately, any compression technique must be lossless, which means unpredictable—generally not



```
    if( m_bParallax )
        m_pd3dDevice->SetPixelShader( m_pPixelShaderParallax );
    else
        m_pd3dDevice->SetPixelShader( m_pPixelShaderBump );

    m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE,   FALSE );
    m_pd3dDevice->SetRenderState( D3DRS_ZENABLE,            TRUE );

    m_pd3dDevice->SetTexture( 0, m_pTexture );
    m_pd3dDevice->SetTexture( 1, m_pNormalHeightTexture );

    // Render the Teapot
```

Disassembly

Address Sample::Render(void)

```
820D3AF0  lwz              r5,164(r1)
820D3AF4  lwz              r4,360(r5)
820D3AF8  lis              r3,-32178        ; 824Eh
820D3AFC  addi             r11,r3,-1144     ; 0FB88h
820D3B00  lwz              r3,0(r11)
820D3B04  bl               @ILT+816(?SetPixelShader@D3DDevice@@QAAJPAUD3DPixelShader@(

    m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE,   FALSE );
820D3B08  li               r5,0
820D3B0C  li               r4,60        ; 003Ch
820D3B10  lis              r10,-32178       ; 824Eh
820D3B14  addi             r9,r10,-1144     ; 0FB88h
820D3B18  lwz              r3,0(r9)
```

Autos

Name	Value	Type
D3DRS_ALPHABLEN	0x0000003c	int
m_pPixelShaderBum	0x4000e810	D3DPixelShader *
m_pd3dDevice	0x40001c80 {m_pRing=0xbfa52df0 m_pRing	D3DDevice *
this	0x7004fae0 {m_Timer={m_fLastElapsedAbs	Sample * const

Threads

	ID	Name	Location	Priority	Suspend
	fb000034	xboxkrnl.exe	0x80071AFC	5	0
	fb000000	xboxkrnl.exe	0x80071AFC	5	0
	fb000030	XAM Pool 5	0x80071AFC	Below Normal	0
	fb00002c	XAM Pool 4	0x80071AFC	Below Normal	0
	fb000028	XAM Pool 3	0x80071AFC	Below Normal	0
	fb000024	XAM Pool 2	0x80071AFC	Lowest	0
	fb000020	XAM Pool 1	0x80071AFC	Below Normal	0
	fb00001c	XAM Pool 0	0x80071AFC	Lowest	0
	fb000018	XAM Scheduler	0x80071AFC	Below Normal	0
	fb000014	XAM UI	0x80071AFC	Lowest	0
	fb000004	Audio Driver	0x80071AFC	25	0
⇨	f9000000	main	Sample::Render	13	0
	f9000004	D3D::WorkerThread	D3D::WorkerThread	13	0

Figure 9. Multithreaded debugging in the Xbox 360 development environment.

good for game optimization. In addition, the required bandwidth would most likely require using a second memory controller in the CPU itself, rather than having a unified memory architecture, further reducing system flexibility.

EDRAM was the logical alternative. It has the advantage of completely removing the render target and the z-buffer bandwidth from the main-memory bandwidth equation. In addition, alpha blending and z-buffering are read-modify-write processes, which further reduce the efficiency of memory bandwidth consumption. Keeping these processes on-chip means that the remaining high-bandwidth consumers—namely, geometry and texture—are now primarily read processes. Changing the majority of main-memory bandwidth to read requests increases main-memory efficiency by reducing wasted memory bus cycles caused by turning around the bidirectional memory buses.

Software

By adopting SMP and SMT, we're using standard parallel models, which keep things simple. Also, the unified memory architecture allows flexible use of memory resources.

Our operating system opens all three cores to game developers to program as they wish. For this, we provide standard APIs including Win32 and OpenMP, as well as D3D and HLSL. Developers can also bypass these and write their own CPU assembly and shader microcode, referred to in the game industry as "to the metal" programming.

We provide standard tools including XNA-based tools Performance Investigator (PIX) and Xbox Audio Creation Tool (XACT). XNA is Microsoft's game development platform, which developers of PC and Xbox 360 games (as well as other platforms) can use to minimize cross-platform development costs.[11] PIX is the graphics profiler and debugger.[12] It uses performance counters embedded in the CPU and GPU and architectural simulators to provide performance feedback.

The Xbox 360 development environment is familiar to most programmers. Figure 9 shows a screen shot from the XNA Studio Integrated Development Environment (IDE), a version of Visual Studio with additional features for game developer teams. Programmers use IDE for building projects and debugging, including debugging of multiple threads. When stepping through source code, programmers find the instruction set's low-level details completely hidden, but when they open the disassembly window, they can see that PowerPC code is running.

Other powerful tools that help Xbox 360 developers maximize productivity and performance include CPU profilers, the Visual C++ 8.0 compiler, and audio libraries. These tools and libraries let programmers quickly exploit the power of Xbox 360 chips and then help them code to the metal when necessary.

Xbox 360 was launched to customers in the US, Canada, and Puerto Rico on 22 November, 2005. Between then and the end of 2005, it launched in Europe and Japan. During the first quarter of 2006, Xbox 360 has been launched in Central and South America, Southeast Asia, Australia, and New Zealand.

Xbox 360 implemented a number of firsts and/or raised the bar for performance for users of PC and game console machines for gaming. These include

- the first CMP implementation with more than 2 cores (3);
- the highest frequency and bandwidth CPU frontside bus (5.4 Gbps and 21.6 GB/s);
- the first CMP chip with shared L2;
- the first game console with SMT;
- the first game console with GPU-unified shader architecture; and
- the first game console with MCM GPU/EDRAM die implementation.

The Xbox 360 core chipset contains approximately 500 M transistors. It is the most complex, highest performance consumer electronic product shipping today and presents a large discrete jump in 3D graphics and gaming performance.

References

1. *PowerPC Microprocessor Family: AltiVec Technology Programming Environments Manual*, version 2.0, IBM Corp., 2003.
2. E. Silha et al., *PowerPC User Instruction Set Architecture, Book I*, version 2.02, IBM Corp., 2005.
3. E. Silha et al., *PowerPC Virtual Environment Architecture, Book II*, version 2.02, IBM Corp., 2005.
4. E. Silha et al., *PowerPC Operating Environment Architecture, Book III*, version 2.02, IBM Corp., 2005.
5. J. Hennessey and D. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed., Morgan Kaufmann, 2002.
6. K. Gray, *The Microsoft DirectX 9 Programmable Graphics Pipeline*, Microsoft Press, 2003.
7. F. Luna, *Introduction to 3D Game Programming with DirectX 9.0*, 1st ed., Wordware, 2003.
8. MSDN DX9 SDK Documentation, Direct3D Overview, 2005, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/dx9_graphics.asp.
9. S. St.-Laurent, *The Complete Effect and HLSL Guide*, Paradoxal Press, 2005.
10. MSDN DX9 SDK Documentation, HLSL Shaders Overview, 2005, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/HLSL_Workshop.asp.
11. Microsoft XNA, 2006, http://www.microsoft.com/xna.
12. MSDN DX9 SDK Documentation, PIX Overview, 2005, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/PIX.asp.

Jeff Andrews is a CPU architect and project leader in Microsoft's Xbox Console Architecture Group, focusing on the Xbox 360 launch CPU. Andrews has a BS in computer engineering from the University of Illinois Urbana-Champaign.

Nick Baker is the director of Xbox console architecture at Microsoft. His responsibilities include managing the Xbox Console Architecture, System Verification, and Test Software teams. Baker has an MS in electrical engineering from Imperial College London.

Direct questions and comments about this article to Jeff Andrews, Microsoft, 1065 La Avenida St., Mountain View, CA 94043; jeffrey.andrews@microsoft.com.