Survey Report on Low-Power Embedded Multimedia Processors

Final Report for the Course:
Architectures for Multimedia Systems - 079759
Politecnico di Milano (Milano, Italy)
Academic Year 2006-2007

Authors: Oscar Barreto, Franciscus Bonetto, Christian Cantelmo, Marco Caprani, Gianfranco Longi, Alessandro Origgi, Luca Piccina, Franco Rampoldi, Roberto Teruzzi

Supervisor: prof. Cristina Silvano, Politecnico di Milano (Italy)

This report is a survey of the most important commercial processor architectures targeted to the embedded, low-power and multimedia application domain.
The first part of this report is dedicated to a set of “strictly-embedded” processors, i.e., the ARM processor architecture and the Trimedia processor core. These are the most predominant processor architectures available in the embedded domain.
The second part of this report is dedicated to graphical processor pipelines. These architectures do not present classical “embedded” features, however, they fall under the general concept of multimedia processor.
The third part of this report is dedicated to high-performance processors. These processors are mostly used in the Desktop and Server markets. This category has been inserted into this report because high-performance hardware techniques are percolating down in the embedded domain where the performance figure of merit is becoming more and more important. Moreover, the problem of low-power consumption is becoming increasingly important in this class of processor architectures, especially when considering high-density server configurations where suitable processor cooling is more and more difficult to achieve.
Table of Contents

Table of Contents ................................................................................................................ 2
1 The ARM processor .................................................................................................... 5
  1.1 Original ARM architecture .......................................................... 5
  1.2 ARM Architectural innovations .............................................. 7
  1.3 ARM Cortex Processor Family ............................................. 9
    1.3.1 Arm Cortex A8 ............................................................ 10
2 The Philips Trimedia Media-Processor .......................................................... 17
  2.1 VLIW Approach ........................................................................... 17
  2.2 Trimedia ISA Instructions ...................................................... 18
  2.3 Prefetching .................................................................................. 19
  2.4 Trimedia Pipeline .......................................................................... 20
  2.5 Implementation ........................................................................... 21
  2.6 Power Consumption and Performance ........................................ 22
3 The XBOX 360 Architecture .......................................................................... 25
  3.1 System Architecture Overview .................................................. 25
  3.2 The Xenon Processor ................................................................. 26
    3.2.1 Xenon design principles: procedural synthesis ..................... 27
    3.2.2 Xenon design principles: TLP & ILP ................................... 28
    3.2.3 Xenon design principles: Cache read/write streaming ....... 28
    3.2.4 Xenon Pipeline ................................................................. 29
4 The AMD ATI R600 GPU ................................................................................. 31
  4.1 Architecture ................................................................................. 32
    4.1.1 Command Processor .......................................................... 33
    4.1.2 Setup Engine ....................................................................... 34
    4.1.3 Ultra Threaded Dispatch Processor ..................................... 34
    4.1.4 Threading and Branching: ................................................... 35
    4.1.5 Shader Units ........................................................................ 36
    4.1.6 Texture Units ........................................................................ 37
    4.1.7 The memory controller ........................................................ 38
5 The Nvidia G80 Architecture ............................................................................ 40
  5.1 Scalar processor design ............................................................... 41
  5.2 Decoupled Shader Math and Texture Operations ......................... 41
  5.3 Branching Efficiency Improvements ........................................... 42
  5.4 Raw Processing and Texturing Filtering Power ........................... 42
  5.5 ROP and Memory Subsystems .................................................. 43
  5.6 Power consumption and heat ...................................................... 43
6 The CELL processor ......................................................................................... 45
  6.1 Power Processing Element ......................................................... 45
  6.2 Synergistic Processor Element .................................................. 48
    6.2.1 Local Store .......................................................................... 49
    6.2.2 Memory flow control .......................................................... 49
  6.3 Element Interconnect Bus ........................................................... 50
  6.4 Memory and I/O .......................................................................... 51
7 The Intel Core 2 Duo ......................................................................................... 52
7.1 Architecture ....................................................................................................... 52
7.2 Advanced Digital Media Boost ......................................................................... 53
7.3 Instruction decoding .......................................................................................... 53
7.4 Instruction Fusion ............................................................................................. 54
7.5 Smart memory access ....................................................................................... 54
8 The Intel Itanium ..................................................................................................... 57
8.1 EPIC support in the Itanium .......................................................................... 57
8.2 The Core Pipeline ............................................................................................. 59
8.3 Memory Subsystem ......................................................................................... 63
Embedded processors
1 The ARM processor

Formed in 1990, ARM (previously, the Advanced RISC Machine, and prior to that Acorn RISC Machine) licenses Intellectual Property (IP) processor designs to Partner companies. These industry-leading providers of semiconductors, systems, software and design tools utilize ARM technology as essential building blocks for the microprocessors, peripherals and SoCs (System-On-Chip) they develop and manufacture. Strictly speaking, ARM Ltd does not manufacture and sell CPU devices based on their own designs, but rather, licenses the processor architecture to interested parties. ARM offers a variety of licensing terms, varying in cost and deliverables. To all licensees, ARM provides an integratable hardware description of the ARM core, as well as complete software development toolset (compiler, debugger, SDK), and the right to sell manufactured silicon containing the ARM CPU. Like most IP vendors, ARM prices its IP based on perceived value. In architectural terms, the lower performance ARM cores command a lower license cost than the higher performance cores.

ARM's business has always been to sell IP cores, which licensees use to create microcontrollers and CPUs based on this core. ARM's processor technology has been licensed by many of the world's leading semiconductor company's. The most successful implementation has been the ARM7TDMI with hundreds of millions sold in several kind of microcontroller equipped device (such as the ARM7TDMI in the iPod).

1.1 Original ARM architecture

The ARM architecture is a 32-bit RISC processor architecture developed by ARM Limited that is widely used in a number of embedded designs. The ARM processor range provides solutions for:

- Open platforms running complex operating systems for wireless, consumer and imaging applications.
- Embedded real-time systems for mass storage, automotive, industrial and networking applications.
- Secure applications including smart cards and SIMs.

Today, the ARM family accounts for over 75% of all 32-bit embedded CPUs, making it one of the most prolific 32-bit architectures in the world. Because of their power saving features, ARM CPUs are dominant in the mobile electronics market, where low power consumption is a critical design goal. ARM CPUs are found in several corners of consumer electronics, from portable devices (PDAs, mobile phones, media players, handheld gaming units, and calculators) to computer peripherals (hard drives, desktop routers).
The ARM Processors Families

The original ARM architecture includes the following RISC features:

- Load/store architecture
- No support for misaligned memory accesses (now supported in ARMv6 cores)
- Orthogonal instruction set
- 16 general purpose registers of 32 bits (1 being the PC)
- Fixed opcode width of 32 bits to ease decoding and pipelining, at the cost of decreased code density
- Mostly single-cycle execution

Other interesting features are:

- Use of a 4-bit condition code on the front of every instruction, meaning that execution of every instruction is optionally conditional. This cuts down significantly on the encoding bits available for displacements in memory access instructions, but on the other hand it avoids branch instructions when generating code for small if statements. This conditional execution of most instructions reduces branch overhead and compensates for the lack of a branch predictor.
- Arithmetic instructions alter condition codes only when desired
- Second register of all arithmetic, logical, and register-register move operations can be shifted or rotated. In fact, in the ARM ISA there are instructions like “Rotate-and-add” or “shift-left-and-add”. This results in the typical ARM program being denser than expected with less memory access thus the pipeline is used more efficiently.
- 32-bit barrel shifter which can be used without performance penalty with most arithmetic instructions and address calculations
- Powerful indexed addressing modes: PC-relative addressing (indeed, on the ARM the PC is one of its 16 registers) and pre- and post-increment addressing modes.
- Simple, but fast, 2-priority-level interrupt subsystem with switched register banks
- 12-bit immediate field interpretation: 8 LSBs are zero extended to 32 bit, rotated right the number of bits specified in the first 4 bits of the field multiplied by two. In this way, it can represent all power of two in a 32 bit word.
• Block load/store: save and restore blocks of registers on function call/return in one cycle;
• Block memory copy
• Full set of coprocessor instructions: the architecture provides a non-intrusive way of extending the instruction set using "coprocessors", which can be addressed using MCR, MRC, MRRC and MCRR commands from software. The coprocessor space is divided logically into 16 coprocessors with numbers from 0 to 15, coprocessor 15 (cp15) being reserved for some typical control functions like managing the caches and MMU operation (on processors that have one).

Even though the ARM runs at what many would consider to be low speeds, it nevertheless competes quite well with much more complex CPU designs. Another item of note is that the ARM has been around for a while, with the instruction set increasing somewhat over time. Some early ARM processors (prior to ARM7TDMI), for example, have no instruction to load a two-byte quantity.

The ARM7 and most earlier designs have a three stage pipeline; the stages being fetch, decode, and execute. Higher performance designs, such as the ARM9, have a five stage pipeline. Additional changes for higher performance include a faster adder, and more extensive branch prediction logic.

In ARM based machines, peripheral devices are usually attached to the processor by mapping their physical registers into ARM memory space or into the coprocessor space or connecting to another device (a bus) which in turn attaches to the processor. Coprocessor accesses have lower latency so some peripherals (for example XScale interrupt controller) are designed to be accessible in both ways (through memory and through coprocessors).

1.2 ARM Architectural innovations

Other features and improvements of the basic ARM architecture were designed in the last years. It is important to notice that not all the innovations explained below are present in all ARM processors. Usually, a composition of some features is implemented on a given processor.

Some ARM processors have a compressed instruction set, called Thumb, that uses a 16-bit-wide instruction encoding (but still processes 32-bit data). In Thumb, the smaller opcodes have less functionality. For example, only branches can be conditional, and many opcodes cannot access all of the CPU's registers. However, the shorter opcodes give improved code density overall, even though some operations require more instructions. Particularly in situations where the memory port or bus width is constrained to less than 32 bits, the shorter Thumb opcodes allows greater performance than with 32-bit code because of the more efficient use of the limited memory bandwidth. Typically embedded hardware has a small range of addresses of 32-bit datapath and the rest are 16 bits or narrower (e.g. the Game Boy Advance). In this situation, it usually makes sense to compile Thumb code and hand-optimise a few of the most CPU-intensive sections using the (non-Thumb) 32-bit instruction set, placing them in the limited 32-bit bus width
memory. The first processor with a Thumb instruction decoder was the ARM7TDMI. All ARM9 and later families, including XScale, have included a Thumb instruction decoder. ARM has implemented a technology, Jazelle DBX (Direct Bytecode eXecution), that allows certain of their architectures to accelerate execution of Java bytecode in hardware, as another execution state, providing acceleration for some bytecodes while calling out to special software for bytecodes it does not support. It interoperates alongside the existing ARM and Thumb states. The first processor with Jazelle technology was the ARM926EJ-S: Jazelle being denoted by the ‘J’ in the CPU name. It is used by mobile phone manufacturers to speed up execution of Java ME games and applications, which is probably what drove development of the technology.

Thumb-2 technology made its debut in the ARM1156 core, announced in 2003. Thumb-2 extends the limited 16-bit instruction set of Thumb with additional 32-bit instructions to give the instruction set more breadth. The resulting stated aim for Thumb-2 is to achieve code density similar to Thumb with performance similar to the ARM instruction set on 32-bit memory. Thumb-2 also extends both the ARM and Thumb instruction set with yet more instructions, including bit-field manipulation, table branches, and conditional execution.

Thumb Execution Environment (shortly ThumbEE), also known as Thumb-2EE, and marketed as Jazelle RCT, was announced in 2005, first appearing in the Cortex-A8 processor. ThumbEE provides a small extension to the Thumb-2 extended Thumb instruction set, making the instruction set particularly suited to code generated at runtime (e.g. by JIT compilation) in managed Execution Environments. ThumbEE is a target for languages such as Limbo, Java, C#, Perl and Python, and allows JIT compilers to output smaller compiled code without impacting performance. New features provided by ThumbEE include automatic null pointer checks on every load and store instruction, an instruction to perform an array bounds check, and the ability to branch to handlers, which are small sections of frequently called code, commonly used to implement a feature of a high level language, such as allocating memory for a new object.

The Advanced SIMD extension, marketed as NEON technology, is a combined 64 and 128 bit SIMD (Single Instruction Multiple Data) instruction set that provides standardized acceleration for media and signal processing applications. NEON can execute MP3 audio decoding on CPUs running at 10 MHz and can run the GSM AMR (Adaptive Multi-Rate) speech codec at no more than 13 MHz. It features a comprehensive instruction set, separate register files and independent execution hardware. NEON supports 8-, 16-, 32- and 64-bit integer and single precision floating-point data and operates in SIMD operations for handling audio/video processing as well as graphics and gaming processing. In NEON, the SIMD supports up to 16 operations at the same time.

VFP technology is a coprocessor extension to the ARM architecture. It provides low-cost single-precision and double-precision floating-point computation fully compliant with the ANSI/IEEE Std 754-1985 Standard for Binary Floating-Point Arithmetic. VFP provides floating-point computation suitable for a wide spectrum of applications such as PDAs,
smartphones, voice compression and decompression, three-dimensional graphics and digital audio, printers, set-top boxes, and automotive applications. The VFP architecture also supports execution of short vector instructions allowing SIMD (Single Instruction Multiple Data) parallelism. This is useful in graphics and signal-processing applications by reducing code size and increasing throughput. Other floating-point and/or SIMD coprocessors found in ARM-based processors include FPA, FPE, iwMMXt. They provide some of the same functionality as VFP but are not opcode-compatible with it.

The Security Extensions, marketed as TrustZone Technology, is found in ARMv6KZ and later application profile architectures. It provides a low cost alternative to adding an additional dedicated security core to a SoC, by providing two virtual processors backed by hardware based access control. This enables the application core to switch between two states (referred to as worlds to reduce confusion with other names for capability domains) in a manner such that information can be prevented from leaking from the more trusted world to the less trusted world. This world switch is generally orthogonal to all other capabilities of the processor and so each world can operate independently of the other while using the same core. Memory and peripherals are then made aware of the operating world of the core and may use this to provide access control to secrets and code on the device. A typical application of TrustZone Technology is to run a rich operating system in the less trusted world, and smaller security-specialized code in the more trusted world.

The AMBA on-chip interconnect is an established, open specification that serves as a framework for SoC designs and IP library development. The AMBA Advanced High-performance Bus (AHB) interface is supported by all many ARM cores and provides a high-performance, fully-synchronous backplane. Multi-layer AHB represents a significant technical advance that reduces latencies and increases the bandwidth available to multi-master systems. Fully compatible with the current AHB specification, Multi-layer AHB is supported on the ARM926EJ-S core and by all members of the ARM10 core family. In 2003, ARM announced the launch of the latest AMBA specification, the AMBA 3 AXI protocol which is targeted at high performance, high clock frequency systems designs and includes a number of features that make it very suitable for high speed, submicron interconnect.

ARM Intelligent Energy Manager (IEM) technology implements advanced algorithms to optimally balance processor workload and energy consumption, while maximizing system responsiveness to meet end-user performance expectations. The Intelligent Energy Manager technology works with the OS and applications running on the mobile phone to dynamically adjust the required CPU performance level through a standard programmer's model.

### 1.3 ARM Cortex Processor Family

The ARM Cortex family, evolution of ARMv7 family, is the latest architecture developed by ARM for the market of embedded processors. The Cortex Family comprises three
series, which all implement the Thumb-2 instruction set to address the increasing performance and cost demands of various markets:

- ARM Cortex-A Series, applications processors for complex OS and user applications. Supports the ARM, Thumb and Thumb-2 instruction sets.
- ARM Cortex-R Series, embedded processors for real-time systems. Supports the ARM, Thumb, and Thumb-2 instruction sets.
- ARM Cortex-M Series, deeply embedded processors optimized for cost sensitive applications. Supports the Thumb-2 instruction set only.

### Cortex family processors summary

<table>
<thead>
<tr>
<th>Feature Summary</th>
<th>Cache Size (Inst/Data)</th>
<th>Tightly Coupled Mgt Memory</th>
<th>Memory</th>
<th>Bus Interface</th>
<th>Thumb, DSP, Jazelle</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM Cortex-A8</td>
<td>Variable -</td>
<td>MMU + TrustZone AMBA 3 AXI</td>
<td>Yes</td>
<td>Yes Yes Yes</td>
<td></td>
</tr>
<tr>
<td>ARM Cortex-M1</td>
<td>Yes -</td>
<td>AMBA AHB-Lite + APB</td>
<td>Yes</td>
<td>No No No</td>
<td></td>
</tr>
<tr>
<td>ARM Cortex-M3</td>
<td>- MPU (optional)</td>
<td>3x AHB-Lite + APB</td>
<td>Yes</td>
<td>No No No</td>
<td></td>
</tr>
<tr>
<td>ARM Cortex-R4(F)</td>
<td>0k-64k Variable MPU</td>
<td>AMBA 3 AXI</td>
<td>Yes</td>
<td>Yes No No</td>
<td></td>
</tr>
</tbody>
</table>

### 1.3.1 Arm Cortex A8

It is the first ARM processor to incorporate all of the new technologies available in the ARMv7 architecture. New technologies seen for the first time include NEON for media and signal processing and Jazelle RCT for acceleration of runtime compilers, such as just-in-time, dynamic or ahead-of-time compilers. Other technologies recently introduced that are now standard on the ARMv7 architecture include TrustZone technology for security, Thumb-2 technology for code density and the VFPv3 floating point architecture. The Cortex-A8 processor is the most sophisticated low-power design yet produced by ARM. To achieve its high levels of performance, new microarchitecture features were added which are not traditionally found in the ARM architecture, including a dual in-order issue ARM integer pipeline, an integrated L2 cache and a deep 13-stage pipe.

The ARM Cortex-A8 architecture description is structured as shown below:

- Superscalar Pipeline;
- Branch Prediction;
- Level-1 Cache;
- Level-2 Cache;
- Neon Media Engine
- Performance Characteristics
Superscalar Pipeline. Perhaps the most significant of the new features in the Cortex-A8 is the dual-issue, in-order, statically scheduled ARM integer pipeline. Previous ARM processors have only a single integer execution pipeline.

The ability to issue two data processing instructions at the same time significantly increases the maximum potential instructions executed per cycle. It was decided to stay with in-order issue to keep additional power required to a minimum. Out-of-order issue and retire can require extensive amounts of logic consuming extra power. The choice to go with in-order also allows for fire-and-forget instruction issue, thus removing critical paths from the design and reducing the need for custom design in the pipeline. Static scheduling allows for extensive clock gating for reduced power during execution.

The dual ALU (arithmetic logic unit) pipelines (ALU 0 and ALU 1) are symmetric and both can handle most arithmetic instructions. ALU pipe 0 always carries the older of a pair of issued instructions. The Cortex-A8 processor also has multiplier and load-store pipelines, but these do not carry additional instructions to the two ALU pipelines. These can be thought of as “dependent” pipelines. Their use requires simultaneous use of one of the ALU pipelines. The multiplier pipeline can only be coupled with instructions that are in ALU 0 pipeline, whereas the load-store pipeline can be coupled with instructions in either ALU.
Branch Prediction. The 13-stage pipeline was selected to enable significantly higher operating frequencies than previous generations of ARM microarchitectures. Note that stage F0 is not counted because it is only address generation. To minimize the branch penalties typically associated with a deeper pipeline, the Cortex-A8 processor implements a two-level global history branch predictor. It consists of two different structures: the Branch Target Buffer (BTB) and the Global History Buffer (GHB) which are accessed in parallel with instruction fetches.

The BTB indicates whether or not the current fetch address will return a branch instruction and its branch target address. It contains 512 entries. On a hit in the BTB a branch is predicted and the GHB is accessed. The GHB consists of 4096 2-bit saturating counters that encode the strength and direction information of branches. The GHB is indexed by 10-bit history of the direction of the last ten branches encountered and 4 bits of the PC.

In addition to the dynamic branch predictor, a return stack is used to predict subroutine return addresses. The return stack has eight 32-bit entries that store the link register value in r14 (register 14) and the ARM or Thumb state of the calling function. When a return-type instruction is predicted taken, the return stack provides the last pushed address and state.

Level-1 Cache. The Cortex-A8 processor has a single-cycle load-use penalty for fast access to the Level-1 caches. The data and instruction caches are configurable to 16k or 32k. Each is 4-way set associative and uses a Hash Virtual Address Buffer (HVAB) way prediction scheme to improve timing and reduce power consumption. The caches are physically addressed (virtual index, physical tag) and have hardware support for avoiding aliased entries. Parity is supported with one parity bit per byte.

The replacement policy for the data cache is write-back with no write allocates. Also included is a store buffer for data merging before writing to main memory. The HVAB is
a novel approach to reducing the power required for accessing the caches. It uses a prediction scheme to determine which way of the RAM to enable before an access.

Level-2 Cache. The Cortex-A8 processor includes an integrated Level-2 cache. This gives the Level-2 cache a dedicated low latency, high bandwidth interface to the Level-1 cache. This minimizes the latency of Level-1 cache linefills and does not conflict with traffic on the main system bus. It can be configured in sizes from 64k to 2M. The Level-2 cache is physically addressed and 8-way set associative. It is a unified data and instruction cache, and supports optional ECC and Parity. Write back, write through, and write-allocate policies are followed according to page table settings. A pseudo-random allocation policy is used. The contents of the Level-1 data cache are exclusive with the Level-2 cache, whereas the contents of the Level-1 instruction cache are a subset of the Level-2 cache. The tag and data RAMs of the Level-2 cache are accessed serially for power savings.

NEON media engine. The Cortex-A8 processor’s NEON media processing engine pipeline starts at the end of the main integer pipeline. As a result, all exceptions and branch mispredictions are resolved before instructions reach it. More importantly, there is a zero load-use penalty for data in the Level-1 cache. The ARM integer unit generates the addresses for NEON loads and stores as they pass through the pipeline, thus allowing data to be fetched from the Level-1 cache before it is required by a NEON data processing operation.

Deep instruction and load-data buffering between the NEON engine, the ARM integer unit and the memory system allow the latency of Level-2 accesses to be hidden for streamed data. A store buffer prevents NEON stores from blocking the pipeline and detects address collisions with the ARM integer unit accesses and NEON loads.
The NEON unit is decoupled from the main ARM integer pipeline by the NEON instruction queue (NIQ). The ARM Instruction Execute Unit can issue up to two valid instructions to the NEON unit each clock cycle. NEON has 128-bit wide load and store paths to the Level-1 and Level-2 cache, and supports streaming from both.

The NEON media engine has its own 10 stage pipeline that begins at the end ARM integer pipeline. Since all mispredicts and exceptions have been resolved in the ARM integer unit, once an instruction has been issued to the NEON media engine it must be completed as it cannot generate exceptions. NEON has three SIMD integer pipelines, a load-store/permute pipeline, two SIMD single-precision floating-point pipelines, and a non-pipelined Vector Floating-Point unit (VFPLite).

NEON instructions are issued and retired in-order. A data processing instruction is either a NEON integer instruction or a NEON floating-point instruction. The Cortex-A8 NEON unit does not parallel issue two data-processing instructions to avoid the area overhead with duplicating the data-processing functional blocks, and to avoid timing critical paths and complexity overhead associated with the muxing of the read and write register ports. The NEON integer datapath consists of three pipelines: an integer multiply/accumulate pipeline (MAC), an integer Shift pipeline, and an integer ALU pipeline. A load-store/permute pipeline is responsible for all NEON load/stores, data transfers to/from the integer unit, and data permute operations such as interleave and de-interleave. The NEON floating-point (NFP) datapath has two main pipelines: a multiply pipeline and an add pipeline.

The separate VFPLite unit is a non-pipelined implementation of the ARM. VFPv3 Floating Point Specification targeted for medium performance IEEE 754 compliant floating point support. VFPLite is used to provide backwards compatibility with existing ARM floating point code and to provide IEEE 754 compliant single and double precision arithmetic. The “Lite” refers to area and performance, not functionality.
Performance characteristics. The numbers below are illustrative of cores implemented using general purpose TSMC process technologies and ARM Artisan Advantage-HS standard cell libraries and Advantage optimized RAM components. All area figures include L2 cache control logic but exclude NEON, ETM and L2 cache RAMs. “Area with cache” figures are for the core with 32Kx32K L1 cache.

The 65nm (LP) dynamic power measured is at 1.2 V nominal and hence is higher than the 65nm(GP) dynamic power which is at 1.0V. However, the 65nm (LP) leakage is significantly lower and this is the major consideration for mobile or battery operated devices which need to conserve power in standby mode.

<table>
<thead>
<tr>
<th>Performance Characteristics</th>
<th>Speed Opt</th>
<th>65 nm (LP)</th>
<th>65 nm (GP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td></td>
<td>0.056</td>
<td>0.110</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>Speed Opt</td>
<td>0.056</td>
<td>0.110</td>
</tr>
<tr>
<td>Area with cache (mm²)</td>
<td>&lt;4</td>
<td>&lt;4</td>
<td></td>
</tr>
<tr>
<td>Area without cache (mm²)</td>
<td>&lt;3</td>
<td>&lt;3</td>
<td></td>
</tr>
<tr>
<td>Power with cache (mW/MHz)</td>
<td>&lt;0.56</td>
<td>&lt;0.46</td>
<td></td>
</tr>
</tbody>
</table>

Performance Characteristics
The Philips Trimedia Media-Processor

The TM3270 is the latest Philips media-processor based on the Trimedia architecture, it addresses the requirement of multi-standard video processing (en-/de-coding) at standard resolution and the associated audio processing requirements.

### Table 1. TM3270 Architecture

<table>
<thead>
<tr>
<th>Architectural Feature</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>5 issue slot VLIW guarded RISC-like operations</td>
</tr>
<tr>
<td>Pipeline depth</td>
<td>7-12 stages</td>
</tr>
<tr>
<td>Address width</td>
<td>32 bits</td>
</tr>
<tr>
<td>Data width</td>
<td>32 bits</td>
</tr>
<tr>
<td>Register-file</td>
<td>Unified, 128 32-bit registers</td>
</tr>
<tr>
<td>Functional units</td>
<td>31</td>
</tr>
<tr>
<td>IEEE-754 floating point</td>
<td>yes</td>
</tr>
<tr>
<td>SIMD capabilities</td>
<td>1 x 32-bit, 2 x 16-bit, 4 x 8-bit</td>
</tr>
<tr>
<td>Instruction cache</td>
<td>64 Kbyte, 128-byte lines, 8 way set-associative, LRU replacement policy</td>
</tr>
<tr>
<td>Data cache</td>
<td>128 Kbyte, 128-byte lines, 4 way set-associative, LRU replacement policy, Allocate-on-write miss policy</td>
</tr>
</tbody>
</table>

Features of the Philips Trimedia Processor

2.1 VLIW Approach

The Very Long Instruction Word or VLIW refers to a CPU architectural approach that takes advantage of explicit instruction level parallelism (ILP).

In this approach, the compiler determines the order of execution of operations (including which operations can execute simultaneously) so the processor does not need the scheduling hardware. As a result, VLIW CPUs offer significant computational power with less hardware complexity (but greater compiler complexity) than is associated with most superscalar CPUs.

VLIW CPUs are usually constructed of multiple RISC-like functional units that operate independently.

Compilers generate initial instruction sequences for the VLIW CPU in roughly the same manner that they do for traditional CPUs, generating a sequence of RISC-like instructions. The compiler analyzes this code for dependence relationships and resource requirements. It then schedules the instructions according to those constraints.

In this process, independent instructions can be scheduled in parallel. Because VLIW's typically represent instructions scheduled in parallel with a longer instruction word that
incorporates the individual instructions, this results in a much longer opcode (thus the term "very long") to specify what executes on a given cycle.

A VLIW instruction may contain up to five operations, which are template-based encoded in a compressed format to limit code size. Every VLIW instruction starts with a 10-bit template field, which specifies the compression of the operations in the next VLIW instruction. As a result, an instruction’s compression template is available one cycle before the instruction’s compressed encoding, which relaxes the timing requirements of the decoding process.

The following figure gives an example of a VLIW instruction containing three operations in slots 2, 3, and 5. Issue slots 1 and 4 are not used, as specified by the “11” encoding of the related compression fields. Since issue slot 1 in not used, the first encoded operation is for issue slot 2.

A VLIW instruction without any operations is efficiently encoded in 2 bytes, with a “11:11:11:11:11” template field.

A VLIW with all operations of the maximum size of 42 bits is encoded in 28 bytes, with a “10:10:10:10:10” template field and 5*42 bits for the operation encoding. This compression scheme allows for an efficient encoding of code with a low amount of instruction level parallelism.

A VLIW Instruction sequence

2.2 Trimedia ISA Instructions

The TM3270 enhances the ISA of its predecessor, the TM3260, with roughly 40 new operations.

The main type of new operations that are important to describe are:

- Two slot operations
- Fractional load operations
- H.264 CABAC decoding operations
Two slot operations are executed by functional units that are situated in two neighboring issue slots. As a result, these operations have twice the register-file bandwidth, allowing for operations with up to four source operands, and up to two destinations operands.

The new fractional operations perform a memory load with interpolation on the retrieved data, which allows for efficient calculation of pixels at fractional horizontal image positions.

As a result, the computational complexity of motion estimation is significantly reduced. The collapsed load operations combine the functionality of an ordinary load operation with a linear interpolation function, as defined by a fractional position.

Performance evaluations of an optimized decoder indicate that a significant part of the overall decoding time may be spent in the CABAC process.

Context-adaptive binary arithmetic coding (CABAC) is a form of entropy coding used in H.264/MPEG-4 AVC video encoding. As such it is an inherently lossless compression technique. It is notable for providing considerably better compression than most other encoding algorithms used in video encoding and is considered one of the primary advantages of the H.264/AVC encoding scheme.

CABAC is only supported in Main and higher profiles and requires a considerable amount of processing to decode compared to other similar algorithms. As a result, Context-adaptive variable-length coding (CAVLC), a lower efficiency entropy encoding scheme, is sometimes used instead to increase performance on slower playback devices.

In the Trimedia TM3270 processor CABAC decoding operations are supported with a good performance in the decoding process. The two most important operations that provide the CABAC decoding are SUPER_CABAC_CTX and SUPER_CABAC_STR.

### 2.3 Prefetching

In the TM3270 prefetch approach is based on memory regions. It allows for a prefetching pattern that reflect the access pattern of a data structure mapped onto a certain address space.

The TM3270 supports four separate memory regions. The identification of these memory regions and the required prefetch patter is under software control and defined by the following parameters \( n = 0, 1, 2, 3 \):

- PFn_START_ADDR
- PFn_END_ADDR
- **PFN_STRIDE**

The first two parameters, PFn_START_ADDR and PFn_END_ADDR, are used to identify a memory region. The third parameter, PFn_STRIDE, is used to specify the prefetch pattern for the associated region. When the processor hardware detects a load from an address A within a prefetch region x, a prefetch request for address A+PFx_STRIDE is sent to the prefetch unit, if the prefetch address is not yet present in the cache. Prefetched data is put directly into the data cache.

![Prefetching scheme on the Trimedia processor](image)

### 2.4 **Trimedia Pipeline**

The TM3270 pipeline has a depth of 7 stages for single cycle latency operations. Stages I1 through I3 hold the sequential instruction cache design: the access of cache tags (stage I1) and cache instruction information (stage I3) proceed in sequence to limit power consumption. Every cycle, a 32-byte aligned chunk of instruction information can be retrieved from the instruction memory. These chunks are stored in a 4-entry instruction buffer in stage P. The instruction buffer decouples the progress of the front-end of the pipeline (stages I1 through I3) from the back-end of the pipeline (stages D through W). In stage P a VLIW instruction is pre-decoded from the information in the instruction buffer.

Stage D decodes the individual operations, and determines operations’ operands through register-file access and operand bypassing. To support the maximum issue rate of five operations per VLIW instruction, the register-file has five 1-bit guard read ports, and ten 32-bit source read ports. Stages X1 through X6 are the execute stages. The amount of execute stages is dependent on an operation’s latency.

Single cycle operations have a single execute stage (X1); collapsed load operations with interpolation have (X1 through X6). Two-slot functional units are depicted in two neighboring issue slots.
Conditional and unconditional jump operations are executed in the X1 stage. Jump operations have five architectural delay slots, reflecting the pipeline distance from the first stage of instruction retrieval (stage I1) to the X1 stage. As a result, no stall cycles are observed during control flow changes, eliminating the need for branch prediction techniques. The scheduler tries to fill the 25 operations in the five jump delay instructions using aggressive predication, possibly based on program profile information.

The load/store unit includes the data cache, and is located in issue slots 4 and 5. Separate cache line refill, copy back, and prefetch units connect this unit to the processor’s bus interface unit (BIU). The BIU is the interface to the rest of the SoC. It includes an asynchronous clock domain transfer, which allows for flexibility when deciding upon processor and SoC operating frequency. Stage W gathers the operation results from the functional units, and allows for up to five simultaneous 32-bit updates to the register-file.

2.5 Implementation

The TM3270 is a fully synthesizable, static design. It uses off-the-shelf single ported SRAM memories. The first realization was in a low power 90 nm process technology. Under worst case operating conditions (125 C, worst-case voltage of 1.08 V, worst case process corner) the processor reaches a frequency of 350 MHz. The following figure gives the processor floorplan, and its partitioning into the major design modules. The SRAM memories were hand-placed. The placement of the standard cell logic was tool-driven; i.e. no labor-intensive hierarchical placement of the modules was required to
come to a placed and routed design. The synthesizability and tool-driven place and route make it relatively easy to port the design to a different process technology.

2.6 Power Consumption and Performance

The low power consumption is fundamental in Soc where the Trimedia TM3270 is used. To obtain a low power consumption there are different techniques. For example the sequential instruction cache design for a 8-way set associative cache greatly reduces the power consumption over a more traditional parallel cache design. Another important technique is that the processor design has been heavily clock-gated; roughly 70 different functional clock domains exist. All stages of all functional units are separately gated: when they are not used, they are not clocked. In this way the system obtained is very low power consumption.

The evaluation of the processor TM3270 compared to his predecessor the TM3260 (using a suite of 50 applications from the media-processing domain) is reported in the following graph where it is possible to see that the introduction of new ISA operations improve performance by 40%, data prefetching improve performance by more 20%.
Relative performance of the Trimedia processor family
Graphic processors
3 The XBOX 360 Architecture

The videogame industry is one of the biggest inside the entertainment market, being console gaming one of the most important branches inside this field. For this, it is not surprise that a lot of effort has been put into making video game consoles each time more and more sophisticated, turning with each step into even more powerful graphical supercomputers.

Nowadays, the XBox 360 is one of the most recent consoles in the market and its technology is an example of how complex have consoles become in the last few years. Analyzing the XBox system gives a good understanding of what are the current trends in game console architectures and what are the most important features to look for in the different console systems currently on the market. This chapter presents a general overview of the XBox 360 system, making special emphasis on the CPU. The first part of this chapter will give a general description of the XBox 360 system architecture. Later, a more detailed description on the main features of the Xenon processor, core of this multimedia system, will be addressed.

3.1 System Architecture Overview

The main component of the XBox 360 system is a single chip CPU (with 165 million transistors), that is a three-way symmetric multiprocessor as showed on the following figure. Each of the CPU cores is a specialized PowerPC chip running at 3.2GHz, and each one with a SIMD Vector Multimedia Extension (VMX128) partially compatible with the VMX instructions in the PowerPC G4 and G5 CPUs.
The three CPU cores share a 1MB 8-way set associative Level2 cache. And each processor counts with 32KB of data and 32KB of instruction Level1 internal cache. Both caches have 128 byte cache line sizes.

The chip's front-side bus/physical interface (FSB) has a 21.6GB/second bandwidth, 10.8 GB/s for write and 10.8 GB/s for read, which can be performed simultaneously. The high frequency clocks are generated on-chip by four phase-locked loops: two for the core clocks, two for the FSB clock.

Regarding the GPU, this one counts with 48 parallel unified (combined vector and scalar) shader ALUs and 10 Mbytes of embedded DRAM (EDRAM), which runs at 256 Gbytes/s for reliable frame and z-buffer bandwidth.

The 512-Mbyte unified main memory controlled by the GPU is a 700-MHz graphics double-data-rate-3 (GDDR3) memory, which 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.

The Xbox 360 CPU chip includes testing and debugging functions, like tracing, configuration control, and performance monitoring features.

The XBox system also counts with some miscellaneous IO ports, such as a JTAG port, a POST monitor, and an interface for a serial EEPROM in case patch logic configuration was needed during bring-up.

### 3.2 The Xenon Processor

The processor for the XBox 360 system was designed by IBM and codenamed as “Waternoose” by IBM and “Xenon” by Microsoft, it is based on the 64 bit PowerPC architecture. It implements the PowerPC instruction set architecture (ISA) with the VMX SIMD vector instruction set customized for graphic workloads. Each of the three cores that compose the Xenon processor (called also PowerPC processing element, or PPE) has a 64KB split L1 cache, this means, 32KB for instructions and 32KB for data. The instruction cache is a two-way set associative cache while the data cache is a four-way set associative cache with store-through, non-allocation on write miss policy, and non-blocking access, so a cache miss doesn’t hold up a subsequent hit.

Also, each PPE is capable of handling up to two simultaneous threads of execution by using simultaneous multithreading, meaning that the Xenon can handle up to six threads simultaneously. The instruction issuance followed is in-order and there is a vector/scalar issue queue (VIQ) that decouples instruction issuance between integer and vector instructions to allow independent work.

As it can be seen on the following figure, each one of the three cores is composed of the following execution units:

- 1 Integer Unit
- 1 Floating Point Unit
- 1 Branch Unit
- 1 Load/Store Unit
- 3 VMX128 Units (Floating-Point, Permute, and Simple)
As implied in the name, the VMX128 counts with 128 registers, each one of 128 bits. Another important feature of the VMX128 unit is the implementation of an added dot product instruction, useful for graphics applications.

As of the Branch Prediction unit, it counts with a 4 KB Branch History Table. Although the Xenon counts with this branch prediction unit, the design of the processor relies on the programmer to use branch hints to get optimal performance out of branch-intensive AI and game control code. This, in combination with loop unrolling compilers can achieve very good performance even on deeply pipelined processors and means an inexpensive way of achieving it.

3.2.1 Xenon design principles: procedural synthesis

One of the major issues in the computer graphics world is the amount of data necessary to represent and render objects on the screen. The 3D models are nothing more than a collection of polygons, which are also a collection of vertices.

In a traditional PC system with a GPU, each model starts out in main memory as stored vertex data. Then, that vertex data is fed from main memory into the GPU (under the direction of the CPU on which the game engine runs), where it is then rendered into a 3D image and output to the monitor as a sequence of frames. For realistic scenes and objects, there is a lot of vertex data that needs to be put on main memory and pushed to the GPU, which means that storing all of that vertex data would take a massive amount of system memory, and moving all of that data into the GPU would take a massive amount of bus bandwidth.

The approach followed by the XBox is to have the CPU procedurally generate the geometry (i.e. the vertex data) of the scene on the fly. To do this, the information that represents the objects on screen is stored on main memory in a compressed way, and then it is passed to the CPU, which generates the vertex data and sends this information to the GPU.

There are many “compression” techniques used, for example 3D surfaces can be described in a more compact way by using higher order curves like Bezier curves, b-splines, etc. Since the GPU can not render this curves, it can only render sets of vertices, the CPU is in charge of transforming this data and deliver it to the GPU.
In this way, the problem of huge amount of main memory space and the extremely fast bandwidth to access it, needed for lots of vertex data, is solved, since the CPU to GPU bandwidth and computing power to procedurally render objects in real time is quite high.

### 3.2.2 Xenon design principles: TLP & ILP

The Xenon processor does not count with a hardware instruction window. In order to extract parallelism from the code stream, the processor must first pool instructions in a special set of buffers, called instruction window. The processor then checks for dependencies in the instructions inside the instruction window, in order to determine which instructions can be executed in parallel.

In the Xenon, instructions are issued in the order they were fetched, and only if two adjacent instructions are independent they are executed in parallel when possible, given the execution units available (FPU, integer unit, etc).

Instead of spending hardware on an instruction window that looks for ILP at run-time, the Xenon relies on the programmer to structure the code stream at compile time so that it contains a high level of thread-level parallelism (TLP). This allows a simpler hardware and frees transistors that can be used to implement more execution units on the die.

The TLP strategy works very well for tasks that can be parallelized at the thread level, like procedural synthesis. However, it doesn’t work as well as an ILP system with wide execution core plus a large instruction window for inherently single-threaded tasks. There are three types of game oriented tasks that are likely to suffer from the lack of an out-of-order processing: game control, artificial intelligence (AI), and physics. Each of these three tasks tends to be relegated to a single thread, and tend to have some inherent ILP that can be exploited.

### 3.2.3 Xenon design principles: Cache read/write streaming

One may think the shared 1MB L2 cache is relatively small for a three core CPU capable of running 6 threads at the same time. Nevertheless, the type of media applications that the Xenon is designed to run tend not to use the cache in a very efficient way. Multimedia applications stream data at high rates and the data that is placed in the cache is not reused. Given that the whole point of cache memory is to reuse data, this makes large caches not very useful for data intensive applications like games. Taking this into consideration, instead of including a big cache, designers opted for a different approach on the cache implementation.

Level 1 and level 2 caches work between the CPU and main memory. They replicate the contents of main memory in order to reduce the amount of time that the CPU has to wait to retrieve code and data from main memory. The process of moving code and data between main memory and cache and keeping them in synchrony is usually managed by the hardware, without the programmer intervention. However, in Xenon’s L1 and L2 caches, in addition to this functionality, they can also work in a particular way in order to increase performance of computer graphical applications as specified by programmers.

In a conventional system, main memory produces the vertex data that represents the graphical models, and the GPU consumes it. In the XBox, the CPU can produce decompressed vertex data at a much higher rate than the main memory, often at rate even faster than the GPU can consume. When this happens, the data needs to be stored...
somewhere while it waits to be consumed by the GPU. This data is stored in the Xenon shared L2 cache.

The XBox system architecture allows the programmer to reserve certain space on the cache to act as a FIFO queue, to store output of a data generation thread. The data generation thread feeds its vertex data output directly into this FIFO queue (bypassing the L1 cache entirely), and then the GPU reads that vertex data directly from this queue using a modified DMA protocol. This mode is known as write streaming mode, the other new mode that the programmer can specify is the read streaming mode. In this mode data can be fed directly to the L1 cache or to the registers (bypassing the L2 cache or the entire cache). This is useful to provide a way of preventing streaming media data (typical in multimedia applications) that won’t be reused, from polluting the caches.

### 3.2.4 Xenon Pipeline

The two parallel issued instructions can go to one of five execution pipes: Branch (which is really part of the instruction unit), Load/Store, Fixed Point, Floating Point, and VMX. Difficult instructions are implemented through microcode. At dispatch they are cracked and converted into multiple micro-ops. Also, the Fixed Point pipe can be described as two separate units: one to handle the simple operations like (add/sub, cmp, logical ops, and rotate), and one to handle the complex operations like multiply/divide.

The diagram bellow shows the complete pipeline structure. The first 4 stages are the fetch stages, where the instructions are moved into the PPE’s front end and some early decoding is done to identify the branch instructions. After this fourth cycle, branches that must be predicted are sent to the branch prediction hardware. The last two cycles in the pipe fetch phase are an early decoding phase and/or buffering stage. Then, the following cycles, in yellow, correspond to the main instruction decoding stage and then issuing stages (IS). From here, instructions follow either the Vector/Scalar operations pipe, the Branch pipeline, the Integer operations pipeline, or the Address Generation and Integer Load/Store. The Vector/Scalar path pipeline is longer, as can be observed from the diagram. Scalar double precision floating point operations have an added 10 cycle latency, while VMX operations have from 4 to 14 cycles latency, depending on the operation. The stages in pink are register file read stages, where operands move from the registers into the appropriate execution unit.
The Xenon pipeline
4 The AMD ATI R600 GPU

The R600 GPU is the new graphic processor produced by AMD/ATI. It is the evolution of the previous GPU (the R500 series) and the Xenos processor, made for Xbox 360 console. This GPU is the DirectX 10 compliant graphic processor of ATI, as the G80 is for nVidia factory.

R600 is a unified, fully-threaded, self load-balancing shading architecture, that complies with the specification for DirectX Shader Model 4.0. The major design goals of the chip are high ALU throughput and maximum latency hiding, achieved via the shader core, the threading model and distributed memory accesses via the chip's memory controller. DirectX technology and API specification are the basis for the GPU architecture definition, so we introduce some key features of the new DirectX 10, in order to better understand the R600 technology:

- **Unified shader model.** Known as Shader Model 4.0, it uses a consistent instruction set across all shader types now. Previously, pixel shader code looked different from vertex shader code. This played well on earlier graphics architectures, which had separate pixel and vertex shader units. This unified ISA (instruction set architecture) makes life much easier for graphics programmers. The new shader model has some other key features, like integer operations.

- **Geometry shaders.** In the past, vertex data would come into the 3D pipeline and be manipulated directly, vertex-by-vertex. Vertex shader operations may modify the characteristics of the existing vertex data, but no new vertices are created, and they can't operate on more than one vertex at a time. DirectX 10 adds a new stage, *geometry shaders*, which lies between vertex and pixel operations. It's possible to actually create new vertices or destroy existing ones with geometry shaders, and to perform operations on more than a single vertex. The Geometry Shader's input is an entire primitive (vertex, line, or triangle) with adjacency information, so it can operate on groups of vertices at once. This is one of the new features of DirectX 10 – geometry shaders can be used to do nifty tricks like some limited forms of tessellation, rapid rendering of shadow volumes, fast rendering to cube maps, and other tricks that can greatly speed up graphics tricks that are too costly or slow to use under DirectX 9.

- **Stream out.** Stream output means that data generated from geometry shaders (or sometimes vertex shaders) can be shipped back to video memory, then reused and shipped back to the GPU to be processed again. This allows for more complex geometries, advanced lighting and even more efficient GPU-based physics simulations.

- **Longer shader programs.** Shader programs can now be 64K (65,535) instructions long, instead of the previous 512 instructions.

- **More flexible shader programming.** Flow control is now completely dynamic, and the number of constant registers has dramatically increased (from 256 to 16 x 4096, broken down into 16 input registers and 4096 temp registers). Load and integer ops now exist. While still very focused on graphics, the ISA is beginning to offer more general-purpose attributes.
**Improved instancing.** Instancing was a technique whereby a single DirectX draw call would send an object to the GPU, with some additional data that told the GPU where to place the object, what color it might be, animation data, and so on. But that created multiple objects that often looked and behaved nearly the same. That data was used to replicate the single object many times in a scene using just one draw call. In DX10, these data are in the form of array indices that point back to arrays. Each array entry contains different instance data, so the multiple objects can look and behave differently.

**Windows Vista only.** DirectX 10 will only run on Windows Vista, because it will require the new device driver model that's one of the core pieces of Vista. However, DirectX 10 capable *hardware* will run applications written for earlier versions of DirectX just fine, and will run DX9 and earlier apps on Windows XP.

Another goal of the developer of modern GPUs is to create a scalable architecture that may be expanded or cut down to create GPUs for different price segments. This is in order to save production costs: to have the same production line for every model of video board based on the same kind of processor helps in saving and optimizing effort, costs, maintenance of machines (only one assembly line).

### 4.1 Architecture

Let's take a high-level look at the new chip architecture. The very first unit that data must pass through when entering the GPU is the command processor that is responsible for interacting with the stream of data from the graphics driver. AMD is claiming that the upgraded unit can offload as much as 30% of the CPU overhead in batch improvements from the driver thus allowing for better overall performance.

The setup engine is the unit responsible for preparing the data and organizing it for processing by the various SPUs (stream processing units). There are three functions one for each type of processing work the SPUs might do: vertex assembly and tessellation, geometry assembly and scan conversion for pixel shaders. Each of these can submit threads to the dispatch processor below it.

The ultra-threaded dispatch processor is one of the most complex parts of the new R600 architecture. The main function here is to send to processing the various "threads" that have been created that include a list of instructions to be executed on some data (be in textures, pixels, etc).

The 320 stream processors are divided into four 80-SPU blocks dependent on a set of arbiters and sequencers responsible for selecting the thread to submit. In connection with the SIMD arrays, the dual arbiters allow two operations at a time to be processed by the SPUs, thus indicating a superscalar architecture. Each of these threads can be "bumped" and their states saved in order to allow more critical threads to pass through and resume later at the dispatchers. To aid in latency hiding, the ability for a thread to be "bumped" keeps it from stalling the pipeline if it is waiting for memory access before it can continue processing.

Dedicated caches exist for shader constants and instructions to allow for unlimited shader program length. There are also read/write caches for interthread communication. The
stream out feature allows the shaders to bypass the ROPs and color buffer or to output sequential data instead of bitmaps; useful for that whole GPGPU thing.

The R600 pipeline

### 4.1.1 Command Processor
The command processor accepts the command stream from the graphics driver. It also performs state validation on code sent to the GPU, ensuring the shader units are correctly configured before the data is sent to them. This helps to reduce CPU load, as code is validated before it gets to the setup engine. Previously the driver was entirely responsible for state checking, but D3D10 compliance mandates the hardware do at least some of the job to reduce CPU overhead when drawing. Combined with runtime overhead reduction in D3D10 as a whole and the simple task of asking the GPU to do some work consumes less host cycles than before.
**4.1.2 Setup Engine**

Once the code has been validated, the Setup Engine can allocate the right resources to the shader program, making sure the shader units are setup to receive either vertex, geometry or pixel shader instructions. The vertex assembler is responsible for arranging and storing vertex data optimally in memory. This is to make sure that fetching vertex data from local memory is not limited by available memory bandwidth.

In addition, the vertex assembler is responsible for organizing and feeding data into the programmable tessellation unit. This unit allows the developer to take a simply poly mesh and subdivide it using one of several subdivision surface types. A vertex evaluation shader instruction is used to determine the maximum tessellation factor (which can be up to 15 times), meaning that there is no need for API support. The tessellator not only improves image quality by adding more details into the scene, but it also can improve performance and conserve memory space by using a very low resolution model that’s sent through the tessellation unit to add more details.

There are also assemblers for both geometry and pixel shader instructions too. The former handles either entire primitives or gathers the required data to generate new primitives, while the latter handles scan conversions, rasterization and interpolation. Data from the pixel shader assembler can be passed into the hierarchical-Z and stencil buffers for evaluation to determine whether or not the pixels will be visible. If they are not visible, the geometry is discarded before rasterization in order to make optimal use of the shader units – there’s no point processing a pixel if it’s never going to be displayed on screen. Once the threads have been created by the Setup Engine, they’re submitted to the dispatch processor.

**4.1.3 Ultra Threaded Dispatch Processor**

The dispatch processor is essentially R600’s scheduler – it determines where and when threads need to move to other parts of the GPU. There are command queues for each one of the three different component assemblers (vertex, geometry and pixel) that fill with threads waiting for a free shader unit.

Each SIMD (or shader cluster) has its own pair of arbiters and sequencers in the dispatch processor that dynamically allocates threads to their respective shader clusters in an interleaved manner. In addition, there are dedicated arbiter and sequencer units for both texture and vertex fetch functions too – these can run independently of the shader cluster arbiters and sequencers. Together, these units are responsible for dynamically allocating threads to the shader units, ensuring that they are kept busy.

In this kind of architecture, hundreds of threads are in flight at any point in time. The thread dispatcher has to make smart choices, as a wrong decision could potentially lead to a pipeline stall or an increase the latency, because an instruction is waiting for data from memory and holding up other instructions in the queue.

In order to avoid this, the threaded dispatch processor can actually suspend instructions mid-flight if it’s waiting for data, allowing another thread to be processed with the smallest amount of latency possible. The suspended thread returns to its respective command queue to wait for the required data to arrive. Once the data arrives, the thread can then continue its path through the shader clusters. We now see more in deep the threading arbitration system.
4.1.4 Threading and Branching:

What we call the cluster dispatch processor is what controls execution across the SIMD (single instruction multiple data) clusters arrangement in the shader core, and there is a similar processor for the sampler array. Those logic blocks control the threading model that R600 implements, to hide latency and take advantage of unit pipelining in order to maintain instructions and data throughput.

We'll go through the shader core's processing first. Input from the setup engine fills up a triplet of command queues, one for each thread type, containing the threads the dispatch hardware runs on the shader core. Each cluster contains a pair of arbiters that run a pair of object threads at a time, per cluster, allowing four clocks of execution before new threads are swapped on and run in place. Thread tracking for execution is controlled by a scoreboard system that lets the hardware run threads out of order on the cluster, tracking dependencies and other parameters (likely the operations being run, and registers being written to and read from) to decide what gets executed next, in place of the currently running threads.

Threads undergoing arbitration also have granular priority, affecting the decision making for what's being run. The basic heuristics are designed to put shader threads waiting for sampler data to sleep, to cover latency and run shading operations unhindered while that's taking place, and it's what you'll find any heavily parallel, heavily threaded design doing in terms of its threading model. The sequencer pair (one per arbiter) inside the dispatch...
processor is there to keep track of where a thread is at in terms of its block of execution. Data from the sequencer can feed back into the arbiter to let it know when a thread is about to finish running, so that new threads can be prepared to take a finishing thread's place.

Like competing architectures, R600 will scale back the number of threads in flight at any given time when there's severe register pressure, so that there's no thread stalling because the register file would otherwise be full. The four SIMD arrays on the chip operate independently, so branch granularity is determined by the width of the SIMD and the depth of the pipeline. For pixel shaders, the effective "width" of the SIMD should typically be 16 pixels, since each stream processor block can process a single four-component pixel (with a fifth slot available for special functions or other tasks). The stream processor units are pipelined with eight cycles of latency, but as we've noted, they always execute two threads at once. That makes the effective instruction latency per thread four cycles, which brings us to 64 pixels of branch granularity for R600.

Instruction blocks and constants are kept by the dispatch processor on chip inside dedicated, virtualised cache memories. They're there to maximum efficiency and let the hardware grab thread state as quickly as possible when executing. Any cache miss here will force the thread that needs the data to sleep, and another thread is swapped and the needed data fetched into the cache so it's ready when it wakes. A cache miss can also force a thread's priority to be reduced, effectively moving it down the command queue, so that it might not be woken up as the next thread behind the one that took its place when it missed.

In terms of threads in flight just for the shader core, the hardware maintains an usual count of thousands (which can cover tens of thousands of objects), depending of course on the resources they want to use. In terms of the sampling threads, the heuristics are simpler because there's no register pressure to account for, just the requirement that the right data is in the right cache at the right time, so that a re-fetch because of a cache miss doesn't occur and chip throughput is high, hiding the latency of the data fetch and any post-fetch filtering.

The threading model working well is also a function of the memory controller's ability to marshal data around the chip from the main pools, including the DRAM pool on the board, to the clients that want to consume data, and store it.

### 4.1.5 Shader Units

The shader core in AMD’s R600 uses five-way superscalar shader processors. These are arranged in clusters of 16 shaders, or 80 stream processing units counting the ALUs individually.

Each ALU can run a separate op per clock, R600 exploiting instruction-level parallelism via VLIW. The VLIW design packs a possible 6 instructions per-clock, per-shader unit (5 shading plus 1 branch) into the complete instructions it issues to the shader units, and those possible instruction slots have to match the capabilities of the hardware underneath. The new shader processor of the R600 chip incorporates 5 scalar ALUs capable of executing one floating-point MAD (Multiply-ADD) instruction per cycle, and one ALU can also execute transcendental instructions like SIN, COS, LOG, EXP, etc. The sixth unit in the shader processor is a branch execution unit responsible for executing flow
control instructions (comparisons, loops, subroutine calls). The ALUs are split in terms of
gates for floating and integer logic, too. There's no 32-bit mantissa in the ALU to support
both, but only one datapath in and out of the sub-ALU, so no parallel processing there.
AMD says that this helps to “practically eliminate flow control performance overhead.”
In addition, each shader unit has its own array of dedicated general purpose registers that
are there to store input data, temporary values and output data. In addition, there is a
single 64KB memory read/write fully associative cache that can be accessed by any of
the shader clusters. Data inside this cache can be exported directly into the stream out
buffer – a new feature in DirectX 10 that allows developers to write directly to memory
from shader to memory without having to go out through the render backend. Access to
the register file is also cached, read and write, by an 8KiB multi-port cache. The cache
lets the hardware virtualise the register file, effectively presenting any entry in the cache
as any entry in the larger register file.

4.1.6 Texture Units
The R600 texture processor is a highly integrated device configured like follows:

- 8 texture address units to calculate the address to sample
- 20 texture samplers
- 4 texture filter units

So you add all that up, and the R600 has 32 processors to control texture lookups, 80 to
fetch texture samples (and feeding the SPUs units), and 16 to perform floating-point
texture filtering.

Four texture units operate independently of the four shader clusters, so we have 16
Texture elaborated per clock cycle. The units can each do eight texture addresses per
clock, of which four are used for unfiltered lookups and the other four are used for
bilinear lookups. Also, each texture unit can fetch 20 FP32 textures for bilinear filtering
and point sampling, while also being able to apply bilinear filtering to four FP16 textures
every clock cycle. AMD says that bilinear filtering FP32 textures is done at half speed.
The units also support trilinear and anisotropic filtering on all formats.

In terms of memory access from the sampler hardware, sampler units aren't tied to certain
clusters as such, rather certain positions inside the cluster. If you visualize the 16 shader
units in a cluster as being four quads of units, each of the four samplers in R600 is tied to
one of those quads, and then across the whole shader core.

Each unit has access to both the 32KB vertex and L1 texture cache, which is said to
improve throughput on unfiltered texture reads. In addition, AMD has implemented a
shared 256KB L2 texture cache. Both caches are fully associative. It allows the chip to
catch and store very large textures and pixels (that are too large for the L1 cache) locally
in order to save bandwidth. Besides caching algorithms, performance hits due to not very
high texturing speed can be avoided by means of the dynamic dispatcher and features like
Fetch4 (it accelerates fourfold the sampling of one-component textures from adjacent
addresses), which is the reason you see the four teensy additional texture address
processors and texture samplers in the diagram. This additional capability to grab data
from memory can be useful for certain tasks like shadowing or stream computing applications.

### 4.1.7 The memory controller

With the Radeon X1000-series (based on R5xx GPU series), ATI proposed a Ring Bus Memory Architecture to support an efficient memory allocation. The controller was called a Ring Bus because data travelled around two “rings” (which only handled reads) to four different stops each with two 32-bit memory channels attached to them. Write capabilities were still handled in the traditional way using a more conventional centralised crossbar switch. R600’s memory controller is an evolution of the one included in the X1000-series GPUs and it makes R600 the first GPU that features a 512-bit memory bus width. This means you can get more bandwidth from existing “mainstream” memory chips, and eventually it’ll deliver silly amounts of bandwidth with high speed GDDR4.

The design moves away from the centralised memory controller featured in R580 and uses a fully distributed memory controller with an independent direct memory access (DMA) unit that manages the ring stops on the bus. Like in R580, each ring stop is attached to a pair of memory channels, but they’re beefed up from 32-bit to 64-bit channels (externally). Internally, the ring bus has bi-directional 512-bit read and write rings so that the data can take the shortest possible route and each stop is connected to adjacent ring stops (one either side) via a 256-bit connection. Even the PCI-Express bus is treated as a ring stop this time around – it’s a source of more memory if the GPU requires it and is essentially an extension of HyperMemory. The DMA unit handles the traffic going across the PCI-Express bus in exactly the same way as it handles data going to and from local memory.

The memory read sequence in a ring bus architecture is:
1. Memory client makes read requests to memory controller.
2. Controller gathers and prioritizes requests, then sends them to memory devices.
3. Memory devices place requested data on the ring bus.
4. Clients pull their requested data off of the ring bus.

The arbitration logic in the Ring Bus Memory Controller is much more sophisticated than that used in previous GPU designs. It uses a feedback system that collects a variety of information from each memory client when it makes a read or write request. This data is run through an algorithm that calculates a priority value to assign to each request. The arbitration logic also tracks how successful the priority value calculations are over time by monitoring how long each client is kept waiting for data, and identifying cases where delays cause performance bottlenecks to develop. This information is then fed back into future priority calculations using weighting parameters to improve their accuracy.

In total there are 84 read clients and 70 write clients spread across the rings. Inside the ring stops, both the read clients and the ring stop itself are connected to a crossbar, which then feeds data to an arbiter before either going back into the ring stop, or moving into local memory. The write clients send data to a multiplexer before forwarding onto the dedicated write arbiter.
The data keeps flowing until it finds the right ring stop which means that there is a potentially large latency problem. However, if you take the fact that there are many threads in flight at any one time into account, latency shouldn't be as much of an issue as it could be. Ultimately, bandwidth is more important than latency in this case, simply because of the amount of data that is flowing around the GPU at any one time – latency can be hidden with efficient scheduling and multiplexing, but a lack of bandwidth can't.
5 The Nvidia G80 Architecture

One the Key features of the GeForce 8800 architecture is the use of numerous scalar stream processors (SPs) to perform shader operations. Stream processors are highly efficient computing engines that perform calculations on an input stream, while producing an output stream that can be used by other stream processors. Stream processors can be grouped in close proximity, and in large numbers, to provide parallel processing power.

Generally, specialized high-speed instruction decode and execution logic is built into a stream processor, and similar operations are performed on the different elements of a data stream. On-chip memory is typically used to store output of a stream processor, and the memory can be quickly read as input by other stream processors for subsequent processing. SIMD (single instruction/multiple data) instructions can be implemented across groupings of stream processors in an efficient manner, and massively parallel stream processor clusters are well-suited for processing graphics data streams.

The following figure shows a collection of stream processors (SPs) with associated numbers of Texture Filtering (TF), Texture Addressing (TA), and a L1 cache shared between 16 SPs in each block, essentially allowing these 16 units to communicate with each other.

The ratios of unit types shown below in a subset slice of a typical GeForce 8800 GPU are maintained when scaling up to 128 SPs specifically in a GeForce 8800 GTX GPU.

Each GeForce 8800 GPU stream processor is fully generalized, fully decoupled, scalar, can dual-issue a MAD and a MUL, and supports IEEE 754 floating-point precision. The stream processors are a critical component of NVIDIA GigaThread technology, where thousands of threads can be in flight within a GeForce 8800 GPU at any given instant. GigaThread technology keeps SPs fully utilized by scheduling and dispatching various types of threads (such as pixel, vertex, geometry, and physics) for execution.
All stream processors are driven by a high-speed clock domain (1.35 GHz) that is separate from the core clock (575 MHz) that drives the rest of the chip.

5.1 Scalar processor design

Leading GPUs to date have used vector processing units because many operations in graphics occur with vector data (such as R-G-B-A components operating in pixel shaders or 4×4 matrices for geometry transforms in vertex shaders). However, many scalar operations also occur. During the early GeForce 8800 architecture design phases, NVIDIA engineers analyzed hundreds of shader programs that showed an increasing use of scalar computations. They realized that with a mix of vector and scalar instructions, especially evident in longer, more complex shaders, it’s hard to efficiently utilize all processing units at any given instant with a vector architecture. Scalar computations are difficult to compile and schedule efficiently on a vector pipeline.

NVIDIA and ATI vector-based GPUs have used shader hardware that permits dual instruction issue. Recent ATI GPUs use a “3+1” design, allowing a single issue of a four-element vector instruction, or a dual issue of a three-element vector instruction plus a scalar instruction. NVIDIA GeForce 6x and GeForce 7x GPUs are more efficient with 3+1 and 2+2 dual-issue design. But they are still not as efficient as a GeForce 8800 GPU scalar design, which can issue scalar operations to its scalar processors with 100 percent shader processor efficiency. NVIDIA engineers have estimated as much as 2× performance improvement can be realized from a scalar architecture that uses 128 scalar processors versus one that uses 32 four-component vector processors, based on architectural efficiency of the scalar design.

This is obtained by converting vector-based shader program code to scalar operations, inside a the GPU to ensure complete efficiency.

5.2 Decoupled Shader Math and Texture Operations

Texture addressing, fetching, and filtering can take many GPU core clock cycles. If an architecture requires a texture to be fetched and filtered before performing the next math operation in a particular shader, the considerable texture fetch and filtering (such as 16× anisotropic filtering) latencies can really slow down a GPU. GeForce 8800 GPUs can do a great job tolerating and essentially “hiding” texture fetch latency by performing a number of useful independent math operations concurrently.

For comparison, a GeForce 7 Series GPU texture address calculation was interleaved with shader floating-point math operations in Shader Unit 1 of a pixel pipeline. Although this design was chosen to optimize die size, power, and performance, it could cause some shader math bottlenecks when textures were fetched, preventing use of a shader processor until the texture was retrieved. GeForce 8800 GPUs attacks the shader math and texture processing efficiency problem by decoupling shader and texture operations so that texture operations can be performed independent of shader math operations.

The following figure illustrates math operations (not dependent on specific texture data being fetched) that can be executed while one or more textures are being fetched from frame buffer memory, or in the worst case, from system memory. Shader processor utilization is improved. In effect, while a thread fetching a texture is executing, the GeForce 8800 GPU’s GigaThread technology can swap in other threads to execute, ensuring that shader processors are never idle when other work needs to be done.
5.3 Branching Efficiency Improvements

An important aspect of overall GPU performance in processing complex DirectX 10 shader workloads is branch efficiency. For background and comparison, GeForce 7 Series GPUs were designed to be efficient when processing typical DirectX 9 shaders. When an if-then-else statement was encountered in pixel shader code, a batch of 880 pixels was processed at once. Some of the pixels in the batch would generally require pixel shading effects applied based on the “then” code path, while the other pixels in the batch just went along the same code path for the ride, but were effectively masked out of any operations. Then the whole batch would take the “else” code path, where just the opposite would occur, and the other set of pixels would respond to the “else” code, while the rest went along for the ride.

GeForce 8800 Series GPUs are designed to process complex DX10 shaders. Programmers will enjoy as fine 16-pixel branching granularity up to 32 pixels in some cases. Compared to the ATI X1900 series, which uses 48 pixel granularity, the GeForce 8800 architecture is far more efficient with 32 pixel granularity for pixel shader programs.

5.4 Raw Processing and Texturing Filtering Power

Each stream processor on a GeForce 8800 GTX operates at 1.35 GHz and supports the dual issue of a scalar MAD and a scalar MUL operation, for a total of roughly 520 gigaflops of raw shader horsepower. But raw gigaflops do not tell the whole performance story. Instruction issue is 100 percent efficient with scalar shader units, and the mixed scalar and vector shader program code will perform much better compared to vector-based GPU hardware shader units that have instruction issue limitations (such as 3+1 and 2+2).

Texture filtering units are fully decoupled from the stream processors and deliver 64 pixels per clock worth of raw texture filtering horsepower (versus 24 pixels in the GeForce 7900 GTX); 32 pixels per clock worth of texture addressing; 32 pixels per clock of 2× anisotropic filtering; and 32-bilinear-filtered pixels per clock.

In essence, full-speed bilinear anisotropic filtering is nearly free on GeForce 8800 GPUs. FP16 bilinear texture filtering is also performed at 32 pixels per clock (about 5× faster than GeForce 7x GPUs), and FP16 2:1 anisotropic filtering is done at 16 pixels per clock. Note that the texture units run at the core clock, which is 575 MHz on the GeForce 8800 GTX.

At the core clock rate of 575 MHz, texture fill rate for both bilinear filtered texels and 2:1 bilinear anisotropic filtered texels is 575 MHz × 32 = 18.4 billion texels/second.
However, 2:1 bilinear anisotropic filtering uses two bilinear samples to derive a final filtered texel to apply to a pixel. Therefore, GeForce 8800 GPUs have an effective 36.8 billion texel/second fill rate when equated to raw bilinear texture filtering horsepower.

5.5 ROP and Memory Subsystems

The GeForce 8800 GTX has six Raster Operation (ROP) partitions, and each partition can process 4 pixels (16 subpixel samples, as shown in the diagram) for a total of 24 pixel/clock output capability with color and Z processing. For Z-only processing, an advanced new technique allows up to 192 samples/clock to be processed when a single sample is used per pixel. If 4× multisampled antialiasing is enabled, then 48 pixels per clock Z-only processing is possible.

The GeForce 8800 ROP subsystem supports multisampled, supersampled, and transparency adaptive antialiasing. Most important is the addition of four new single-GPU antialiasing modes (8×, 8×Q, 16×, and 16×Q), which provide the absolute best antialiasing quality for a single GPU in the market today.

The ROPs also support frame buffer blending of FP16 AND FP32 render targets, and either type of FP surface can be used in conjunction with multisampled antialiasing for outstanding HDR rendering quality. Eight MRTs (Multiple Render Targets) can be utilized, which is also supported by DX10. Each of the MRTs can define different color formats. New high-performance, more efficient compression technology is implemented in the ROP subsystem to accelerate color and Z processing.

As shown in Figure 10 six memory partitions exist on a GeForce 8800 GTX GPU, and each partition provides a 64-bit interface to memory, yielding a 384-bit combined interface width. The 768 MB memory subsystem implements a high-speed crossbar design, similar to GeForce 7x GPUs, and supports DDR1, DDR2, DDR3, GDDR3, and GDDR4 memory. The GeForce 8800 GTX uses GDDR3 memory default clocked at 900 MHz. With a 384-bit (48 byte wide) memory interface running at 900 MHz (1800 MHz DDR data rate), frame buffer memory bandwidth is very high at 86.4 GBps. With 768 MB of frame buffer memory, far more complex models and textures can be supported at high resolutions and image quality settings.

5.6 Power consumption and heat

The GeForce 8800 GTX is obviously a very power-hungry part, as you might have guessed since it requires dual power connectors. NVIDIA tells that the TDP of the card is around 185W. NVIDIA also recommends a 450W true power supply to support the GeForce 8800 GTX but sometimes it gets unstable on cheap brands ones. Temperature of the core is around 65° on idle and get up to 90° on heavy load.
High-performance processors
6 The CELL processor

The Cell Broadband Engine implements a single-chip multiprocessor with nine processors. The function of the processor elements is specialized into two types: the Power processor element (PPE) is optimized for control tasks and the eight synergistic processor elements (SPEs) provide an execution environment optimized for data processing. The design goals of the SPE and its architectural specification were to optimize for a low complexity, low area implementation. The PPE is built on IBM’s 64-bit Power Architecture with 128-bit vector media extensions and a two-level on-chip cache hierarchy. It is fully compliant with the 64-bit Power Architecture specification and can run 32-bit and 64-bit operating systems and applications. The SPEs are independent processors, each running an independent application thread. The SPE design is optimized for computation-intensive applications. Each SPE includes a private local store for efficient instruction and data access, but also has full access to the coherent shared memory, including the memory-mapped I/O space. Both types of processor cores share access to a common address space, which includes main memory, and address ranges corresponding to each SPE’s local store, control registers, and I/O devices.

6.1 Power Processing Element

The PPE is a 64-bit Power-Architecture compliant core able to run 2 threads simultaneously, in fact when one thread is stalled and is waiting for data the second thread can issue instructions keeping the instruction units busy. The PPE is not the primary processor, it is used as controller and it controls and synchronizes all the SPUs which must execute the greater part of the operations. It will run the operating system and...
most of the applications but highly intensive parts of the OS and applications will be offloaded to the SPEs.
The PPE is a dual issue, dual threaded, in-order processor which does not dynamically reorder instructions at issue time. The core interleaves instructions from two computational threads at the same time to optimize the use of issue slots, maintain maximum efficiency, and reduce pipeline depth.
The hardware architecture is an “old style” RISC design which uses considerably less power than other PowerPC devices, even at higher clock rates. Its pipeline depth is only 23 stages, significantly less than what one might expect for a design that reduces the amount of time per stage by nearly a factor of 2.
The PPE includes support for the VMX vector instructions, (also known as "AltiVec" or "Velocity Engine").
The PPE supports a conventional cache hierarchy with 32-KB first-level instruction and data caches and a 512-KB second-level cache. The second-level cache and the address-translation caches use replacement management tables to allow the software to direct entries with specific address ranges at a particular subset of the cache. This mechanism allows for locking data in the cache and can also be used to prevent overwriting data in the cache by directing data that is known to be used only once at a particular set. Providing these functions enables increased efficiency and increased real-time control of the processor.
The processor provides two simultaneous threads of execution within the processor and can be viewed as a two-way multiprocessor with shared dataflow.
All architected states are duplicated, including all architected registers and special-purpose registers, with the exception of registers that deal with system-level resources, such as logical partitions, memory, and thread control. Non-architected resources such as caches and queues are generally shared for both threads.
The processor is composed of three units:

1. The instruction unit (IU) is responsible for instruction fetch, decode, branch, issue, and completion.
2. A fixedpoint execution unit (XU) is responsible for all fixedpoint instructions and all load/store-type instructions.
3. A vector scalar unit (VSU) is responsible for all vector and floating-point instructions.

The IU fetches four instructions per cycle per thread into an instruction buffer and dispatches the instructions from this buffer. After decode and dependency checking, instructions are dual-issued to an execution unit. A 4-KB by 2-bit branch history table with 6 bits of global history per thread is used to predict the outcome of branches. The IU can issue up to two instructions per cycle. Simple vector, complex vector, vector floating-point, and scalar floating-point arithmetic cannot be dual-issued with the same type of instructions. However, these instructions can be dual-issued with any other form of load/store, fixed-point branch, or vector-permute instruction.
A VSU issue queue decouples the vector and floating-point pipelines from the remaining pipelines. This allows vector and floating-point instructions to be issued out of order with respect to other instructions.
The XU consists of a 32- by 64-bit general-purpose register file per thread, a fixed-point execution unit, and a load/store unit. The load/store unit consists of the L1 D-cache, a translation cache, an eight-entry miss queue, and a 16-entry store queue. The load/store unit supports a non-blocking L1 D-cache which allows cache accesses under misses. The VSU floating-point execution unit consists of a 32- by 64-bit register file per thread, as well as a ten-stage double-precision pipeline. The VSU vector execution units are organized around a 128-bit dataflow. The vector unit contains four subunits: simple, complex, permute, and single-precision floating point. There is a 32-entry by 128-bit vector register file per thread, and all instructions are 128-bit SIMD with varying element width (2-elements by 64-bit, 4-elements by 32-bit, 8-elements by 16-bit, 16-elements by 8-bit, and 128-elements by 1-bit).
6.2 Synergistic Processor Element

The SPE implements a new instruction-set architecture optimized for power and performance on computing-intensive and media applications. Each SPE is a 128 bit VLIW vector processor (SIMD), it can compute 2 instruction simultaneously, it is capable of 4 X 32 bit operations per cycle and it has 256 Kbyte of local memory (called LS, Local Storage) at high speed. Like the PPE the SPEs are in-order processors and have no Out-Of-Order capabilities. Each SPE contains 128 x 128 bit registers, there are also 4 (single precision) floating point units.

The PPE and SPEs are highly integrated. The PPE provides common control functions, runs the operating system, and provides application control, while the SPEs provide the bulk of the application performance. The PPE and SPEs share address translation and virtual memory architecture, and provide support for virtualization and dynamic system partitioning. They also share system page tables and system functions such as interrupt presentation. Finally, they share data type formats and operation semantics to allow efficient data sharing among them. Each SPE consists of the SPU and the synergistic memory flow (SMF) controller. The SMF controller moves data and performs synchronization in parallel to SPU processing and implements the interface to the element interconnect bus, which provides the Cell BE with a modular, scalable integration point.

(a) Synergistic Processor Architecture (b) Pipeline elements
6.2.1 Local Store
The local store supports both 128-byte access from direct memory access (DMA) read and write, as well as instruction fetch, and a 16-byte interface for load and store operations. The Local Storage memory is not the cache memory, so there isn’t hardware management of faults and each operation of load, prefetch or update in the memory has to be done explicitly by the code, the compiler or the programmer. They are in effect a second-level register file. The SPEs operate on registers which are read from or written to the local stores. The local stores can access main memory in blocks of 1Kb minimum (16Kb maximum) but the SPEs cannot act directly on main memory. LS memory is visible to the SPE unit which can load instruction and data in it. Every SPU is able to approach the local memory of another SPU quickly and therefore when a SPU has completed the elaborations another SPU can capture the data for successive elaborations.

6.2.2 Memory flow control
Data and instructions are transferred between this local memory and system memory by asynchronous coherent DMA commands, executed by the memory flow control unit included in each SPE. Addresses can be passed between the PPE and SPEs, and the operating system can share memory and manage all of the processing resources in the system in a consistent manner. The local store is mapped into the memory map of the processor, but this memory (if cached) is not coherent in the system.
A single-port SRAM cell is used to minimize area. In order to provide good performance the local store was designed with both narrow (128-bit) and wide (128-byte) read and write ports. The wide access is used for DMA reads and writes as well as instruction (pre)fetch. Because a typical 128-byte DMA read or write requires 16 processor cycles to place the data on the on-chip coherent bus, even when DMA reads and writes occur at full bandwidth, seven of every eight cycles remain available for loads, stores, and instruction fetch. Similarly, instructions are fetched 128 bytes at a time, and pressure on the local store is minimized. The highest priority is given to DMA commands, the next highest priority to loads and stores, and instruction (pre)fetch occurs whenever there is a cycle available. A special no-operation instruction exists to force the availability of a slot to instruction fetch when necessary.
6.3 **Element Interconnect Bus**

Four 16Byte bus data rings connecting 12 bus elements. In that case all processors element overlaps. The central arbiter supports up to three concurrent transfers per data ring. Each element port simultaneously supports 16Byte in and 16Byte out data path. The EIB is a communication bus internal to the Cell processor which connects the various on-chip system elements: the PPE, the memory controller (MIC), the eight SPE, and two I/O interfaces, for a total of 12 participants. The EIB also includes a centralized unit to perform arbitration issues on the interconnect.

The EIB is implemented as a circular ring and is comprised of four 16B-wide unidirectional channels. Each channel can convey up to three transactions concurrently and runs at half the system clock rate. Supposing the maximum concurrency, with three active transactions on each of the four rings, the peak instantaneous EIB bandwidth is 96B per clock (12 concurrent transactions * 16 bytes wide / 2 system clocks per transfer).
6.4 Memory and I/O

All the internal processing units need to be fed so a high speed memory and I/O system is an absolute necessity. For this purpose Sony and Toshiba licensed the high speed "Yellowstone" and "Redwood" technologies from Rambus, these are used in the XDR RAM and FlexIO.

Both of these are interesting technologies not only for their raw speed but they have also been designed to simplify board layouts. Engineers spend a lot of time making sure wires on motherboards are all exactly the same length so signals are synchronized. FlexIO and XDR RAM both have a technology called "FlexPhase" which allow signals to come in at different times reducing the need for the wires to be exactly the same length, this will make life considerably easier for board designers working with the Cell.

As with everything else in the Cell architecture the memory system is designed for raw speed, it will have both low latency and very high bandwidth. As mentioned previously the SPEs access memory in blocks of 128 bytes.

The Cell will use high speed XDR RAM for memory. A Cell has a memory bandwidth of 25.6 Gigabytes per second which is considerably higher than any PC but necessary as the SPEs will eat as much memory bandwidth as they can get. Even given this the buses are not large (72 data pins in total), this is important as it keeps chip manufacturing costs down. The Cells runs it’s memory interface at 3.2 Gigabits per second per pin though memory in production now is already capable of higher speeds than this. XDR is designed to scale to 6.4 Gigabits per second so memory bandwidth has the potential to double.
7 The Intel Core 2 Duo
The Intel Core Microarchitecture is a multi-core processor microarchitecture based on an updated version of the Pentium M core and could be considered the latest evolutionary step of the Intel P6 platform. After the P6 platform the Intel developed another one called NetBurst, however during the 2005 Intel Developer Forum, Intel formally announced that they would be dropping the Pentium 4's Netburst microarchitecture in favor of a more power-efficient microarchitecture for the entire line of processors.

Power saving features are getting more and more important for both mobile and server segments in order to improve battery life in the first one and to reduce costs of maintainance in the second. That's why the main guide lines in developing this architectures take into account first a particularly low power consumption and then the multiple core architecture, the hardware virtualization support and SSE3 set of instructions. First processors that use this architecture are Merom, Conroe and Woodcrest; Merom is for the mobile computing, Conroe for the desktop systems and Woodcrest for servers and workstations. These three type of cores differs mainly for clockspeed, cache size and power consumption.

7.1 Architecture
First of all, we have to underline that the Pentium 4's performances was designed to scale primarily with clockspeed increase. In contrast, Core's performances will scale primarily with increases in the number of Cores per die and in the increment of caches size and secondarily with modest increase of clockspeed.

Intel 64 architecture increases the linear address space for software to 64 bits and supports physical address space up to 40 bits. The technology also introduces a new operating mode referred to as IA-32e mode. IA-32e mode operates in one of two sub-modes: (1) compatibility mode enables a 64-bit operating system to run most legacy 32-bit software unmodified, (2) 64-bit mode enables a 64-bit operating system to run applications written to access 64-bit address space.

An Intel 64 architecture processor supports existing IA-32 software because it is able to run all non-64-bit legacy modes supported by IA-32 architecture. Most existing IA-32 applications also run in compatibility mode.

A great distinctive features of the P6 was the issue port structure. One of the main problems of this arrangement was that with the addition of other execution unit in the latest version of this architecture (maintaining the same number of ports) two of the 5 ports in particular became overcrowded. To overcome this situation, in addition to an enlarged reservation station (32 entries) and an increased number of execution units Intel Core add an additional port to the execution core. The six-port scheme permit to have 3 issue port dedicated to arithmetic and logic instructions with respect to the previous 2 of the Pentium III allowing up to 6 instructions per cycle (without taking into account the Macro-fusion feature).
7.2 Advanced Digital Media Boost

When the SSE 128 bit instructions was introduced in the Pentium line, the results weren’t as pretty as programmers hoped because the datapath of the processor remained 64 wide. The decoder first splits the instruction into two, 64-bit micro-ops, one for the upper 64 bits of the vector and another for the lower 64 bits. Then this pair of micro-ops is passed to the appropriate SSE unit for execution. The result of this hack is that all 128-bit vector operations take a minimum of two cycles to execute on the P6.

Intel decide to overcome this problem by making the floating-point and vector internal data buses 128 bits wide, a feature that also means only a single micro-op needs to be generated, dispatched, scheduled, and issued for each 128-bit vector operation.

7.3 Instruction decoding

The P6 instruction Decoding section were composed by a set of 2 simple/fast and 1 complex/slow decoder. The majority of the x86 instructions were decoded by simple decoders into one micro-ops each one, however the rest could be decoded only by the complex decoder into 2 to 4 micro-ops. By this way the P6 core's three decoders can output a maximum of six micro-ops per cycle into the micro-op buffer, and the decoding unit as a whole can send up to three micro-ops per cycle on to the ROB.

Since the Core's execution core have been widened considerably and more micro-ops/cycle could reach the back end, Intel decides to increase the decode rate by adding one more simple decoder.
Another interesting feature of the Core platform is the ability to fuse together two instructions. This feature called Macro-fusion could be applied only at a certain type of instructions such as the compare-and-test and the branch instructions. Any of the decoders can generate this kind of fusion but only one can be generated per cycle. This reduction of two instructions to only one permit also to reduce the impact on the buffers and to increase the bandwidth of instructions executed per cycle. Another technique that has similar effect as Macro-fusion but functioning differently is the Micro-fusion. This consists of a couple of micro-ops that are tracked by the Reorder Buffer as only one. Like macro-fusion, micro-ops fusion enables the ROB to issue and commit more micro-ops using fewer entries and less hardware. These two features allow also to reduce power consumption.

### 7.5 Smart Memory Access

Level 1 Data Memory is highly out-of-order and uses a variety of methods to achieve its performance. Included among these are Intel Smart Memory Access and its two key features: memory disambiguation and instruction pointer based (IP-based) prefetcher to the level 1 data cache. Typical x86 software code contains about 38 percent memory stores and loads. Generally the number of loads are twice the number of stores. To prevent data inconsistency, dependent memory-related instructions are normally executed in the same order they appear on the program.
An interesting solution has been developed to well support the case of loads and stores that don't share the same address. For example, I cannot try to move a load in the execution order above a store having an unknown address because they could refer the same memory location. However, many loads are location unrelated to recently executed store with the risk of detecting a false dependence. In order to prevent such a false dependence, Core's microarchitecture includes a technique called Memory Disambiguation. This feature uses a predictor and an appropriate algorithm to eliminate these false dependencies. The basic objective is to be able to ignore unknown store-address blocking conditions whenever a dispatched load operation is predicted to not collide with a store checking all the dispatched store addresses. This prediction is then adjusted by checking all addresses of dispatched store instructions to find if a store instruction (that in program order would precede the load) has written the same memory address as the load instruction. If there is a conflict, the pipe is flushed and the execution restarts from the load instruction.

The memory disambiguation predictor is based on a hash table that is indexed with a hashed version of the load IP address bits. Each predictor entry is a counter able to reset that saturate at a certain value. During the load's retirement we could:

- increment the counter (no increment if the saturation value is achieved) in the case it met unknown store addresses but none of them collided
- reset the counter if it collided with at least one older store dispatched after the load

Note that it requires that a certain number of consecutive iterations of a load having the same IP behave well.

The predictor is looked up when a load instruction is dispatched from the RS to the pipe. If the relative counter is saturated the load instruction can be considered safe and then discard unknown store-address blocking condition. If, on the other hand, the counter is not saturated yet and there is a relevant unknown store address the load is blocked. Since mispredictions can cause pipe flush, the Core microarchitecture includes a mechanism to temporarily turn off memory disambiguation to prevent performance losses. This mechanism is based on the constant monitoring of the rate of success of the predictor.

Another important feature part of the Smart Memory Access is the Instruction Pointer-Based Prefetcher to Level 1 Data Cache. A prefetch is a unit that prefetches memory data before requested, placing this data in cache for “just-in-time” execution. Of particular interest is the IP-based prefetcher that prefetches data to the Level 1 data cache. The purpose of the IP prefetcher is to predict what memory addresses are going to be used by the program and delivered data just in time. To improve the accuracy of the prediction, the IP prefetcher tags an history array of the loads instructions with the IP of the loads. Based on this history, the prefetcher tries to predict the address of the next load accordingly to a fixed distance between subsequent access to the same memory area. The prefetcher then sends a request with the predicted address and brings the resulting data to the Level 1 Data cache.
However there could be possible side effects such as overloading of resources becoming from the excessive use of this tool. First of all the prefetch request goes to a FIFO buffer where the requests waits for “favorable conditions”. In addition, in order to manage all the eight prefetch units Core microarchitecture uses a prefetch monitors with multiple watermark mechanism to control possible overloads. The result is a good trade-off between responsiveness to program needs and utilization of the bandwidth.
8 The Intel Itanium

The Itanium processor is the first implementation of the IA-64 instruction set architecture (ISA); it has the support for 64-bit addressing, reliability for mission-critical applications, full IA-32 instruction set compatibility in hardware, and scalability across a range of operating systems and platforms.

The processor employs EPIC (explicitly parallel instruction computing) design concepts, these constructs provide powerful architectural semantics and enable the software to make global optimizations across a large scheduling scope, thereby exposing available instruction-level parallelism (ILP) to the hardware. Through its use of EPIC technology, the processor fundamentally shifts the balance of responsibilities between software and hardware. The software performs global scheduling across the entire compilation scope, exposing ILP to the hardware. The hardware provides abundant execution resources, manages the bookkeeping for EPIC constructs, and focuses on dynamic fetch and control flow optimizations to keep the compiled code flowing through the pipeline at high throughput. The tighter coupling and increased synergy between hardware and software enable higher performance with a simpler and more efficient design.

The processor provides a six-wide and 10-stage deep pipeline, running at 800 MHz on a 0.18 micron process. This combines both abundant resources to exploit ILP and high frequency for minimizing the latency of each instruction. The resources consist of four integer units, four multimedia units, two load/store units, three branch units, two extended-precision floating-point units, and two additional single-precision floating-point units (FPUs).

The system bus provides glueless multiprocessor support for up to four-processor systems and can be used as an effective building block for very large systems. The advanced FPU delivers over 3 Gflops of numeric capability.

In contrast to traditional processors, the machine’s core is characterized by hardware support for the key ISA constructs that embody the EPIC design style. This includes support for speculation, predication, explicit parallelism, register stacking and rotation, branch hints, and memory hints.

8.1 EPIC support in the Itanium

The Itanium processor introduces a number of unique microarchitectural features to support the EPIC design style.

These features focus on the following areas:
- supplying plentiful fast, parallel, and pipelined execution resources, exposed directly to the software;
- supporting the bookkeeping and control for new EPIC constructs such as predication and speculation;
- providing dynamic support to handle events that are unpredictable at compilation time so that the compiled code flows through the pipeline at high throughput.

The core of the machine is the wide execution engine, designed to provide the computational bandwidth needed by ILP-rich EPIC code that abounds in speculative and predicated operations.
The execution control is augmented with a bookkeeping structure called the advanced load address table (ALAT) to support data speculation and, with hardware, to manage the deferral of exceptions on speculative execution. Operands are fed into this wide execution core from the 128-entry integer and floating-point register files. The register file addressing undergoes register remapping, in support of register stacking and rotation. An instruction dispersal network feeds the execution pipeline and a decoupled fetch engine exploits advanced prefetch and branch hints to ensure that the fetched instructions will come from the correct path and that they will arrive early enough to avoid cache miss penalties. Finally, memory locality hints are employed by the cache subsystem to improve the cache allocation and replacement policies, resulting in a better use of the three levels of on-package cache and all associated memory bandwidth. EPIC features allow software to more effectively communicate high-level semantic information to the hardware, thereby eliminating redundant or inefficient hardware and leading to a more effective design. Using EPIC constructs, the compiler optimizes the code schedule across a very large scope, the result is an EPIC machine in which the close collaboration of hardware and software enables high performance with a greater degree of overall efficiency.

Three important aspects characterize the core pipeline of the Itanium:

- wide EPIC hardware delivering a new level of parallelism (six instructions/clock): The processor provides hardware for these execution units: four integer ALUs, four multimedia ALUs, two extended-precision floating-point units, two additional single precision floating-point units, two load/store units, and three branch units. For enterprise and commercial codes, the MII/MBB template combination in a bundle pair provides six instructions or eight parallel operations per clock (two load/store, two general-purpose ALU operations, two post increment ALU operations, and two branch instructions). Alternatively, an MIB/MIB pair allows the same mix of operations, but with one branch hint and one branch operation, instead of two branch operations. For scientific code, the use of the MFI template in each bundle enables 12 parallel operations per clock, for digital content creation codes that use single precision floating point, the SIMD (single instruction, multiple data) features in the machine effectively enable up to 20 parallel operations per clock;

- deep pipelining (10 stages) enabling high frequency of operation: The cycle time and the core pipeline are balanced and optimized for the sequential execution in integer scalar codes, by minimizing the latency of the most frequent operations, thus reducing dead time in the overall computation. The high frequency (800 MHz) and careful pipelining enable independent operations to flow through this pipeline at high throughput.
8.2 The Core Pipeline

Given the high execution rate of the processor (six instructions per clock), an aggressive front end is needed to keep the machine effectively fed, especially in the presence of disruptions due to branches and cache misses. Acting in conjunction with sophisticated branch prediction and correction hardware, the machine speculatively fetches instructions from a moderate-size, pipelined instruction cache into a decoupling buffer. Software-initiated prefetch probes for future misses in the instruction cache and then prefetches such target code from the level-2 (L2) cache into a streaming buffer and eventually into the instruction cache.

The three important characteristics are implemented in order to achieve good performance:

- Speculative fetches: The 16-Kbyte, four-way set-associative instruction cache is fully pipelined and can deliver 32 bytes of code (two instruction bundles or six instructions) every clock. The fetched code is fed into a decoupling buffer that can hold eight bundles of code. The buffer can continue to feed the back end even when the front end is disrupted by fetch bubbles due to branches or instruction cache misses.

- Hierarchy of branch prediction: The processor employs a hierarchy of branch prediction structures to deliver high-accuracy and low-penalty predictions across a wide spectrum of workloads (if a branch misprediction led to a full pipeline flush, there would be nine cycles of pipeline bubbles before the pipeline is full again).
The branch prediction hardware is assisted by branch hint directives provided by the Compiler, the directives provide branch target addresses, static hints on branch direction, as well as indications on when to use dynamic prediction. These directives are programmed into the branch prediction structures and used in conjunction with dynamic prediction schemes. The machine provides up to four progressive predictions and corrections to the fetch pointer:

- Reester 1 - Single-cycle predictor: A special set of four branch prediction registers (called target address registers, or TARs) provides single-cycle turnaround on certain branches (for example, loop branches in numeric code), operating under tight compiler control. When the instruction pointer of the candidate branch hits in these registers, the branch is predicted taken, and these registers provide the target address for the reester. On such taken branches no bubbles appear in the execution schedule due to branching.

- Reester 2 – Adaptive multiway and return prediction: For scalar codes, the processor employs a dynamic, adaptive, two-level prediction scheme to achieve well over 90% prediction rates on branch direction. The branch prediction table (BPT) contains 512 entries (128 sets x 4 ways). Each entry, selected by the branch address, tracks the four most recent occurrences of that branch.

- Reester 3 and 4 – Branch address calculation and correction: Once branch instruction opcodes are available (ROT stage), it’s possible to apply a correction to predictions made earlier. The BAC1 stage applies a correction for the exit condition on modulo-scheduled loops through a special “perfect-loop-exit-predictor” structure that keeps track of the loop count extracted during the loop initialization code; in case of misses in the earlier prediction structures, BAC1 extracts static prediction information and addresses from branch instructions in the rightmost slot of a bundle and uses these to provide a correction.

- Software-initiated prefetch: Another key element of the front end is its software-initiated instruction prefetch. Prefetch is triggered by prefetch hints (encoded in the BRP instructions as well as in actual branch instructions) as they pass through the ROT stage. Instructions get prefetched from the L2 cache into an instruction-streaming buffer (ISB). The instruction cache filters prefetch requests. The cache tags and the TLB have been enhanced with an additional port to check whether an address will lead to a miss. Such requests are sent to the L2 cache. To fully hide the latency of returns from the L2 cache, BRP instructions that initiate prefetch should be hoisted 12 fetch cycles ahead of the branch, this kind of hoisting of BRP instructions provide a mechanism for the compiler to eliminate instruction fetch bubbles.

After instructions are fetched in the front end, they move into the middle pipeline that disperses instructions, implements the architectural renaming of registers, and delivers operands to the wide parallel hardware. The hardware resources in the back end of the machine are organized around nine issue ports. The instruction and operand delivery
hardware maps the six incoming instructions onto the nine issue ports and remaps the virtual register identifiers. It then provides the source data to the execution core. The dispersal and renaming hardware exploits high-level semantic information provided by the IA-64 software.

The instruction dispersal mechanism disperses instructions presented by the decoupling buffer to the processor’s issue ports. The processor has a total of nine issue ports capable of issuing up to two memory instructions (ports M0 and M1), two integer (ports I0 and I1), two floating-point (ports F0 and F1), and three branch instructions (ports B0, B1, and B2) per clock. The decoupling buffer feeds the dispersal in a bundle granular fashion (up to two bundles or six instructions per cycle), with a fresh bundle being presented each time one is consumed.

The dispersal algorithm is fast and simple, with instructions being dispersed to the first available issue port, subject to two constraints: detection of instruction independence and detection of resource oversubscription:

- Independence: The processor must ensure that all instructions issued in parallel are either independent or contain only allowed dependencies (such as a compare instruction feeding a dependent conditional branch);
- Oversubscription: The processor must also guarantee that there are sufficient execution resources to process all the instructions that will be issued in parallel. The IA-64 ISA feature of instruction bundle templates facilitates this oversubscription problem.

The dispersal strategy also guarantees that certain instruction types can only occur on specific locations within any bundle. As a result, the dispersal interconnection network can be significantly optimized; the routing required from dispersal to issue ports is roughly only half of that required for a fully connected crossbar.

After dispersal, the next step in preparing incoming instructions for execution involves implementing the register stacking and rotation functions. Register stacking is an IA-64 technique that significantly reduces function call and return overhead. It ensures that all procedural input and output parameters are in specific register locations, without requiring the compiler to perform register-register or memory-register moves. This enables low-overhead procedure calls, providing significant performance benefit on codes that are heavy in calls and returns, such as those in object-oriented languages.

Register rotation is an IA-64 technique that allows very low overhead, software-pipelined loops. It broadens the applicability of compiler-driven software pipelining to a wide variety of integer codes. Rotation provides a form of register renaming that allows every iteration of a software-pipelined loop to have a fresh copy of loop variables. This is accomplished by accessing the registers through an indirection based on the iteration count. Both stacking and rotation require the hardware to remap the register names.

The Itanium processor efficiently supports the register remapping for both register stacking and rotation with a set of adders and multiplexers contained in the pipeline’s REN stage.

The register-stacking model also requires special handling when software allocates more virtual registers than are currently physically available in the register file. A special state machine, the register stack engine (RSE), handles this case—termed stack overflow. This engine observes all stacked register allocation or deallocation requests. When an
overflow is detected on a procedure call, the engine silently takes control of the pipeline, spilling registers to a backing store in memory until sufficient physical registers are available. While these registers are being spilled or filled, the engine simply stalls instructions waiting on the registers; no pipeline flushes are needed to implement the register spill/restore operations.

Register stacking and rotation combine to provide significant performance benefits for a variety of applications, at the modest cost of a number of small adders, an additional pipeline stage, and control logic for a programmer-invisible register stack engine.

The execution core is the heart of the EPIC implementation. It supports data-speculative and control-speculative execution, as well as predicated execution and the traditional functions of hazard detection and branch execution. Furthermore, the processor’s execution core provides these capabilities in the context of the wide execution width and powerful instruction semantics that characterize the EPIC design philosophy. All of the control paths within the core pipeline fit within the given cycle time—detecting and dealing with data hazards was one such key control path, to achieve high performance, the strategy for dealing with detected data hazards is based on stalls—the pipeline only stalls when unavailable data is needed and stalls only as long as the data is unavailable. This strategy allows the entire processor pipeline to remain filled, and the in-flight dependent instructions to be immediately ready to continue as soon as the required data is available. This contrasts with other high-frequency designs, which are based on flushing and require that the pipeline be emptied when a hazard is detected, resulting in reduced performance. On the Itanium processor, innovative techniques reap the performance benefits of a stall-based strategy and yet enable high frequency operation on this wide machine. The scoreboard control is also enhanced to support predication.

The processor provides an abundance of execution resources to exploit ILP. The integer execution core includes two memory and two integer ports, with all four ports capable of executing arithmetic, shift and-add, logical, compare, and most integer SIMD multimedia operations. The memory ports can also perform load and store operations, including loads and stores with postincrement functionality.

Predication (instead of waiting for conditional branches (if/then statements) to be resolved, the processor can execute both branches simultaneously. x86 processors must try to predict the most likely outcome, and suffer a performance hit whenever a prediction is wrong) is another key feature of the IA-64 architecture, allowing higher performance by eliminating branches and their associated miss-prediction penalties, predication turns a control dependency (branching on the condition) into a data dependency (execution and forwarding of data dependent upon the value of the predicate). If spurious stalls and pipeline disruptions get introduced during predicated execution, the benefit of branch miss-prediction elimination will be squandered. Care was taken to ensure that predicates are implemented transparently in the pipeline.

Predicates are used to configure the data-forwarding network, detect the presence of hazards, control pipeline advances, and conditionally nullify the execution and retirement of issued operations. The predicate register file is a highly multiported structure. It is accessed in parallel with the general registers in the REG stage.
8.3 Memory Subsystem

The processor provides three levels of on-package cache for scalable performance across a variety of workloads. At the first level, instruction and data caches are split, each 16 Kbytes in size, four-way set-associative, and with a 32-byte line size. The dual-ported data cache has a load latency of two cycles, is write-through, and is physically addressed and tagged. The L2 cache uses a four-state MESI (modified, exclusive, shared, and invalid) protocol for multiprocessor coherence. The third level of on-package cache is 4 Mbytes in size, uses a 64-byte line size, and is four-way set-associative. It communicates with the processor at core frequency (800 MHz) using a 128-bit bus. This cache serves the large workloads of server- and transaction processing applications, and minimizes the cache traffic on the frontside system bus. The L3 cache also implements a MESI protocol for microprocessor coherence.

The IA-64 instruction set architecture defines a set of memory locality hints used for better managing the memory capacity at specific hierarchy levels. These hints indicate the temporal locality of each access at each level of hierarchy. The processor uses them to determine allocation and replacement strategies for each cache level. Additionally, the IA-64 architecture allows a bias hint, indicating that the software intends to modify the data of a given cache line. The bias hint brings a line into the cache with ownership, thereby optimizing the MESI protocol latency.

The processor uses a multidrop, shared system bus to provide four-way glueless multiprocessor system support. No additional bridges are needed for building up to a four-way system. Systems with eight or more processors are designed through clusters of these nodes using high-speed interconnects. The 64-bit system bus uses a source-synchronous data transfer to achieve 266 Mtransfers/s, which enables a bandwidth of 2.1 Gbytes/s. The combination of these features makes the Itanium processor system a scalable building block for large multiprocessor systems.